Diplomarbeit

Untersuchungen zur Parallelisierung
von BV-Algorithmen
für eingebettete Multicore-Systeme

Arne Garbade
1575139

Universität Bremen
Fachbereich 3
Informatik

15. Juni 2009

Gewidmet meinen lieben Eltern
Angelika und Martin Garbade
Danke für die tolle Unterstüzung

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit entsteht im Zuge einer Diplomarbeit bei Advanced Driver Information Technology GmbH (ADIT), einem Joint Venture zwischen der Robert Bosch GmbH und der japanischen DENSO CORPORATION. Die ADIT beschäftigt sich mit der Konzipierung und Entwicklung einer Systemplattform für Fahrerinformationssysteme. Ziel dieser Konzipierungen ist es, die verschiedenen informationsgebenen Geräte (Fahrerassistenz, Entertainment) zusammen in einen Gerät zu integrieren. Das Problem, vor dem Entwickler solcher Informationssysteme stehen, ist die Entwicklung leistungsfähiger Hardware-Plattformen die Implementierungen von konkurrenzfähigen Navigationssystemen erlauben. Darüber hinaus müssen diese Systeme auch den durch allgemeinen Fortschritt gewachsenen Ansprüchen in der Informationstechnologie der Endkunden gerecht werden.

Aktuelle Prozessoren wie der Intel Core i7 von der Intel Corporation (Intel) oder der AMD Opteron von Advanced Micro Devices, Inc. (AMD) haben zwar genug Leistung um die Informationen aus Navigation und Multimedia in adäquater Zeit zu verarbeiten. Der Einsatz im Automotiv-Bereich ist aus Kostengründen (Anschaffung) und der Wärmeentwicklung nicht geeignet.

Anders als Desktop Systeme sind eingebettete Systeme auf ihren Verwendungszweck hin optimiert und werden heute in den meisten mobilen Geräten eingesetzt. Die Wärmeentwicklung dieser Geräte ist so gering, dass keine aktive Kühlung erforderlich ist. Zusätzlich erlaubt die Einführung der Mehrkern-Prozessoren im Bereich der eingebetteten Systeme nun auch das parallele Verarbeiten größerer Datenmengen, wie sie im oben beschriebenen Automotive Bereich entstehen. Einer der rechen- und datenintensivsten Aufgaben eines Fahrerinformationssystems ist beispielsweise die Objekterkennung durch Bildverarbeitung (BV). Hier können mit Hilfe der BV-Algorithmen unter anderem Schilder im Straßenverkehr erkannt werden. Das Erkennen und die dafür erforderliche Berechnung muss jedoch in Echtzeit berechnet werden, das heißt, in hinreichend kurzer Zeit und mit korrektem Ergebnis. Das kann jedoch nur dann geschehen, wenn die Berechnungsroutinen auf alle Prozessorkerne verteilt werden. Im Einzelnen hat beispielsweise ein Kern des ARM11 MPCore von ARM, Ltd (ARM) 400Mhz. Eingebettete Mehrkern-Prozessoren wie der ARM11 MPCore erhalten ihre Gesamtleistung aber aus dem Verbund ihrer Kerne. Um die Objekterkennung realisieren zu können, muss die Leistungsfähigkeit dieser Mehrkern-Systeme optimal ausgenutzt werden.

Eine Möglichkeit alle Kerne des Systems parallel einzusetzen, ist der so genannte SMP-Betrieb (Symmetric Multiprocessing) eines Mehrkern-Systems. Wie im Abschnitt 2.3 noch näher erläutert wird, sind an einen SMP-Betrieb Bedingungen geküpft. Durch diese Bedingungen werden Voraussetzungen an Hardware, Betriebssystem und Software gestellt. Eine dieser Voraussetzungen ist, das Teile einer Software verteilt auf je einem Kernen parallel ausgeführt werden können.

In dieser Arbeit wird untersucht, wie diese Voraussetzungen erfüllt werden können. Hierfür soll zunächst der BV-Algorithmus SIFT (Abschnitt 2.4.4) von der seriellen Ausführung in eine parallel ausführbare Form geändert werden. Der SIFT ist ein rechen- und datenintensiver Algorithmus und daher sehr gut geeignet,

um Last auf das Testsystem auszuüben. Hierzu muss zunächst die BV-Bibliothek OpenCV (Abschnitt 2.4.3), auf dem der SIFT-Algorithmus aufsetzt, auf das von eSOL gepfelgte Betriebssystem eT-Kernel portiert werden. Im Anschluss daran wird der Algorithmus selbst auf eT-Kernel portiert.
Eine vollständige Implementierung des parallel ausführbaren Algorithmus hätte den zeitlichen Rahmen dieser Arbeit gesprengt. Daher wird eine theoretische Konzipierung der Implementation durchgeführt. Die reale Implementierung wird auf das Aufteilen der Eingabebilder und das auf alle Kerne verteilte Berechnen beschränkt. Diese Beschränkung wirkt sich jedoch auf das Ergebnis des SIFTs aus. Durch die Aufteilung der Bildbereiche gehen bei der Berechnung Informationen über das Bild verloren, so dass ein korrektes Ergebnis des Algorithmus nicht gewährleistet ist. Da jedoch bei den Untersuchungen nicht das Ergebnis des SIFT-Algorithmus den Schwerpunkt ausmacht, ist das vollständige Ergebnis des SIFT in dieser Arbeit nicht entscheidend.

Bei den Untersuchungen liegt auf den verschiedenen Scheduling-Verfahren des Betriebssystems eT-Kernel der Fokus, da diese Verfahren die Voraussetzung zum SMP an das Betriebssystem erfüllen sollen. Im Abschnitt 3 werden diese drei Scheduling-Verfahren genauer untersucht und einer Reihe von Tests unterzogen, in denen ermittelt werden soll, welches Leistungspotential jedes einzelne Verfahren im Vergleich zur seriellen Variante hat. Außerdem werden die Scheduling-Verfahren auch untereinander verglichen und bewertet.

Hier werden zum Einen die reinen Ausführungszeiten des Algorithmus, zum Zweiten die Kosten[1] durch Migrationen und zum Dritten das Caching-Verhalten für die Bewertung der Scheduling-Verfahren herangezogen. Migrationen sind deshalb von Interesse, da Tasks auf einem System im SMP-Betrieb von einem Kern auf einen anderen migriert werden kann und dort weiter ausgeführt werden. Diese Migrationen sind mit zusätzlichen Kosten verbunden, da es dazu nötig ist die betroffenen Cachelines ebenfalls auf den neuen Kern zu übertragen. Für gewöhnlich wird die Übertragung dieser Cache-Inhalte über den Speicherbus getätigt, was dazu führt, dass weiterer Datenverkehr auf dem Speicherbus entsteht und das System dadurch verlangsamt wird. Das Caching-Verhalten gibt Aufschluss darüber, wie oft Daten aus dem Hauptspeicher nachgeladen werden müssen. Dies ist ebenfalls eine Kennzahl zur Bestimmung der Effizienz eines Scheduling-Verfahrens.

Als Testsystem dient das ADIT-Testboard. Es besitzt einen ARM11 MPCore (2.2.1) mit vier Prozessorkernen je 400MHz. Als Betriebssystem wird das bereits erwähnte eT-Kernel verwendet. Die erforderlich Messinstrumente werden in eigenen Funktionen implementiert, die dann die aufgezeichneten Werte in verschiedene Dateien schreiben.

Der Aufbau dieser Arbeit gliedert sich nach folgendem Schema. Zunächst werden in Abschnitt 2 die Grundlagen für diese Arbeit erläutert. Darin enthalten sind alle relevanten Begriffe und Konzepte, die für das verfolgen der Untersuchungen notwendig sind. Im Abschnitt 3.1 werden die Maßnahmen beschrieben, die vor den Untersuchungen ergriffen werden müssen. Unter anderem wird dort die Portierung der Software und die Implementierung grundlegender Messfunktionen beschrieben.

[1] Bedeutet in diesem Zusammenhang: Zeit die benötigt wird, um eine Aufgabe zu erledigen

Der Abschnitt 3 wird dann die eigentliche Untersuchung der Scheduling-Verfahren beschrieben. Teil dieser Beschreibungen ist die Testvorbereitung, der Testaufbau, Testdurchführung und die Interpretation der Ergebnisse mit anschließenden Schlussfolgerungen. Zwischen den Kapiteln der Untersuchungen der seriellen und der parallelen Ausführung des SIFT-Algorithmus befindet sich der eingeschobene Abschnitt 3.4 in dem beschrieben wird, wie der SIFT zu einem parallel ausführbaren Algorithmus implementiert wurde.
Am Schluss dieser Arbeit werden die Ergebnisse und Schlussfolgerungen im Abschnitt 4 noch einmal zusammengefasst beschrieben. Hier werden zudem weitere Aspekte genannt, die für weiterführende Arbeiten von Interesse sind.

2 Grundlagen

2.1 Stand der Technik

Im folgenden Kapitel wird sowohl der Stand der Technik (SDT) als auch der Stand der Forschung (SDF) beschrieben. Der Teil des SDTs geht zunächst auf den Bereich der Bildverarbeitung ein. Danach folgt eine kurze Zusammenfassung der derzeitigen Echtzeitbetriebssysteme und deren Anwendungsbereiche. Am Schluss dieses Kapitels wird dann der SDF beschrieben und geht auf bereits existierende Forschungsergebnisse im Bereich der Schedulingverfahren ein.

2.1.1 Bildverarbeitung

Die Bildverarbeitung (kurz BV) ist heute schon aus vielen Bereichen unseres alltäglichen Lebens nicht mehr wegzudenken. Überall, wo es darum geht mit Hilfe eines Bildes Informationen zu sammeln, werden BV-Algorithmen eingesetzt. Und längst hat sich diese Methode der Verarbeitung von Informationen auch in sicherheitskritische Bereiche erfolgreich etabliert. Mediziner nutzen bereits Bildverarbeitungssysteme um ihre Diagnosen sicher und schneller stellen zu können [zu Bexten und Hiltner(1998)].

Die Bildverarbeitung nutzt spezielle Algorithmen um Informationen aus digitalen Bildern zu extrahieren. Oft bestehen diese komplexen Algorithmen aus einer Zusammenstellung von kleinen Bildverarbeitungsprozeduren. Der in dieser Arbeit verwendete SIFT-Algorithmus (Scale-Invariant-Feature-Transform) ist solch ein komplexer BV-Algorithmus [Lowe99(1999)]. Hier werden mit Hilfe verschiedener BV-Prozeduren (Faltungsmatrizen, etc.) Bilder für eine Merkmalerkennung vorbereitet. Eine genauere Beschreibung der Arbeitsweise befindet sich im Kapitel 2.4.4 auf Seite 24.

2.1.2 Echtzeitbetriebssysteme

Im Vergleich zu den Desktop PCs (wo es die drei großen Betriebssysteme - Linux, MS Windows und Mac OSX gibt) gibt es in der Welt der eingebetteten Systeme viele und unterschiedliche Echtzeitbetriebssysteme (kurz RTOS[2]). QNX beispielsweise, stellt mit *Neutrino* ein RTOS für medizinische Geräte zur Verfügung [QNX(2009)]. eSOL hat mit eT-Kernel ein Betriebssystem entwickelt, das überwiegend im Bereich der Fahrerinformationssysteme verwendet wird[eSoL(2009)]. Microsoft hingegen hat mit MS Mobile ein RTOS auf dem Markt, welches hauptsächlich bei mobilen Geräten wie PDAs und SmartPhones eingesetzt wird. Eine detaillierte Beschreibung zu den Anforderungen eines RTOS sind im Kapitel 2.4.1 auf Seite 19 zu finden.

2.1.3 Stand der Forschung

Es gibt heute viele verschiedene wissenschaftlich dokumentierte Arbeiten über die Problematik der Flaschenhälse bei Mehrkern-Systemen. Es fallen insbesondere die Arbeiten zu Caching Protokollen wie *Snoop-Protocol* und *Directory-Protocol* auf [Agarwal u. a.(1988)Agarwal, Simoni, Hennessy und Horowitz]. Darüber hinaus gibt es auch Arbeiten dahingehend, wie Ressourcen intelligent durch Scheduling genutzt werden können, um beispielsweise gleichartige

[2] Real-Time Operating System

Tasks mit unterschiedlichem Arbeitsaufwand geschickt auf CPUs zu verteilen. Problematisch sind aber auch hier die Engpässe auf dem Speicherbus [Stensland u. a.(2008)Stensland, Griwodz und Halvorsen].
Bei den Untersuchungen der Caching-Verfahren wurde beispielsweise gezeigt, dass *Snoop Control* Protokolle zwar recht schnelle Prozeduren sind um die Cache-Kohärenz aufrecht zuhalten, jedoch sind diese Protokolle nicht beliebig skalierbar. Im Hinblick auf die wachsende Zahl der Prozessoren, wird durch den Einsatz der Broadcasts [3] auf dem Speicherbus (oder Prozessorkernen) immer mehr Bandbreite auf dem Bussystem verbraucht. Dies führt bei steigender Benutzung der Broadcasts und gleichbleibender Bandbreite des Bussystems zu einem Engpass. Denn durch die wachsende Zahl der Prozessoren, steigt auch der Aufwand zur Sicherstellung, das der Cache kohärent bleibt. Weiter wird gezeigt, dass ein *Directory-Protocol* bei einer Vergrößerung eines Mehrkern-Systems mehr Möglichkeiten bieten kann, als im Vergleich zu den *Snoop-Protocols*. Nachteilig ist hier jedoch, dass das zentrale Verzeichnis (also das Directory) bei jedem Cache-Miss und Write-Hit durchsucht werden muss, um zu klären, welcher Cache von der temporären Inkonsistenz betroffen ist [Agarwal u. a.(1988)Agarwal, Simoni, Hennessy und Horowitz].
Betrachtet man nun die Engpässe vom Scheduling aus, findet man auch hier Probleme bezüglich der Datenraten auf den Kommunikationsbussen. Stensland (et. al.) zeigen in ihren Arbeiten zwar zum einen, dass es möglich ist CPU-Belastungen geschickt zu verteilen (in ihrem Beispiel Videokonferenzen) gehen aber auch auf die Engpässe im Bereich der CPU-Kommunikation ein. Sie zeigen Anhand vier verschiedenen Methoden (*First Hit*, *Next Hit*, *Random* und *Worst Hit*) wie oft statisch ein (Teil-)Prozess *nicht* verteilt wird, weil beispielsweise die Echtzeitanforderungen nicht erreicht wurden. Ihre Folgerung war, dass ein Scheitern der Echtzeitanforderung oft daran lag, dass zu lang gesucht wurde, bis eine geeignete CPU gefunden werden konnte.

2.2 Hardware

2.2.1 ARM 11 MPCore

Der in dieser Arbeit verwendete ARM11 MPCore basiert auf der ARM 11 Mikroarchitektur und trägt vier Prozessorkerne mit jeweils 400 MHz Taktfrequenz - also einer Gesamtrechenleistung von 1,6 GHz. Das verwendete Speichersystem ist eine Unified Memory Architecture (UMA)[4].

Unterstützt wird das ARMv6K Instruction Set, welches das 32-Bit ARM, 16-Bit Thumb und 8-Bit Jazelle Instructions Set ausführen kann. Außerdem wird SIMD, eine Medienerweiterung, unterstützt.

Thumb Eine Erweiterung der ARM Architektur. Sie beinhaltet die meisten 32-Bit ARM Anweisungen, welche in 16-Bit breite Opcodes codiert worden sind, um Speicheranforderungen zu verringern.[ARM Ltd.(2009)ARM-11MPCore]

[3] Ein Broadcast spricht **alle** in einem Komunikatoinssystem an angeschlossenen Systeme an
[4] Ein globaler Speicher für alle Kerne. Die Anbindung (Bandbreite und Zugriffsgeschwindigkeit) an den Speicher ist für alle Kerne gleich

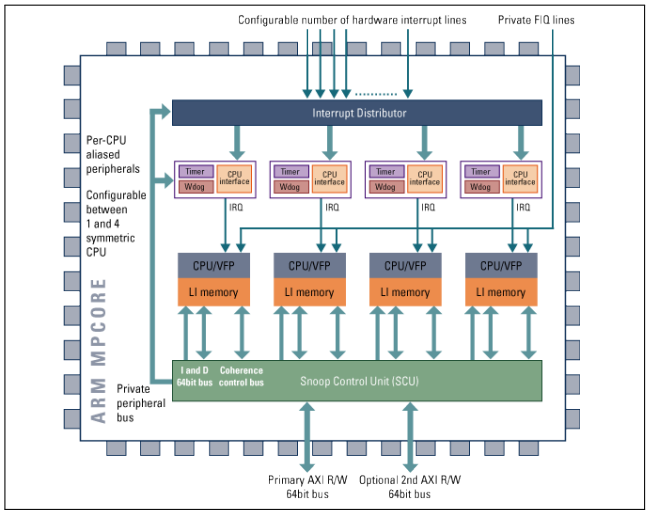

Abbildung 1: Schematischer Aufbau eines ARM11 MPCore.
Quelle: http://www.jp.arm.com

SIMD **S**ingle **I**nstruction, **M**ultiple **D**ata: Ist eine von Michael J. Flynn definierte Befehlssatzerweiterung[Flynn(1972)], die es ermöglicht mit einer Anweisung mehrere gleichartige Datensätze zu verarbeiten. Außer dem SIMD unterscheidet die Flynn'sche Klassifikation Befehlssatzerweiterung zwischen

- SISD (Single Instruction, Single Data)
  Hierbei handelt es sich um eine Architektur, die in einem Prozessortakt, immer eine Instruktion und einen Datensatz zur Zeit be- verarbeiten kann.
- MISD (Multiple Instruction, Single Data)
  Bei dieser Architektur können, gleich mehrere Instruktionen hinter einander auf einen Datensatz angewendet werden, ohne dafür nach jeder Instruktion eine neue zu laden.
- MIMD (Multiple Instruction, Multiple Data) Hier können in einem Takt sowohl mehrere Instruktionen geladen und ausgeführt werden, als auch mehrere Datensätze verarbeiten.

Diese Architektur wird von Intel seid 1997 in ihre Prozessoren integriert und nennt sich Multi Media Extension (kurz MMX). Mit MMX ausgerüstet ist der Prozessor in der Lage parallelisiert große Datenmengen zu verarbeiten. Hierzu wurden ganz nach dem SIMD Vorbild Datensätze mit einem Befehl parallel verarbeitet. Seid dem Intel Pentium III Katmai Prozessor, wurde MMX durch Streaming SIMD Extensions (SSE) erweitert. Mit SSE wurde die Bandbreite der parallel zu verarbeitenden Daten erhöht und die Anzahl der Instruktionen des Befehlssatzes vergrößert.

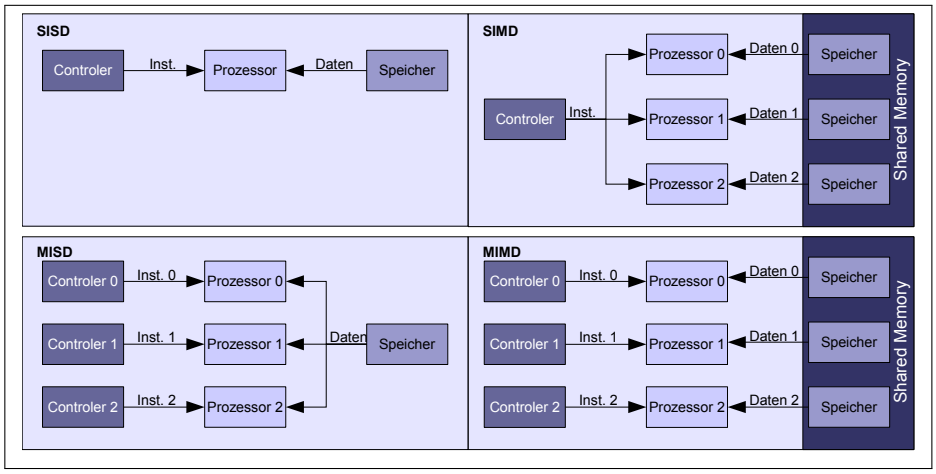

Abbildung 2: Schematische Darstellung der Befehlssatzerweiterung nach [Flynn(1972)] .
Quelle http://en.wikipedia.org

2.2.2 Cache

Oft wird die Geschwindigkeit eines Computers an der Taktung der CPU gemessen. Dies jedoch stellt nur einen Teil der tatsächlichen Leistung eines Computers dar. Aus diesem Grund wird in dieser Arbeit auch das Speicherverhalten der durch die verschiedenen Scheduling-Verfahren mit in die Untersuchung aufgenommern. Denn die Kommunikation zwischen der CPU und dem Hauptspeicher ist ein weiterer Bestandteil der Rechenleistung in einem Rechnersystem. So sind Latenzzeiten und die Übertragungsgeschwindigkeit der Daten von Speicher zur CPU mitentscheidend dafür, ob ein System optimal arbeitet. Speichermodule der Klasse DDR2-1066 schaffen 8,5 GByte/s und im Dual-Channel Modus 17 Gbyte/s. Eine 3GHz CPU „verbraucht“ jedoch, gemessen an ihrer Taktung, 60 GByte/s allein an Instruktionen[Benz(2008)].

Um diesen Engpass entgegenzuwirken, werden auf CPU-Kernen (der sog. Die) kleinere aber schnelle Speichermodule verbaut - die Caches. Dieser Speicher einer CPU befindet sich ebenfalls auf der Die und hat dadurch kürzeste Wege zum CPU-Kern. Daher ist die Kommunikation zwischen CPU und Speicher in diesem Bereich besonders schnell. Jedoch ist die Idee, mehr Cache-Speicher auf der Die zu verbauen, nicht einfach durchsetzen. Die zumeist aus SRAM-Zellen bestehenden Cache-Module haben im Vergleich zu den DRAM-Modulen des Hauptspeichers größere Zellbausteine. So besteht ein DRAM für gewöhnlich aus einem Transistor und einem Kondensator (1T1C). Der SRAM ist mit 6 Transistoren und einem Kondensator erheblich größer [Allan u. a.(2002)Allan, Edenfeld, Joyner, Kahng, Rodgers und Zorian]. Da aber die räumliche Nähe zum CPU-Kern entscheidend ist, ist es nicht möglich beliebig viel Speicher auf der Die zu platzieren. Der größere Abstand zum Kern würde dann wieder für längere Wege sorgen und die Latenzzeit vergrößern. Hinzu kommen noch die zusätzlichen Kosten, die eine Erweiterung des Cachesspeichers verursachen würde. Aufgrund der Komplexität des 6T1C SRAMs ist dieser teurer als der im DRAM verwendete 1T1C-Speicher.

Um den Mangel an Platz möglichst zu kompensieren, geht man bei dem CPU-Design einen Kompromiss ein. Der Speicher wird in 2 bis 3 Stufen aufgeteilt. Auf der ersten Stufe befindet sich der L1-Cache (Level 1). Dieser Speicher ist in den meisten Fällen 64KByte groß und wird sehr dicht am Kern verbaut. Dadurch hat er die kürzeste Latenzzeit und ist somit am schnellsten. Der L1-Cache beinhaltet oft den Instruction- und Data-Cache.

Der Level 2 Cache ist weiter vom Kern entfernt, arbeitet im CPU-Takt und hat, bedingt durch die räumliche Trennung, eine langsamere Zugriffsgeschwindigkeit als der L1-Cache. Hier macht sich bereits der Weg den die Daten zurücklegen müssen bemerkbar. Die L2-Speichergröße variiert je nach Prozessortyp von 512 KByte bis 6 MByte.

Bei einigen CPU-Designs gibt es noch eine weitere Cachestufe (Level 3 Caches). Dieser Speicher wird meistens mit weniger schnellen Taktraten angesprochen und ist daher noch langsamer, als der L2-Cache. Dafür werden die L3 Speicher für gewöhnlich mit mehr Speicherkapazität ausgestattet.

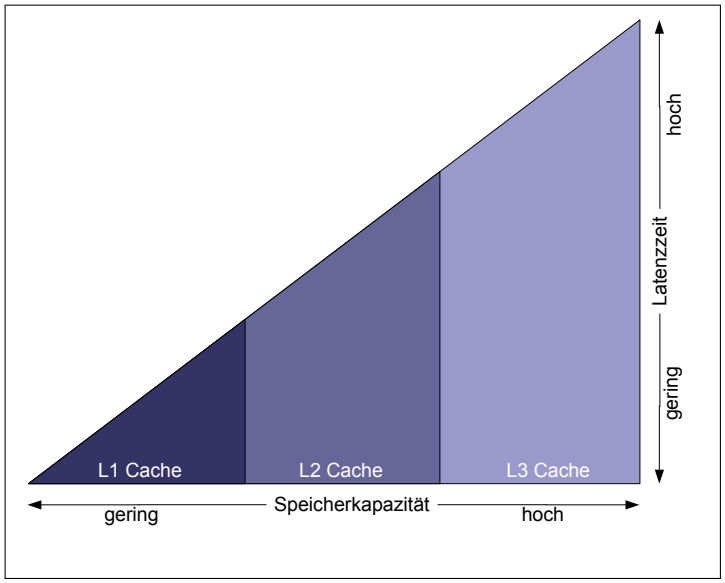

Abbildung 3: Cache-Speicher im Vergleich.
L1: Wenig Speicherkapazität, kurze Latenzzeiten
L2: Mittlere Speicherkapazität, mittlere Latenzzeit
L3: Hohe Speicherkapazität, hohe Latenzzeit

Aus diesen verschiedenen Cache Stufen ergeben sich jedoch auch Herausforderungen. Damit ein CPU-Kern mit einem Datum arbeiten kann, muss dieses Datum vom Hauptspeicher in den lokalen Cache-Speicher geladen werden. Wird dann mit dem Datum gearbeitet entsteht Inkonsistenz zwischen der lokalen Cache-Kopie und dem Original des Hauptspeichers. Diese Inkonsistenz muss behoben werden, sobald eine weitere Ressource (meist ein weiterer Kern) dieses Datum anfordert.

Wie bereits im Kapitel 2.1 beschrieben, gibt es für die Einhaltung der Kohärenz verschiedene Ansätze. Hier wird nun die so genannte Snoop Control Unit (SnCU) beschrieben. SnCUs überwachen die in den Caches liegenden Daten und weisen den Cache bei Bedarf an, Daten zurück in den Hauptspeicher zu schreiben. Dies passiert immer dann, wenn eine weitere Ressource diese Daten anfordert. In einem Mehrkern-System wird diese Aufgabe dahingehend erweitert, dass auch die einzelnen Caches untereinander kohärent sind. Diese Überwachung wird in den meistens Systemen durch das so genannte MESI[5]-Protokoll geregelt.

Dieses Protokoll dient beispielsweise in einem Mehrkern-System dazu, die einzelnen exklusiven Caches untereinander kohärent zu halten. Das bedeutet, dass die SnCU immer genau darüber Informiert sein muss, welche Daten in welchem Speicher (sowohl im Hauptspeicher, als auch in den verschiedenen Caches) vorhanden sind und ob diese Daten von einem Kern verändert wurden. Um dies zu erreichen, werden den Daten in den Caches zwei Kontroll-Bits hinzugefügt. Sie enthalten die Informationen, in welchem Zustand sich ein Datum befindet. Hierbei unterscheiden die beiden Bits die vier vom MESI (siehe Add. 4) definierten Zustände

[5] **M**odified **E**xclusive **S**hared **I**nvalid

- Modified
  Soll ein Datum im Cache eines Kerns verändert werden, wird sein *modified*-Bit gesetzt. Ist zusätzlich das *shared*-Bit gesetzt, werden alle Kopien des Datums von der SnCU invalidiert. Die anderen Ressourcen müssen dann ihre lokale Kopie aktualisieren, bevor sie mit dem Datum arbeiten dürfen.

- Exclusive
  Definiert ob ein Datum exklusiv von einer Ressource verwendet wird. Ist so ein Datum in einem Cache vorhanden, muss die SnCU bei einer weiteren Anfrage dieses Datums den exklusiv Status aufheben und auf *shared* setzen.

- Shared
  Definiert ob ein Datum von verschiedenen Ressourcen verwendet wird. Wird dieses Datum geändert, muss die SnCU dieses Datum für alle anderen Ressourcen invalidieren.

- Invalid
  Invalidiert wird ein Datum dann, wenn eine Ressource ein Datum ändert und dieses Datum zusätzlich als *shared* markiert ist.

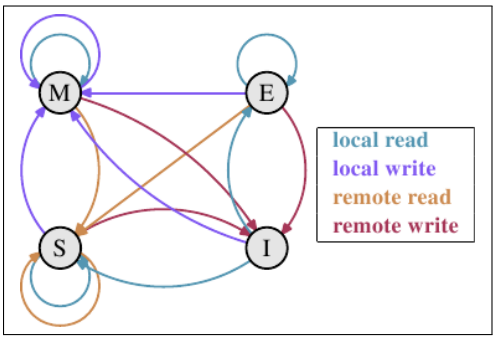

Abbildung 4: Stark vereinfachte Illustration des MESI-Protokolls

2.3 Symetric Multiprocessing

Um ein Rechnersystem in einen vollständigem SMP-Betrieb betreiben zu können, müssen hierfür drei Voraussetzungen erfüllt sein.

1. Hardware Ebene:
   Eine homogene Prozessor/Mehrkern Architektur
2. Betriebssystem Ebene:
   Ein Betriebssystem allein verwaltet diese Prozessoren oder Kerne und es muss die laufenden Tasks auf diesen Prozessoren verteilen können.
3. Software/Applikation Ebene:
   Ein Task muss sich in mehrere Tasks zerlegen lassen können, um auf den Kernen parallel ausgeführt werden zu können.

Im folgenden werden diese drei Voraussetzungen kurz erleutert.

2.3.1 Hardware Ebene

Unter einer homogenen Prozessor-Architektur versteht man beispielsweise Prozessoren, die mehrere identische Kerne besitzen und einen gemeinsamen Datenspeicher haben. Um die Kohärenz dieser Speicher aufrecht zuerhalten, wird eine Snoop Control Unit (SnCU) (siehe Abschnitt 2.2.2) eingesetzt. Die SnCU sorgt mit Hilfe des MESI-Protokolls (siehe Abschnitt 2.2.2) dafür, dass die Kohärenz zwischen den verschiedenen Datenspeichern sichergestellt ist.

2.3.2 Betriebssystem Ebene

Damit die gesamte Leistung des Rechnersystems ausgenutzt werden kann, muss das Betriebssystem alle Kerne verwalten können, um die Last der laufenden Tasks auf die Kerne verteilen.

2.3.3 Software Ebene

Ein Task, der die Vorteile des vollständigen SMP-Betrieb ausnutzen möchte, muss seine Rechenlast in weitere Sub-Tasks oder Threads aufteilen können. Erst dann kann das Betriebssystem durch verteilen der Teile, die gesamte Rechenleistung des Systems ausnutzen.

Mischformen die ein partielles SMP erlauben, könnten beispielsweise folgend definiert sein.

Als Grundlage für diese Erläuterung dient ein homogenes Mehrkernsystem mit 4 Prozessorkernen.

1. Das Rechnersystem besitzt 4 Prozessorkerne und darüberhinaus spezielle Hardware wie DSPs (Sound-Karte) oder GPUs. Damit könnten die 4 Prozessorkerne im SMP Betrieb verwendet werden, spezielle Aufgaben jedoch kann an einen DSP deligieren.
2. Ein Betriebssystem verwaltet 3 der 4 Kerne. Auf dem verbleibendem Kern wird ein weiteres Betriebssystem ausgeführt.

3. Durch Scheduling-Verfahren des Betreibssystems wird ein Teil der Tasks auf allen Kernen verteilt ausgeführt (*TSM*). Der anderer Teil (beispielsweise Treiber und Betriebssystem-Tasks) werden auf einen Kern gebunden und nur dort ausgeführt (*SPM on TSM*).

2.4 Software

2.4.1 Echtzeitbetriebssysteme

Betriebssysteme haben im allgemeinen die Aufgabe, die Betriebsmittel eines Rechnersystems zu verwalten, Tasks zu steuern und ebenfalls zu verwalten. Sie abstrahieren die Hardware und schaffen somit Schnittstellen für Entwickler um bequem neue Software zu implementieren. Zudem schützt das Betriebssystem ein Rechnersystem vor Software mit Programierfehlern oder Schadsoftware. Ein Konzept zum Schutz des Rechnersystems ist beispielsweise, die Virtualisierung des Speichers um Tasks daran zu hindern, in den Speicherbereich eines anderen Tasks zu schreiben.

Bei der Task-Verwaltung ist insbesondere die parallele Ausführung[6] eine Kernanforderung. Diese Anforderung wird durch den Scheduler realisiert, der anhand einer Regel an alle Tasks CPU-Zeit vergibt. Zudem kann der Scheduler bei einem Mehrkernsystem entscheiden, auf welchem CPU-Kern ein Task ausgeführt werden soll, bzw. kann bei einer hohen Belastung eines einzelnen Kerns ein Task auf einen anderen Kern verschoben werden. Ein solches Verschieben, wird auch als Task-Migration bezeichnet. Eine detaillierte Beschreibung der Schedulingmethoden des in dieser Arbeit verwendeten eT-Kernels befindet sich im Kapitel 2.4.2 auf Seite 20

An Echtzeitbetriebssysteme werden noch weitere Anforderungen gestellt. Hier gilt es nicht nur ein korrektes Ergebnis zu liefern, sondern innerhalb eines festen Zeitrahmens (sog. Deadlines). Der Zeitpunkt bis zu dem ein korrektes Ergebniss erzeugt werden darf, kann dabei sowohl "hart"[7] als auch "weich"[8] definiert werden. Wichtig dabei ist jedoch nicht, wieviel Zeit das System tatsächlich für eine designierte Reaktion benötigt, solange die Ausführung innerhalb des vorgeschriebenen Zeitrahmen eingehalten wird. Darüberhinaus müßen auch die Entwickler der Software darauf acht geben, dass der zeitkritische Task so implementiert wird, dass das Betriebssystem in der Lage ist, die Echtzeit Anforderung einzuhlaten.

Abgesehen von den Limitierungen der Ausführungszeit ist die Reaktionszeit eines Echtzeitbetriebssystems ein wichtiger Faktor. Typischerweise gibt es 3 Zustände, die ein Task annehmen kann:

1. Task in Ausführung
2. Task bereit für Ausführung und wartet auf CPU-Zeit
3. Task geblockt durch anderen Task/Interrupt

Die Behandlung eines Interrupts (Beispielsweise ausgelöst durch ein Eingabegerät wie Tastaturen) stellt beispielsweise "Echtzeit-Anforderung" an das Betriebssystem

[6] "quasi" parallele Ausführung bei SingleCore Systemen
[7] Ein Überschreiten der Ablaufzeit wird als Fehler bewertet
[8] Hier kann man selbst entscheiden, welche Latenzzeiten noch als akzeptabel gilt

und muss in definierter kurzer Zeit aus dem Zustand **bereit** oder **blockiert** in den Zustand **in Ausführung** überführt werden. Hinzu muss der Tasks der von der CPU weichen muss, aus dem Zustand **in Ausführung** in den Zustand **blockiert** überführt werden. Im Durchschnitt benötigt ein Scheduler für das Auslagern des derzeitig ausgeführten Tasks zwischen 3 und 20 Instruktionen. Für das Laden des auszuführenden Tasks 5 bis 30 Instruktionen.

2.4.2 Das Echtzeitbetriebssystem: eT-Kernel (Multi-Core Edition)

Software die ursprünlich für ein SingleCore System entwickelt wurde, ist nicht automatisch für den Einsatz auf MultiCore Systemen geeignet. Eine der Hauptfunktionen des eT-Kernel (Multi-Core Edition) ist die Fähigkeit eine oder mehere SingleCore Umgebungen auf einem MultiCore System zu erzeugen. Innerhalb dieser Umgebung wird ein Task so ausgeführt, als währe er noch immer auf einem SingleCore System. In Abbildung 5 wird beispielhaft gezeigt, wie SingleCore Anwendungen auf zwei verschiedene SingleCore Umgebungen eines MultiCore Systems ausgeführt werden können. Die Tasks $1, \ldots, 3$ laufen hier ausschliesslich auf Kern 0, die Tasks $4, \ldots, 6$ analog auf Kern 1. Die Anpassung der Software - wenn denn eine vorgenommen werden muss - ist minimal.

Neben der erzeugten SingleCore Umgebungen erlaubt das Betriebssystem zusätzlich den Einsatz einer parallel laufenden MultiCore Umgebung. Die schwarze, vertikal gestrichelte Linie in Abbildung 5 deutet die Trennung der einzelnen Umgebungen an. Parallel zu den SingleCore Umgebungen werden die Kerne 2 und 3 zu einer MultiCore Umgebung zusammengefasst. Hierbei ist es wichtig, dass die Tasks $7, \ldots, 12$ sowohl auf einen Kern beschrenkt sein können, als auch zwischen ihnen Migrieren. Ob ein Task migrieren darf, ist dann vom zugewiesenem Scheduling Verfahren abhähing. Die Zusweisung eines der vier Verfahren, wird bei der Taskkreierung vollzogen. Ist der Task einmal einem Scheduling Verfahren zugeordnet, erlaubt das Betriebssystem einen Wechsel zu einem anderem Verfahren nicht mehr.

Die Nomenklatur der Scheduling Verfahren ist an dieser Stelle etwas verwirrend. Denn zu den Scheduling Verfahren auf der übergeordneten Betriebssystemebene, existieren zusätzlich die gewohnten Scheduler (FIFO oder Round Robin) der einzelnen Kerne, die wie gewohnt entscheiden, welcher Task in ihrer Liste als nächstes ausgeführt wird. In Abb. 6 ist stark vereinfacht dargestellt, auf welchen Level sich die beiben Scheduler jeweiles befinden. Im folgenden Abschnitt sind mit dem Ausdruck Scheduling Verfahren jeweils *SPM*, *TSM*, *SPM on TSM* und *SLR on TSM* aus Level 2 gemeint.

Die Scheduling Verfahren stellen die Ausführungsrichtlinie dar und sorgen dafür, dass Tasks auf die Prozessorkerne verteilt und ausgeführt werden. Wie oben erwähnt gehört dabei jeder Task einer Scheduling Unit (SU) an. Bei den Scheduling Verfahren unterscheidet eT-Kernel in vier Variationen, die im Folgenden beschrieben werden. Eine detailliertere Beschreibung entnimmt man der eT-Kernel (Multi-Core Edition) Spezifikation.

Single Processor Mode (SPM) Wurde ein Task mit diesem Scheduling Verfahren kreiert, so kann dieser auch nur auf einem Kern ausgeführt werden, der einer SingleCore Umgebung angehört. Wie weiter oben bereits angedeutet, können hier

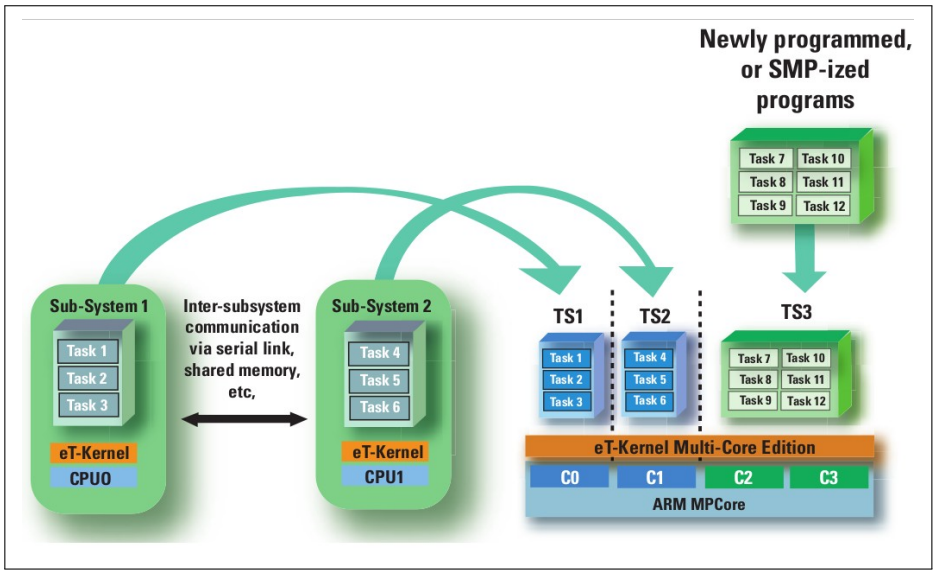

Abbildung 5: Beispielhafte Aufteilung in einem 4-Kern System: Kern 0 und 1 sind jeder für sich eine SingleCore Umgebung. Kern 2 und 3 sind zu einer MultiCore Umgebung zusammengefasst.
Quelle: [Gondo(2006)]

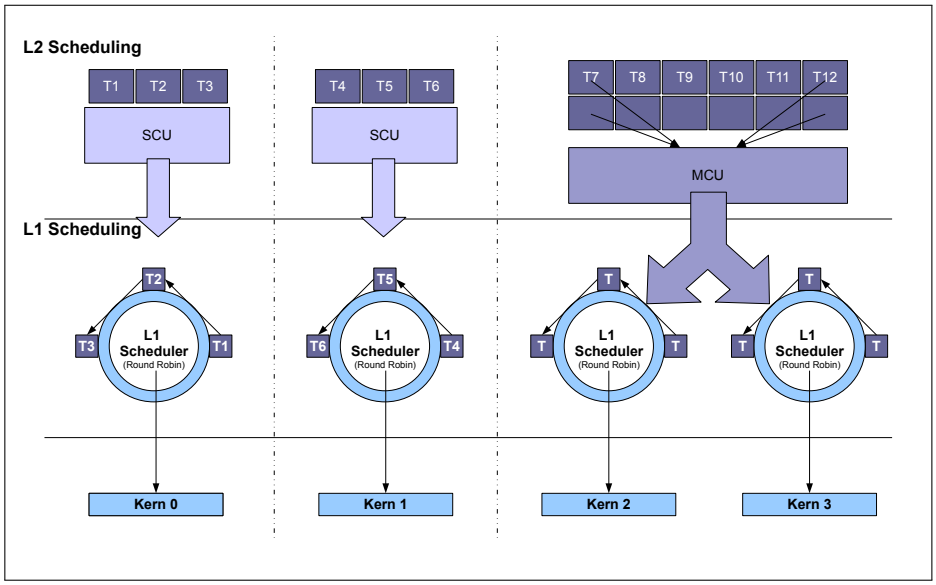

Abbildung 6: Im eT-Kernel existieren zwei Level in denen jeweils ein Scheduling verfahren eingesetzt wird. L1 sind die bekannten Betriebssystem Scheduler (FIFO oder Round Robin). L2 sind die eT-Kernel Scheduling Verfahren *SPM*, *TSM*, usw.

vorallem Amwendungen ausgeführt werden, die ursprünglich für einem SingleCore System implementiert wurden. Darüberhinaus sind Anwendungen auf diesem Kern sicher vor Unterbrechungen durch Anwendungen auf anderen Kernen. Migrationen jeglicher Art (weder von, noch zu einem Kern) sind im SPM Verfahren nicht erlaubt. Dadurch ist dieses Verfahren besonders gut für zeitkritische Anwendungen oder Interruptroutinen geeignet.

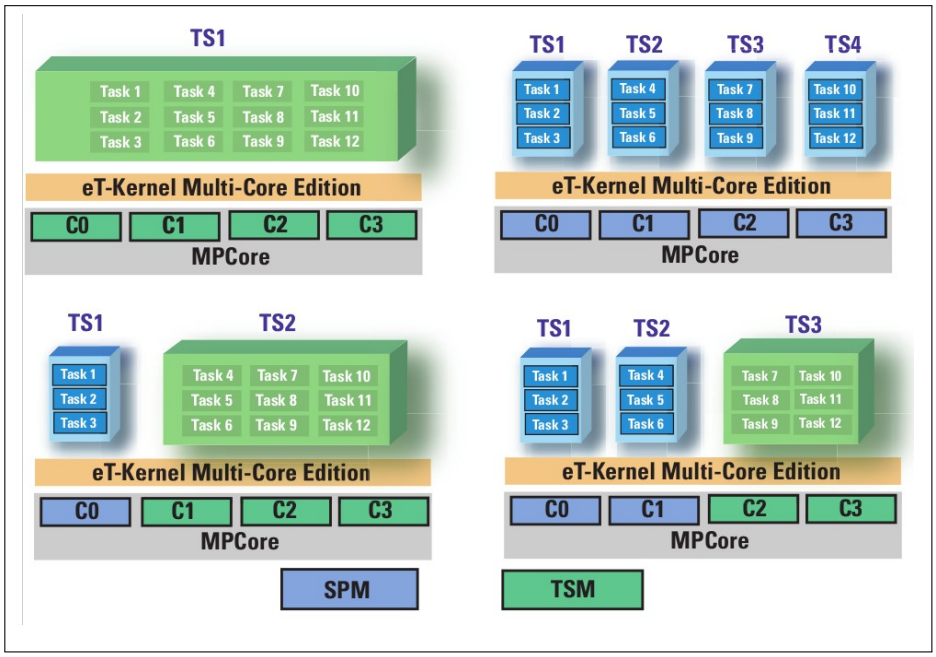

Abbildung 7: Darstellung der möglichen Aufteilung eines MultiCore Systems. Hellblau sind SCUs und Hellgrün eine MCU. Innerhalb der MCU können die Scheduling Verfahren dafür sorgen, ob ein Task Megrieren darf *TSM*), oder nicht (*SPM on TSM*). Zusaätzlich können Gruppendefiniert werden, nur seriell verarbeitet werden sollen (*SLR on TSM*). Quelle: [Gondo(2006)]

True SMP (TSM) Hier entscheidet das Betriebssystem selbständig, welcher Task auf welchem Kern aus geführt werden soll und wie viel Rechenzeit er dort bekommt. Man nennt diesen Modus auch vollständiges SMP, da hier die in einer MultiCore Umgebung zusammengefassten Kerne, Tasks dynamisch zugeteilt bekommen. Jedoch müssen für die im *TSM* Verfahren ausgeführten Tasks entsprechend implementiert worden sein. Nebenläufigkeit ist hier durch real existierendes paralleles Arbeiten besonders häufig. Tasks die Threads kreieren und sich dabei Speicherbereiche teilen, müssen darauf achten, dass der Speicher sicher ist - d.h. durch Sicherungsverfahren wie Mutex, Barrieren, etc.
Sind die entsprechenden Maßnamhmen zum sicheren parallelen Ausführen eines Tasks getroffen, entfalltet sich der entscheidende Vorteil dieses Modus. Die verfügbare Rechenleistung des Systems kann voll ausgeschöpfen werden. Die laufenden Tasks werden auf alle Kerne verteilt, und parallel bearbeitet. Jedoch wird

es sehr schwer in diesem Modus deterministische Aussagen zu treffen, wo ein Task tatsächlich ausgeführt wird. Im Laufe dieser Arbeit mußten daher einige Annahmen getroffen werden, ob es zu Migrationen gekommen ist, oder nicht. Vollständige Ergebnissen liessen sich in einigen Fällen durch fehlende Hardware-Unterstützung nicht ermitteln. Mehr Informationen dazu befinden sich im Abschnitt 3.

Single Processor Mode (SPM) on TSM stellt einen der beiden Spezialfälle des *TSM* Verfahrens dar. Ist es beispielsweise nicht gewünscht oder möglich einen Task im reinen *SPM* einer SCU läufen zu lassen (z.B. durch Hardware restriktionen), kann dieser Task auch diesem Scheduling Verfahren zugeordnet werden. *SPM on TSM* zeichnet sich dadurch aus, dass es die Möglichkeit bietet einem auf einer MCU ausgeführtem Task auf einen Kern dieser MCU zu binden. In Abbildung 7 sind auf der rechten Seite vier Tasks die im *SPM on TSM* Verfahren ausgeführt werden. Zwei von diesen Tasks sind auf den Kern 2, die anderen beiden auf Kern 3 gebunden. Sie werden ausschliesslich auf diesen Kernen ausgeführt. Migration ist durch dieses Scheduling Verfahren nicht erlaubt.
Jedoch ergibt sich aus diesem Szenario ein Unterschied zum *SPM* einer SCU. So könnte beispielsweise ein im *TSM* ausgeführter Task von Kern 3 auf Kern 2 migrieren. Was dazu führt, dass die Aussage „Keine Störung durch kernfremde Tasks“ hier nicht mehr zutrifft.

Serielle Prozesse (SLR) auf TSM Dies ist der zweite Spezialfall des *TSM* Verfahrens. Hat man beispielsweise eine Software, die durch ihre Implementierung geeignet ist „quasi“ parallel auf einem SingleCore System zulaufen. Kann es durch reales paralleles Arbeiten dennoch zu Nebenläufigkeitsproblemen kommen. Um nun ein aufwändiges Neugestalten der Software zu verhindern, kann diese Software im *SLR on TSM* Verfahren genutzt werden. Hier werden Tasks, zu einer sog. Resource-Group (RG) zusammen gefasst. Wird eines dieser Gruppenmittglieder auf einem Kern der MCU ausgeführt, müssen alle weiteren Mitglieder darauf warten, bis das erste Mitglied nicht mehr auf einem Kernen ausgeführt wird. Wie in Abbildung 7 zusehen ist, sind die RG nicht auf einen Kern beschränkt und können von einem Kern zum anderen migriert werden.

2.4.3 Programm Bibliothek OpenCV

In dieser Arbeit wird die Programmbibliothek OpenCV, welche von Intel entwickelt wurde, verwendet. 2006 war der erste Release-Termin und erschien für die Programmiersprachen C und C++. Zuvor wurde die erste Alpha-Version im Jahr 2000 bei der “IEEE Conference on Computer Vision and Pattern Recognition” vorgestellt. Mittlerweile ist OpenCV an Sourceforge übergeben worden und wird seitdem von der OpenSource Gemeinschaft weiterentwickelt. Die aktuelle Version (zum Zeitpunkt der Erstellung dieser Arbeit) ist 1.1 pre vom 15. Oktober 2008. OpenCV ist für MS Windows, Mac OS X, Linux und verschiedene Real Time Betriebssysteme verfügbar. Zum Zeitpunkt des Alpha-Releases erschien das von Dr. Gary Rost Bradski und Adrian Kaehler geschriebene Buch „Learning OpenCV“ im O'Reilly Verlag. Die Anwendungsgebiete dieser Bibliothek erstrecken sich in den gesamten Bereich der Bildverarbeitung, sowie dem maschinellen Sehen und einigen Bereichen des maschinellem Lernens. Die Bibliothek beinhaltet über 500 Bildverarbeitungsalgorithmen und -funktionen. Besonders interessant für diese Arbeit ist diese Bibliothek aus

zwei Gründen. Zum einen, handelt es sich hierbei um freie Software die für wissenschaftliche Zwecke frei verwendet werden darf. Zum anderen setzt der in dieser Arbeit verwendete SIFT-Algorithmus (Siehe Kapitel 2.4.4) auf die von OpenCV implementierten Funktionen auf.
Der SIFT-Algorithmus verwendet aus der OpenCV-Bibliothek

- die Lade- und Speicherfunktionen,
- Bildgrößenmanipuilationen mit verschiedenen Interpolationsfunktionen,
- Bildformatkonvertierungen (RGB zu Graustufen, Invertieren, . . .)
- Gauß'schem Weichzeichner,
- sowie verschiedene Matrizenmanipulationsfunktionen und -transformationen

2.4.4 SIFT - Scale-invariant feature transform

In dieser Arbeit wurde der von David G. Lowe entwickelte SIFT (Scale-invariant feature transform) Algorithmus als Last-Werkzeug eingesetzt [Lowe(2004)]. Die verwendete Implementierung dieses Algorithmus stammt von Rob Hess, Oregon State University. Da diese Version des Algorithmus in der Programmiersprache C implementiert wurde, war sie sehr gut geeignet, um in dieser Arbeit eingesetzt zu werden. Für die SingleCore Testreihen musste an dieser Implementierung lediglich die Zeitmessung (Abschnitt 3.2.3) hinzugefügt werden. Für den Einsatz der MultiCore Variante wurde der Algorithmus dahingehend geändert, das dieser auch parallel die Berechnung eines Bildes vornehmen kann.

In den folgenden Absätzen wird die Arbeitsweise des von Rob Hess implementierten SIFT-Algorithmus beschrieben. Zunächst wird auf die vorbereitende Maßnahmen eingegangen, die getätigt werden, um ein Bild für den Algorithmus zu präparieren. Danach wird beschrieben, was Merkmalspunkte sind und wie diese Punkte vom Algorithmus in einem Bild gesucht werden. Zum Abschluss wird kurz erläutert, wie die gefundene Merkmale mit denen aus einem Referenzbild verglichen und zugeordnet werden. Den Abschluss dieses Abschnitts bildet die Beschreibung des Konzeptes zur Implementierung der parallelen Ausführen des SIFT-Algorithmus.

Zunächst wird das Bild für den Algorithmus vorbereitet. Hierzu wird das Bild in ein Graustufenbild übertragen und anschließend mit einem Gauß'schem Weichzeichner glättet. Die Glättung sorgt dafür, dass Rauschen im Bild verringert wird.

Aufbau der Gauß-Pyramide Nachdem das Bild vorbereitet wurde, wird es zum Erzeugen der Gauß-Pyramide verwendet. Hierzu wird es in sogenannte Oktaven und Intervalle transformiert. Eine Oktave stellt dabei eine Stufe der Pyramide da. Ein Intervall teilt eine Stufe in verschiedene Schichten auf. In Abbildung 8 wird so eine Pyramide in Oktaven und Intervalle zerlegt dagestellt.
Diese Oktaven und Intervalle bilden den *Scale Space*. Der *Scale Space* sorgt dafür, dass dieses Bild Maßstab unabhängig mit einem anderen Eingabebild verglichen werden kann. So können Objekte verglichen werden, die aus unterschiedlichen Distanzen aufgenommen wurden.
Der Aufbau dieser Pyramide läuft wie folgt ab. Eine Oktave wird gebildet, indem

die Auflösung des Eigabebildes in Länge und Breite halbiert wird. Die nächste Oktave wird dann aus der zuvor erzeugten Oktave erzeugt. Dadurch verringert sich die Auflösung der Oktaven mit jeder Iteration. Die Anzahl der zu erstellenden Oktaven wird in dieser Implementierung in Abhängigkeit von der Größe (Länge und Breite) des Eingabebildes berechnet.

Die Intervalle werden erzeugt, indem eine Oktave dupliziert und dann mit dem gauß'schem Weichzeichner geglättet werden. Das folgende Intervall wird dann aus dem zuvor erzeugtem Intervall auf gleiche Weise erzeugt. Die Stärke des Weichzeichnens wird für jedes Intervall dynamisch berechnet. Die Anzahl der Intervalle ist auf mindestens 3 festgelegt, da die spätere DoG-Pyramide mindestens zwei Intervalle je Oktave besitzen muss.

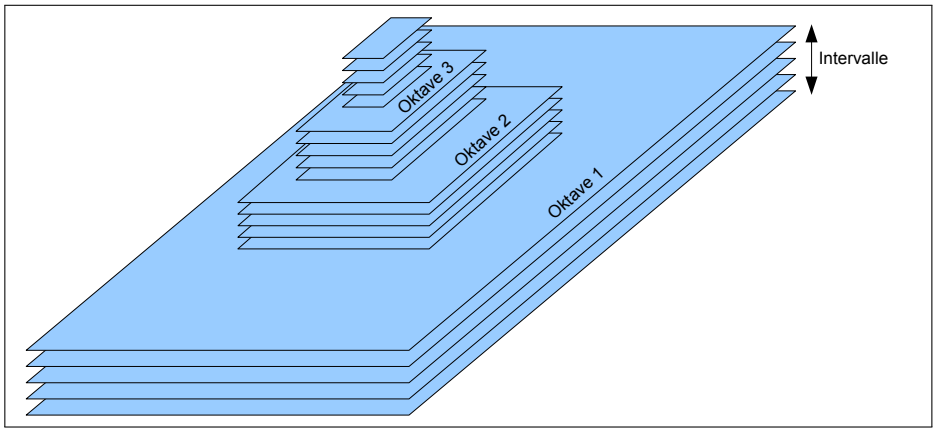

Abbildung 8: Aufbau einer Pyramide aus inkrementellen Gausbildern

Aufbau der DoG-Pyramide Nachdem die $Gauß$-Pyramide erzeugt wurde, wird diese verwendet um die so genannte Difference of Gauß-Pyramide (DoG-Pyramide) zu erzeugen. Die DoG-Pyramide kann als eine Art Suchraum verstanden werden.

Gebildet wird diese Pyramide, in dem die Differenz aus zwei Intervallen einer Oktave der $Gauß$-Pyramide berechnet wird. Dazu wird der Wert eines Bildpunktes des ersten Intervalls vom korrespondieren Bildpunkt des zweiten Intervalls abgezogen. Für die folgende Suche nach den Merkmalspunkten sind mindestens zwei dieser DoG-Intervalle notwendig.

Lokalisieren Um einen möglichen Merkmalspunkt in einem Bild zu lokalisieren, wird jedes Bild der DoG-Pyramide Bildpunkt für Bildpunkt durchlaufen und auf Extremwerte getestet. Hierbei wird zunächst geprüft, ob der absolute Wert des aktuellen Bildpunktes den Kontrastschwellwert (wird dynamisch ermittelt) des derzeitigen Intervalls überschreitet. Wird der Schwellwert überschritten, erfolgt daraufhin die Prüfung, ob der aktuelle Bildpunkt im Vergleich zu seinen Nachbarbildpunkten einem Extrempunkt entspricht. Dazu werden die acht

Nachbarbildpunkte des aktuellen Intervalls und die neun Nachbarbildpunkte des vorhergehenden und nachvollgenden Intervalls mit dem aktuellen Wert des Bildpunktes verglichen (Siehe Abb. 9 auf Seite 26). Da dies in der DoG-Pyramide geschieht, ist dieser Test sowohl für ein Maximum als auch ein Minimum zu prüfen, da der Wert eines Bildpunktes in der DoG-Pyramide durch die Differenzbildung auch negative ganze Zahlen annehmen kann.
Handelt es sich um einen Extremwert, wird die Position des Bildpunktes inter-

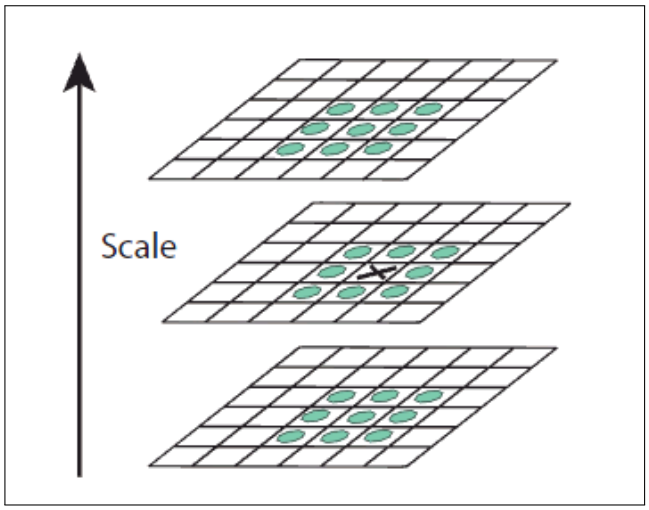

Abbildung 9: Die 3x3x3 Nachbarschaft bei der Extremwertbetrachtung
Quelle: [Lowe(2004)]

poliert. Eine Interpolation ist deshalb notwendig, da es in Oktaven mit geringer Bildauflösung (also weiter "oben" auf der Pyramide) nicht möglich wäre, die Position des Merkmalspunktes genau zu beschreiben. Zur Bestimmung der Position wird ein Approximationsverfahren verwendet. (Siehe [Lowe(2004)] Kapitel 4).
Abschließend wird noch geprüft, ob der Merkmalspunkt an einer Kantenecke liegt. Ist dies der Fall wird der Merkmalspunkt verworfen, da diese Merkmale anfällig für Rauschen (Noise) im Bild sind und damit fälschlicherweise als Merkmal erkannt werden könnten.

Charakteristisch für die Erzeugung der Pyramiden und der Detektion ist, dass hier jweils kontinuirlich durch das Eingabebild itereiert wird, bis die Pyramiden aufgebaut beziehungsweise mögliche Merkmale lokalisiert wurden. Die möglichen Merkmalspunkte selbst sorgen jedoch in den nächsten Verarbeitungsschritten dafür, dass heufig im Speicher gesprungen werden muss. Wie man sehen wird, werden die Bildpunkte direkt "angesprungen". Die sorgt für Last der Speicherverwaltung. Daher werden bei den späteren Untersuchungen unterschiede in den Ausführungszeiten der Threads nicht vermeidbar sein.

Erzeugen der Deskriptoren Die ermittelten möglichen Merkmalspunkte werden nun verwendet um, für jeden dieser Merkmalspunkte einen Deskriptor anzulegen.
Ein Deskriptor ist eine Liste aus 128 Elementen. In dieser Liste werden die invarianten Informationen wie Maßstab, Beleuchtung, Aufnahmewinkel und Ro-

tation gespeichert. Diese Informationen werden später bei der Zuordnung zweier Merkmalspunkten aus zwei verschiedenen Eingabebildern verwendet.

Für die Maßstabsunabhängigkeit muss zunächst der Maßstab für alle gefundenen Merkmalspunkte mit der Gleichung

$$scale = \sigma * 2^{\frac{O_m + I_x}{I_{total}}} \tag{1}$$

mit O_m = Oktave in der das Merkmal gefunden wurde, I_m = Intervall in der das Merkmal gefunden wurde und I_{total} = Anzahl aller Intervalle aus O berechnet werden.

Die Rotationsinvarianz wird durch eine Lokale Orientierung des Merkmalpunktes berechnet. Dazu wird die Richtung und Stärke jedes Gradienten innerhalb eines durch die Oktave definierten Radius iterativ ermittelt. Die zu verwendende Oktave wird durch den bereist ermittelten Maßstabswert bestimmt. Bei jeder Iteration wird die Position des Referenz-Pixel gegen den Uhrzeigersinn um den Merkmalspunkt rotiert (Abb. 10). Die so berechneten Orientierungen des Gradienten werden mit der Stärke gewichtet und in ein Histogramm eingetragen. Die Stärke m eines Gradienten wird mit der Gleichung (Siehe [Lowe(2004)] Kapitel 5)

$$m(x,y) = \sqrt{(B(x+1,y) - B(x-1,y))^2 + (B(x,y+1) - B(x,y-1))^2} \tag{2}$$

mit dem Bildpunkt $B(x,y)$ berechnet.
Die Richtung Θ eines Gradienten ist gegeben durch

$$\Theta(x,y) = \tan^{-1}\left(\frac{\delta y}{\delta x}\right) \tag{3}$$

mit

$$\delta x = B(x+1,y) - B(x-1,y) \tag{4}$$

und

$$\delta y = B(x,y+1) - B(x,y-1) \tag{5}$$

Der größte Peak des erzeugten Histogramms ist zugleich der dominanteste Vektor und zeigt die Hauptrichtung des Gradienten an. Aus zwei weiteren Peaks aus dem Histogramm wird jeweils ein weiteres Merkmal erzeugt. Diese Merkmalgruppe wird zusammengefasst und zu einem Merkmal approximiert. Somit besitzen die Merkmalspunkte einen Maßstabs-Wert der gegen Größenänderungen invariant ist. Zusätzlich ist die Orientierung eines Merkmals bestimmt worden, was eine Invarianz gegen Drehrichtungsänderungen beinhaltet.

Nun wird der eigendliche Deskriptor Vector der einzelnen Merkmalspunkte berechnet. Da dieser Vektor ähnlich wie die Orientierung aus der Umgebnung mit einem bestimmten Radius zusammengefasst wird, ist ein Deskriptor immer unterscheidbar von einem weiterem dieses Bildes. Zudem ist dieser Deskriptor zum Teil invariant gegen Beleuchtungsänderungen und Aufnahmewinkel.

Für reine zweidimensionale Abbildungen würden die bisherigen Schritte ausreichen, um Merkmalspunkte aus zwei verschiedenen Bildern zu vergleichen. Da aber oft auch dreidimensionale Verschiebungen sowie Unterschiede der Beleuchtung

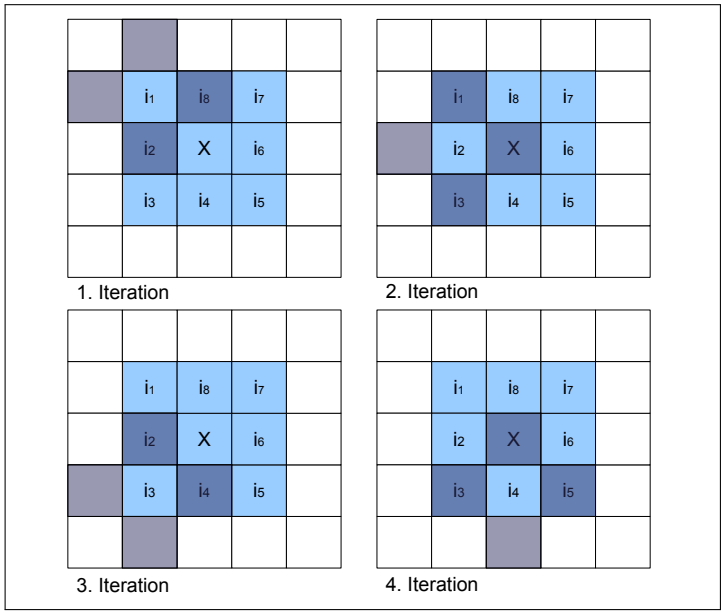

Abbildung 10: Vereinfachte darstellung des Iterationsverfahrens zur Bestimung der Gradientenorientierung

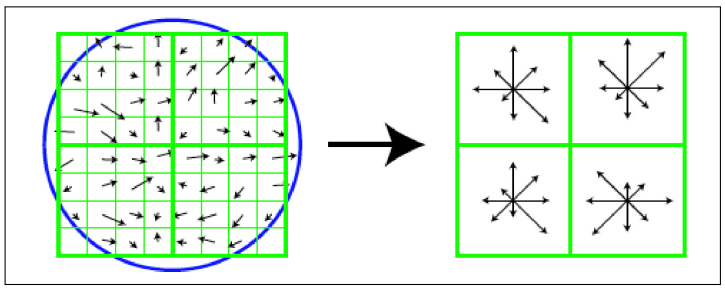

Abbildung 11: Ein Merkmalspunkt wird berechnet durch den Maßstabswert, Orientierung und Stärke des Gradienten (links). Diese werden gewichtet je nach Entverfernung zum Merkmalspunkt schwächer gewichtet. Diese Gradienten werden dann zusammengefasst und in ein Histogramm mit 8 Orientierungen eingetragen (rechts).
Quelle: [Lowe(2004)]

auftreten, sind noch weitere Schritte notwendig.

Beleuchtungsunterschiede werden in einem Bild als Kontrast bezeichnet. Ändert sich der Kontrast eines Bildes im Vergleich zu einem anderen, würden die Werte des Bildpunktes mit der Beleuchtungsänderung multipliziert (multiplikative Farbmischung). Das Gleiche würde mit den Deskriptoren passieren. Um das zu verhindern, werden die Deskriptoren normalisiert und zu einer Einheitslänge geformt. Das allein reicht aber noch nicht aus. Nicht alle Kontrastveränderungen wirken sich gleichmäßig auf das gesamte Bild aus (Schatten, Blendeffekte). Die Kontrastveränderung wirkt sich jedoch nur auf die relative Gradientenstärke aus, nicht aber auf seine Ausrichtung. Somit wird das Problem der veränderten Beleuchtung gelöst, indem die Stärke eines Gradienten mit Hilfe eines Schwellwert begrenzt und anschließend noch einmal normalisiert wird. Das bedeutet nun, dass die Stärke eines Gradienten nicht länger entscheidend ist, dafür aber die Ausrichtung stärker gewichtet wird.

Schritt 3: Merkmalspunkte vergleichen - Zusätzliche Implementierung durch Rob Hess Ist das erste Bild Berechnet worden, werden alle Merkmalspunkte dieses Bildes anschließend in einen kD-Baum (eine k-Dimensionale Binär-Baumstruktur) ein getragen. Das Eintragen in diese Struktur erleichtert (und verkürzt) später, die Suche nach einem bestimmten Merkmalspunkt. Wurde ein weiteres Bild berechnet, werden die Merkmalspunkte anhand ihrer Deskriptoren verglichen. Damit aber nicht jedes Merkmal mit jedem verglichen werden muss, wird anhand der Deskriptoren entlang der Baumstruktur nach den ähnlichsten Deskriptoren gesucht. Abschließend wird der euklidische Abstand zwischen den Deskriptor des Eingabebildes und denen aus der Baumstruktur berechnet. Das Merkmalspaare, das den kleinsten Abstand haben und einen Schwellwert unterschreiten, werden dann als Merkmalspaar akzeptiert.

Parallelisierung des Algorithmus Eine mögliche Implementierung der parallel ausführbaren Version des SIFT-Algorithmus ist in Abbildung 12 dargestellt. Dies ist jedoch eine Konzeption der möglichen Implementierung. Sie konnte aus zeitlichen Gründen nicht umgesetzt werden. Die alternative Implementierung hat aufgrund der vereinfachten Parallelisierung wesentlichen einen Nachteil. Wird das Eingabebild zerlegt und dann getrennt voneinander berechnet, gehen Informationen aus den Randbereichen verloren. Dies führt dazu, dass der Deskriptor im originalen Bild an dieser Stelle eine unterschiedliche Charakteristik aufweist. Dadurch wird für diesen Deskriptor kein passender Gegenpart im Referenzbild gefunden.

Das Ergebnis des SIFT-Algorithmus wirkt sich jedoch nicht auf die Untersuchung der Scheduling-Verfahren aus, da hier die Kennzahlen nicht von der Korrektheit des Algorithmus abhängt. Daher wird der parallel ausgeführte SIFT nach der Detektorphase abgebrochen, und die Kennzahlen bis dahin dokumentiert. Um dennoch einen Vergleich mit den seriell ausgeführtem SIFT anstellen zu können, wird eine ausgewählte Testreihe ebenfalls nach der Detektorphase beendet.

Die Implementierung des vereinfachten parallel ausführbaren SIFT-Algorithmus wird im Abschnitt 3.4 beschrieben und leiten die Testreihen der Multi Threaded Variante ein.

Im ersten Schritt wird ein Eingabebild in vier Teile zerlegt und ebenfalls vier Threads erzeugt, deren Aufgabe darin besteht aus dem Eingabebild ein Graustufenbild zu erzeugen. Die Threads erhalten die Bildsegmente und werden auf die vier Kerne des Systems verteilt ausgeführt.
Ist diese Grauton-Konvertierung abgeschlossen erhält der seriell ausgeführte Task die einzelnen Bildsegmente und erzeugt die nächsten vier Threads. Diese Threads werden mit den Bildsegmenten initiiert und erzeugen die Gauß- und DoG-Pyramide. Ist die Erzeugung der Pyramiden abgeschlossen, werden diese an den seriellen Task übergeben und dort zu jeweils einer Pyramide zusammengefasst. Da die Berechnung der Anzahl an Oktaven von der Bildbreite beziehungsweise Bildhöhe abhängt, kann es passieren, das die erzeugte Anzahl an Oktaven nicht die ist, die ein serieller Task generiert hätte. Sind zu wenige Oktave erstellt worden, übernimmt der seriell ausgeführte Teil die zusätzliche Generierung fehlender Oktaven.
Stimmt die Anzahl der Oktaven, werden Beispielsweise 8 Threads erzeugt, die jeweils eine Referenz auf das Bild erhalten. Zusätzlich zu Referenz werden die Threads mit Offsets initiiert. Diese Offsets bestimmen die Start- und Endkoordinaten zwischen denen der Thread die Berechnung der Merkmalspunkte vollzieht. Da auf den Pyramiden lediglich lesen aber nicht geschrieben wird, ist hier ein Nebenläufigkeitsproblem ausgeschlossen. Wurde ein Merkmalspunkt gefunden, wird es in eine globale und vor Nebenläufigkeit geschützte Liste mit Merkmalspunkten eingetragen.
Nachdem die Merkmalspunkte Berechnet wurden, wird der kD-Baum vom seriell ausgeführtem Programmteil aufgebaut. Die Suche mit neuen Merkmalspunkten innerhalb des Baumes kann dann wieder parallel erfolgen.

Eine Idee ist, dass Bild zwar weiterhin zu zerlegen und die einzelnen Pyramiden erzeugen zulassen, danach aber die einzelnen Pyramiden wieder zu einer großen zu vereinen. Da durch die Zerlegung möglicherweise weniger Oktaven erzeugt wurden, können diese gegebenenfalls seriell der vereinten Pyramide hinzugefügt werden.

Hiernach werden neue Threads erzeugt, welche mit Start- und Endkoordinaten initiiert werden. Dies Koordinaten dienen als Bildsegment, in dem der Thread die Merkmalspunkte sucht. Gefundene Merkmalspunkte werden in eine globale Liste eingetragen, damit weitere Threads beginnen können, die Merkmalspunkte mit denen aus dem Referenzbild zu vergleichen. Merkmalspaare werden dann wieder in eine gemeinsame Liste zusammengefasst. Dies sollte das Problem mit den fehlenden Merkmalspaaren lösen.

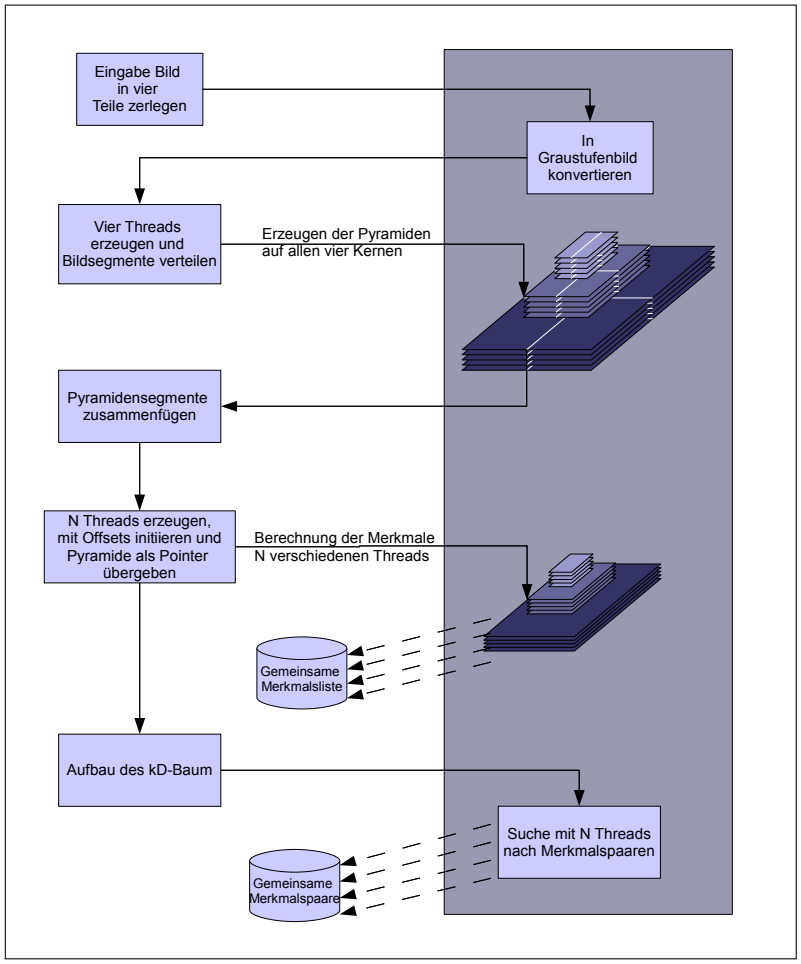

Abbildung 12: Eine Möglichkeit den SIFT-Algorithmus parallel ausführungen zu lassen, ohne dabei den Verlust der Merkmalspunkte in Kauf nehmen zu müssen. Der Dunkel gefärbte Teil stellt den parallel ausführbare Teil des Algorithmus da

3 Untersuchung der Scheduling-Verfahren

Wie bereits eingangs dieser Arbeit erwähnt, entstand diese Arbeit im Zuge einer Diplomarbeit bei ADIT Deutschland und untersucht Scheduling Verfahren auf ihre Fähigkeiten hin, Algorithmen parallel ausführen zu können. Dabei stehen die vom eT-Kernel bereitgestellten Scheduling Verfahren *SPM*, *TSM* und die Mischform *SPM on TSM* im Fokus dieser Arbeit. Diese Verfahren ermöglichen, mit wenig Programmieraufwand, Software von einem SingleCore System auf ein MultiCore System zu portieren. Unter diesen Scheduling Verfahren gibt es teilweise große Unterschiede, bezüglich ihrer Ausführungsrichtlinien und Lastverteilung auf die verfügbaren Prozessoren. Untersucht wurde daher, wie Leistungsfähig diese Verfahren sind und zu welchen Kosten sie diese Leistung erbringen.

Im folgenden Abschnitt werden zunächst die vorbereitenden Arbeiten beschrieben, die nötig waren um die Untersuchung starten zu können. Hier wird zu Anfang die Softwareportierung der SIFT Implementierung und OpenCV beschrieben und dabei auf Probleme und Lösungen beim portieren eingegangen. Im Anschluss der Vorbereitungen wird die eigentliche Untersuchung der Scheduling Verfahren beschrieben. Auch hier wird der Abschnitt in drei Teile aufgeteilt. Der erste Teil beschreibt die Untersuchung des seriell ablaufenden SIFT-Algorithmus und dient gleichzeitig als Vergleichgrundlage für die spätere Untersuchung des parallelisierten SIFT-Algorithmus. Der zweiten Teil der Untersuchung beschriebt die notwendigen Maßnahmen, die ergriffen wurden, um den SIFT-Algorithmus für eine parallele Ausführung vorzubereiten. Der dritte Teil beschreibt letztlich die Untersuchung der Scheduling Verfahren bei der parallelen Ausführung des SIFT-Algorithmus.

3.1 Portierung

Die Quelldateien der SIFT Implementierung lagen zu Beginn dieser Arbeit in C für Linux Betriebssysteme in der Version 1.1.1-20070913 bereit und musste zunächst auf das Betriebssystem eT-Kernel für ARM-Przessoren portiert werden. Da diese SIFT Implementierung auf OpenCV basiert und diese als C/C++ Linux Implementierung in Version 1.1pre1 bereit lag, musste diese ebenfalls nach eT-Kernel portiert werden.

Da eT-Kernel ANSII C unterstützt und die SIFT und OpenCV Implementierungen ebenfalls in ANSII C bzw. C++ implementiert sind, mussten keine Änderungen am Quellcode selbst der Implementierungen vorgenommen werden. Problematisch beim Portieren war jedoch das Ermitteln der korrekten Compiler-Einstellungen. Zwar werden die Implementierungen des SIFTs und OpenCV jeweils mit dem vom *GNU build system* erzeugten Configure-Scripten ausgeliefert. Jedoch ist diesen Scripten ein Cross-Compilieren (also das portieren) nach eT-Kernel und ARM-Prozessoren nicht bekannt. Dieses Problem konnte behoben werden, indem man das Configure-Script zunächst mit den Einstellungen eines Cross-Compilieren startete, die Zielplattform und den Prozessortyp jedoch bei „*unkown*“ beließ. Die fehlenden Einstellungen zum Compilieren zu ermitteln gestaltete sich schwieriger. Zwar sind die Compiler-Einstellungen im eT-Kernel Manual dokumentiert, jedoch hätte ein tiefgreifendes Studium der Optionen zu viel Zeit beansprucht. Eine weitere Möglichkeit bot sich in Referenz-Compilaten des eT-Kernel. Hierzu wurden aus den Compiler-Aufrufen des ARM-Compilers die Einstellungen zum Compilieren

abgeleitet. Nachdem das Configure-Script die nötigen Makefiles erstellt hatte und die zusätzlichen Einstellungen zum Compilieren ermittelt waren, mussten die Compiler-Einstellungen von Hand in die Makefiles nachträglich eingepflegt werden. Für die SIFT Implementierung war das Portieren hier abgeschlossen. Bei der Bildverarbeitungsbibliothek OpenCV hingegen mussten noch eine weitere Anpassung in dem Configure-Skript vorgenommen werden.

Dem OpenCV Configure-Skript wurde zwar mit Startoptionen mitgeteilt, dass es sich um eine Portierung handelt, war aber nicht in der Lage, den Typen der ausführbaren Binärdatei zu bestimmen. Die Bestimmung des Typs der Binärdatei wird in diesem Configure-Script erreicht, in dem das Script versuchsweise Dateien compiliert und nach einem erfolgreichen Compilieren die Datei ausführt. Da aber weder Windows noch Linux ohne entsprechende Erweiterung in der Lage sind eT-Kernel basierende Programme auszuführen, führte dies zu einem Fehler und das Script wurde an dieser Stelle beendet. Gelöst wurde dieses Problem mit einer Anpassung an das Configure-Script. Da der Dateityp bekannt war, wurde dieser Test übersprungen und die Informations um welchen Dateityp es sich handelt wurde von Hand hart in das Script eingefügt.

Zusätzlich zum Anpassen des OpenCV Configure-Skriptes wurden dem Skript noch weitere Einstellungen übergeben. Beispielsweise sind alle nicht notwendigen OpenCV Erweiterungen (z.B. libjpeg, libtiff, ffmpeg, etc. . .) deaktiviert worden, um zu verhindern, dass keine weiteren Programmbibliotheken benötigt wurden.

3.2 Implementierungen

Nach dem Portieren der Software musste noch ein Testszenario erzeugt werden. Hierzu wurden Bibliotheken entwickelt, die den Programmablauf steuern, Messdaten aufzeichnen sowie vorverarbeiten können. In Tabelle 1 befindet sich eine kurze Übersicht der entwickelten Bibliotheken. In den folgenden Absätzen werden die Funktionalitäten der Bibliotheken beschrieben.

3.2.1 Lib: CP15_Ops

Der CP15 ist einer der beiden Co-Prozessoren des ARM-11 MP-Cores und besitzt 15 spezielle Register, mit denen sich die vier Prozessorkerne überwachen und steuern lässt. In dieser Arbeit wurden die Register c7 und c15 verwendet.

c7 - Cache Operation Register Mit diesem Register lässt sich sich der Instruktion und Datencache der Prozessorkerne invalidieren und/oder „säubern“ (cleaning). Eine vollständige Beschreibung der Funktionen dieses Register entnimmt man der ARM-11 Referenzbeschreibung [ARM Ltd.(2009)ARM-11MPCore].
Um bei jedem Start einer Messreihe die gleichen Bedingungen zu erzeugen, wurde vor jedem Start einer Messreihe der Daten- und Instructionscache invalidiert und “gesäubert” (invalidate&clean). Diese Funktione musste in Assembler implementiert und innerhalb eines Subsystems (siehe 3.2.3) des Betriebssystems veröffentlicht (ex-

Bibliothek	Funktionen
CP15_Ops	Initialisieren und Auslesen des CP15 C15 des ARM11MPCore
	Invalidieren & säubern des Data- und Instructionscaches
Scrambler	Migrationen erzwingen durch Ausführen eines hoch priorisiertem Tasks
	Invalidieren & säubern des Data- und Instructionscaches
Subsystem Calls	Auslesen von T-Kernel Systemuhren
	Setzen einer logischen Kern ID (nur SPM und SPM on TSM)
Dipl Utils	Enthält den gesamten Inhalt der zusätzlich Implementierten Funktionen dar. Enthält Funktionen zum Auslesen verschiedener Messdatensätze, Aufbereiten der Daten, erstellen der Log-Dateien aus den Datensätzen.

Tabelle 1: Innerhalb dieser Arbeit entstandenen Programme und Bibliotheken

portiert) werden. Das Exportieren war nötig, damit das "Aufräumen" der Caches direkt aus dem Testprogramm aufgerufen werden konnte.

c15 - Performance Monitor Register Das Register c15 kontrolliert neben anderen Funktionen, welche Ereignisse PNM0 und PNM1 zählen sollen. Eine vollständige Beschreibung der Funktionen entnimmt man der Spezifikation des ARM11 MPCore [ARM Ltd.(2009)ARM-11MPCore].
Implementiert wurde zum einen die Initialisierung des Registers, zum anderen die Funktion zum Auslesen des Registers. Zusätzlich musste hier, analog zum c7 Register, ein Subsystem[9] des Betriebssystem erzeugt werden um die implementierten Funktionen Systemweit bereitzustellen.

3.2.2 Lib: Scrambler

Die Scrambler-Bibliothek stellt zwei Funktionen um auf dem Testsystem für "Unruhe" zu sorgen zur Verfügung. Zum einen wurde ein *Datencache-Scrambler* implementiert, zum anderen eine Funktion, die das Betriebssystem dazu veranlasst das Testprogramm von einem Kern zum nächsten zu migrieren. In den folgenden Absätzen wird die Funktionsweise dieser beiden Scrambler erläutert.

Der Datencache Scrambler Der Scrambler-Task wird bei seiner Kreierung direkt auf einen Prozessorkern festgelegt und muss immer dem *SPM* oder *SPM on TSM* Verfahren zugeordnet werden. Damit wird verhindert, dass der Task vom Betriebssystem auf einen anderen Kern migriert wird. Somit wird sichergestellt, dass nicht zwei Scrambler-Tasks auf einem Kern für zusätzliche Cachemisses sorgen, bzw. ein Kern ohne Scrambler-Task besetzt ist.
Damit die Anzahl der zusätzlichen Cachemisses kontroliert werden kann, wurde

[9] Enthält Module, die dynamisch dem Betriebssystem hinzugefügt werden können.

innherhalb dieser Tasks zusätzlich eine Timer-Funktion hinzugefügt. Der Timer kann im Nanosekundenbereich eingestellt werden. Für die Testreihen jedoch haben sich Pausen im Millisekundenbereich als praktikabel erwiesen.

Die Funktionsweise gibt es in zwei verschiedene Ausführungen.

- Nutzung der Invalidierungsfunktion aus der CP15_ops-Bibliothek
- Nutzung der internen Scramble-Funktion. Diese Funktion beinhaltet ein 2-dimensionales Array, dessen gesamte Indizies genau einmal vollständig neu geschrieben werden. Bei der Ausführung der Funktion wurden damit immer 16KByte (2xfache Menge der Cachekapazität) in den Cache laden. Diese Menge reicht aus, damit alle Cachelines vollständig zurück in den Hauptspeicher geschrieben werden. Da der Task immer in definierten Frequenzen ausgeführt wird, erhöhen sich so gemessene Cachemisses.

Der Task-Pusher Der Task-Pusher wird, ähnlich wie der Datencache Scrambler, bei der Kreierung bereits auf einen Prozessorkern festgelegt (setzt eine Variante des *SPM* oder *SPM on TSM* voraus). Insgesamt werden vier dieser Tasks kreiert und auf die Kerne verteilt ausgeführt. Die Funktionsweise macht sich das Prioritäten gelenkte Scheduling des Betriebssystem zu nutze. Der Task-Pusher wird mit einer Priorität gestartet. Wird der Task-Pusher aktiv, wird das laufende SIFT-Algorithmus vom Betriebssystem auf einen andern Kern migriert.

3.2.3 Verwendete Subsysteme

Die des eT-Kernels Subsysteme wurden als Bindeglied zwischen Betriebssystem und Testanwendung benötigt, damit man Funktionen aus der Testanwendung heraus mit privilegiertem Rechten ausführen kann. Die oben beschriebenen Funktionen müssen mit privilegiertem Rechten ausgeführt werden, da sie sonst ohne Wirkung auf das Testsystem wären.

Control Prozessor Register API - Register Initiieren und Auslesen Das Subsystem beinhaltet die Funktionen zum Initialisieren und Auslesen der Register c7 (siehe 3.2.1) und c15 (siehe 3.2.1) und stellt somit die API der implementierten Scrambler-Bibliothek dar.

Zeitnehmer API - Auslesen der Systemzeit Neben den Regsiteroperationen stellt das Subsystem die Funktionen zum Auslesen der aktuellen Systemzeit zu Verfügung. Dies wird deshalb auf dieser Ebene durchgeführt um möglichst exakte Messdaten zu erhalten. Die verfügbare Posix-Funktion *clock_gettime()* hatte Messungenauigkeiten (σ $1,03s$) und wurde daher nicht verwendet. Im Vergleich dazu lag σ bei der nativen eT-Kernel Funktion *get_time_otm* bei $0,34$.

LCID API - Migrieren von Hand Die LCID API ermöglicht, einen Task von einem Kern zu einem anderen migrieren zu lassen. Diese Art der Migration ist eine spezielle Funktion des eT-Kernels. Damit die, in der Multi Threaded Variante durchgeführten, Tests auf alle Kerne verteilt werden konnten, musste diese Funktion für die Testanwendung bereitgestellt werden.

Option	Wert	Bedeutung
Trials	50	Anzahl der Testdurchläufe
Sign	1	Eingabebild in dem nur das zu findene Zeichen enthalten ist
Scene	1	Eingabebild in dem die ganze Szenerie enthalten ist

Tabelle 2: Zusammenfassung der statischen Testparameter. Sie gelten, wenn nicht anders erwähnt, für alle Testreihen der seriellen Ausführung des SIFT-Algorithmus

3.2.4 DiplUtils Bibliothek

Diese Bibliothek beinhaltet alle IO- und Utility-Funktionen, die für die Dokumentation der Testreihen nötig waren. Einen vollständigen Überblick der Funktionen befindet sich in der Quellcode Dokumentation.

3.3 Testreihen der seriellen Variante

Im folgendem Abschnitt werden die Testreihen und Ergebnisse des seriell ausgeführtem SIFT-Algorithmus beschrieben. Um einen Überlick über die durchgeführten Testreihen zu geben, begint der Abschnitt mit einer kurzen Zusammenfassung der Testreihen. Im Anschluß daran werden die einzelnen Testreihen genauer beschrieben und die Ergebnisse diskutiert.

3.3.1 Statische Testkonfiguration

Für die Testreihen der seriellen Ausführung des SIFT-Algorithmus werden nun zunächst die statischen Konfiguration der Testreihen beschrieben. Diese Konfiguration ändert sich nur dann, wenn es ausdrücklich im Testaufbau der Testreihe genannt wurde. Tabelle 2 gibt einen Überblick über diese statischen Einstellungen.

Ein Trial ist eine vollständige Ausführung des SIFT-Algorithmus. Die Anzahl der Trials liegt bei 50 Stück. Mit diesen Trials soll vermieden werden, dass Ausreißer in den Messwerten die Resultate verfälschen.

Das Zeichen- und **Szenenbild** sind die standardmäßig geladenen Eingabebilder (Abb. 13). Das Zeichenbild beinhaltet das gesuchte Straßenverkehrszeichen. Im Szenenbild ist das gesamte Szenerie enthalten. Diese Bilder wurden gewählt, weil es typische Merkmale zeigt, wie Straßenschilder im ländlichen Gegenden aufgestellt sind und gesehen werden.
Das Zeichenbild enthält das Zeichen, dass vom SIFT-Algorithmus im Szenenbild wieder erkannt werden soll.

3.3.2 Zusammenfassung der Experimente

Zu Beginn der Experimente der seriellen Ausführung des SIFT-Algorithmus wurden die Ausführungszeiten der drei verschiedenen Scheduling Verfahren durchgeführt.

Abbildung 13: **Links:** Szenenbild in dem die Merkmalspunktes gesucht werden, die bereits vorher im Zeichen Bild berechnet wurden.
Rechts: Zeichenbild. Hier werden die Merkmalspunkte berechnet, die zur Identifikation des Schildes verwendet werden.

Scheduling Mode	Ausführungszeiten	Differenz
SPM on TSM	$19s$	$340ms$
SPM	$19,08s$	$450ms$
TSM	$18,62s$	$0ms$

Tabelle 3: Resultat der initialen Testreihe des seriell ausgeführtem SIFT-Algorithmus. TSM Bildet die Basis der Differenzbildung und hat daher die Differenz $0ms$

Dieser Test wurde auf dem Betriebssystem-Prozessor durchgeführt. Bei den Resultaten ergab sich ein Ergebnis (siehe Tab. 3), welches man nicht erwartet hatte. Die Ausführung im *TSM* Modus war $\approx 380ms$ bis $\approx 450ms$ schneller als jene im *SPM*- und *SPM on TSM*-Modus. Dies führte zur Annahme, dass der Scheduler im *TSM*-Mode die Ausführung des Tasks auf einen anderen Kern migrierte. Diese Annahme wurde in einer weiteren Testreihe, in der die Migrationen mitgezählt wurden, bestätigt. Zusätzlich wurden zum Vergleich auch die Testreihen im *SPM*- und *SPM on TSM*-Modus auf einem anderem Kern ausgeführt. Die Ausführungszeiten (siehe Tab. 5) verkürzten sich zwar, jedoch war die Ausführung im *TSM* Modus noch immer schneller.

Zum Abschluss der Experimente, standen noch die Testreihen zur Bewertung der Migrationen aus. Migrationen sind auf Grund der erhöhten Speicherbusbelastung bei der Betrachtung der Systemperformance von besonderem Interesse. Aus diesem Grund wurde die Testreihe für *TSM* wiederholt und zusätzlich vier Stör-Tasks aktiviert. Diese Tasks (verteilt auf alle Kerne) hatten die Aufgabe, durch ihre hohe Priorität, den Scheduler zu veranlassen, den im *TSM*-Mode ausgeführten Task von Kern zu Kern zu migrieren. Die Resultate dieser Testreihe waren bemerkenswert. Der *TSM* ausgeführte Task hatte noch immer eine kürzere Ausführungszeit, als jene,

Scheduling Mode	Ausführungszeiten	Differenz
SPM on TSM	$18,7s$	$120ms$
SPM	$18,8s$	$179ms$

Tabelle 4: Resultat der Testreihe zur Validierung der These bzgl. Task-Migration. Die Ausführungszeit der vorhergehenden Testreihe des TSMs-Versuchs (Siehe Tabelle 3) bildet auch hier die Basis zur Differenzbildung

Scheduling Mode	Migrationen	Ausführungs-zeiten	Differenz
TSM ohne Stör-Task	0	$12,245s$	$0ms$
TSM mit Stör-Task bei $33,33Hz$	631	$12,283s$	$38ms$
TSM mit Stör-Task bei $50Hz$	751	$12,391s$	$146ms$
SPM on TSM ohne Stör-Task	0	$12,39s$	$120ms$
SPM ohne Stör-Task	0	$12,65s$	$179ms$

Tabelle 5: Resultat der Testreihe zur Ermittlung der Migrationskosten. Diese Testreihe wurde ohne Deskriptorphase durchgeführt. Die Differenzen der im *SPM* und *SPM on TSM* Modus durchgeführten Testreihen sind mit der längsten Ausführungszeit der im *TSM*-Modus durchgeführten Testreihe gebildet worden

bei denen der *SPM(on TSM)*-Mode verwendet wurde. Es konnte gezeigt werden, dass die DDI-Erweiterung des MESI-Protokols die Kosten durch Migrationen nur minimale Auswirkungen auf ein zeitaufwändigen Task hat.

3.3.3 Experimente im Detail

In diesem Abschnitt folgt nun eine detailierte Beschreibung der durchgeführten Untersuchung. Zudem werden die Ergebnisse präsentiert und diskutiert.

Wie bereits in der Zusammenfassung oben angedeutet, rücken zum einen die reinen Ausführungszeiten des Algorithmus, als auch die Kosten durch Migrationen in den Fokus der Testreihen. Denn anders als bei Rechnersystemen mit einem Kern, können Tasks auf einem Mehrkern System von einem Kern auf einen anderen migrieren und dort weiter ausgeführt werden. Diese Migrationen sind mit Kosten (Zeitaufwendungen) verbunden, da es dazu nötig ist die betroffenen Cachelines auf den neuen Kern zu übertragen, bevor der Task wieder weiter ausgeführt werden kann. Die Übertragung dieser Cache-Inhalte wird über den Speicherbus realisiert, was dazu führt, dass weitere Daten über den Speicherbus übertragen werden. Dies führt dann wiederum dazu, dass das System verlangsamt werden kann (Kapitel 2.2.2 auf Seite 15). Der ARM11 MPCore verfügt jedoch über die *DDI*[10]-Erweiterung des MESI-Modells, die es ihm ermöglicht, die von der Migration betroffenen Cachelines direkt an einen anderen Kern zu schicken, ohne dafür den Speicherbus zubenutzen. Die folgenden Testreihen sollen zeigen, wie leistungsfähig dieses Feature ist. Darüberhinaus soll mit den Ergebnissen eine Aussage treffen getroffen werden, wie

[10] Direct Data Intervention

teuer Migrationen trotz MESI-Erweiterung DDI sind und wie effizient der Einsatz der drei Scheduling-Verfahren bei der Ausführung des SIFT-Algorithmus ist.

Initiale Testreihe Um eine Einschätzung des Leistungspotentiales der Scheduling-Verfahren treffen zu können, wurde in dieser Arbeit zunächst auf dem Testsystem der SIFT-Algorithmus (Kapitel 2.4.4) seriell ausgeführt und die benötigte Ausführungszeit dokumentiert. Ausserdem dienen die Resultate aus dieser Testreihe als Basis zum Vergleichen der nachfolgenden Testreihen.

Testaufbau Die Konfiguration dieser Testreihen bestand aus den im Abschnitt 3.3 beschriebenen statischen und den folgenden dynamischen Parametern:

- **Die Scheduling-Verfahren** *SPM*, *TSM* und *SPM on TSM*. Das vierte verbleibende Verfahren *SRL on TSM* wurde in dieser Arbeit nicht untersucht, da der Algorithmus keine Programmteile besitzt, die bei einem parallelem Verarbeiten im Konflikt zu einander stehen. Um die drei Scheduling-Verfahren bewerten zu können, wurden Geschwindigkeitstests durchgeführt und Informationen zu Migrationen und Cachingverhalten erzeugt.
- **Der CPU Kern**, auf dem der Task gestartet wurde. In den Fällen *SPM* und *SPM on TSM* ist der Task auf dem Ursprungskern verbleiben. Bei *TSM* könnte der Task migrieren und auf einem anderen Kern weiter ablaufen.
- Die Messpunkte dieser Testreihe wurden so gesetzt, dass die Berechnungszeit des Zeichen- und Szenenbildes gemessen wurden. Zusätzlich wurden Messpunkte für die gesamte Durchführungszeit des Algorithmus gesetzt.

Die Wahl des CPU-Kerns basiert in dieser Testreihe auf der Tatsache, dass der CPU-Kern 0 der Betriebssystem-Kern (oder auch OS-Core[11] genannt) ist. Dieser Kern wurde ausgewählt, da auf diesem Kern ähnliche Verhältnisse herrschen, wie bei einem System mit nur einem CPU-Kern (auch hier teilen sich System- und Anwenderprozesse die CPU-Zeit).

Vorbereitungen Eine Zeitmessung ist mit der von Rob Hess implementierten Version des SIFT-Algorithmus nicht möglich. Aus diesem Grund musste zuvor eine Funktion und eine Struktur zum abspeichern der Messdaten implementiert werden. Die Funktion wurde dann an den Stellen im Algorithmuss eingesetzt, an denen die gerade anliegende Systemzeit gemessen wurde. Nach der Bestimmung des Zeitpunktes wurde der Wert in die Struktur gepeichert. Nach jedem vollständigen Durchlauf des Algorithmus wurde die Struktur ausgelesen und die Ausführungzeit berechnet. Anschliessend wurden diese Daten für spätere Analysen fortlaufend an das Ende eine Datei geschrieben.

Resultate und Bewertung Das Resultat dieser Testreihe ergab, dass das Scheduling-Verfahren *TSM* die Ausführung des SIFT-Algorithmus in kürzester Zeit durchführen lies. Die Testreihe mit *TSM* verwaltetem Algorithmus sind durchschnittlich $\approx 380ms$ - $450ms$ schneller als die Ergebnisse aus den Versuchen

[11] Operating System Hosting Core

mit *SPM* und *SPM on TSM* (siehe Abb. 14).

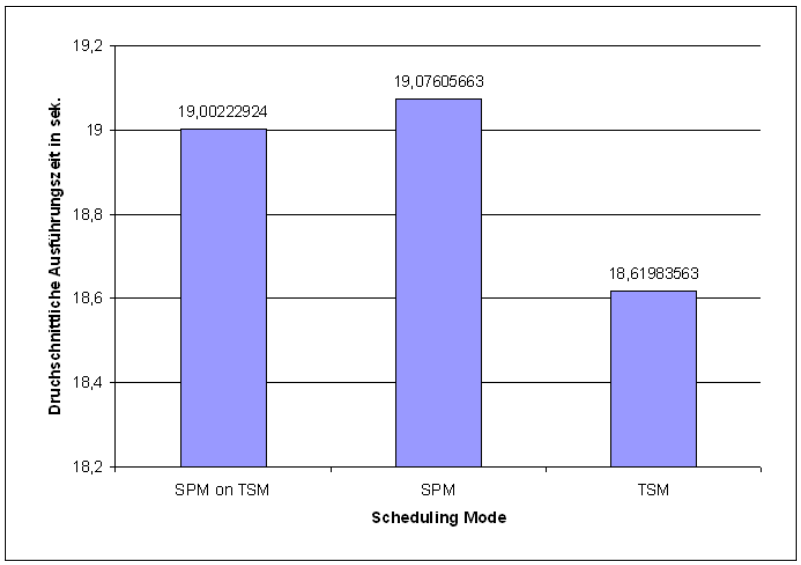

Abbildung 14: Initiale Testreihe der seriellen Variante:
Darstellung der Ausführungszeiten der Scheduling Modi. der *TSM*-Modus beschleunigt die Ausführung am stärksten

Eine mögliche Erklärung war, dass der Scheduler im *TSM* Verfahren, den Task des Algorithmus auf einen anderen Kern der CPU migrierte. Ist der Task erst einmal auf einen anderen Kern gewechselt, so kann dieser die gesamte Ressource CPU (und somit dessen Cache) für sich allein beanspruchen. Andere Tasks, die einen ungehinderten Ablauf auf dem neuen Kern verhindern hätten können, gab es nur auf dem BS-Kern.

Verifikation: Migration Zum Bestätigen der Annahme das Migrationen die Ausführung des Algorithmus begünstigen, wurden drei weitere Testreihen bestimmt. Es sollen die Task-Migrationen während eines Durchlaufs gemessen werden.

Vorbereitungen Migrationen lassen sich nicht mit den im eT-Kernel bereitgestellten Funktionen messen. Hierfür wurde der EvenTrek [Gondo(2006)] des eBinders eingesetzt. Dieses Tool ist in der Lage, Ereignisse wie Task-Migrationen aufzuzeichnen und zu speichern.

Testaufbau Zum Bestätigen der Annahme, dass Migrationen eine Task-Ausführung begünstigen kann, wurden drei weitere Testreihen durchgeführt, in denen die Scheduling-Verfahren *SPM* und *SPM on TSM* auf einem freien[12] Kern ausgeführt und die Ausführungszeiten dokumentiert wurden. Darüber hinaus wurde

[12] Frei meint in diesem Zusammenhang einen Kern auf dem keine weiteren Tasks oder Prozesse ausgeführt werden

ein weiterer Test mit dem *TSM*-Verfahren durchgeführt und das Migrationsverhalten aufgezeichnet[13].

Resultate und Bewertung Die Beobachtung des Migrationsverhaltens hatte ergeben, dass der im *TSM* ausgeführte Task tatsächlich von dem BS-Kern zu einem anderen Kern migriert wurde und dadurch ungehindert fortgesetzt werden konnte.

Untermauert wird diese Aussage, durch die Ergebnisse der im *SPM* bzw. *SPM on TSM* ausgeführten Programmabläufe. Wie in Abbildung 15 zu sehen, hatte sich die Ausführungszeit in beiden Fällen verkürzt und den Werten aus dem ersten *TSM*-Test angeglichen. Dennoch hat die im *TSM* ausgeführte Testreihe kürzere Ausführungszeiten ergeben. Diese Beobachtungen deuten darauf hin, dass der Algorithmus migriert wird, sobald ein anderer Task mit höherer Priorität zur Ausführung auf dem aktuellem Kern bereitsteht.

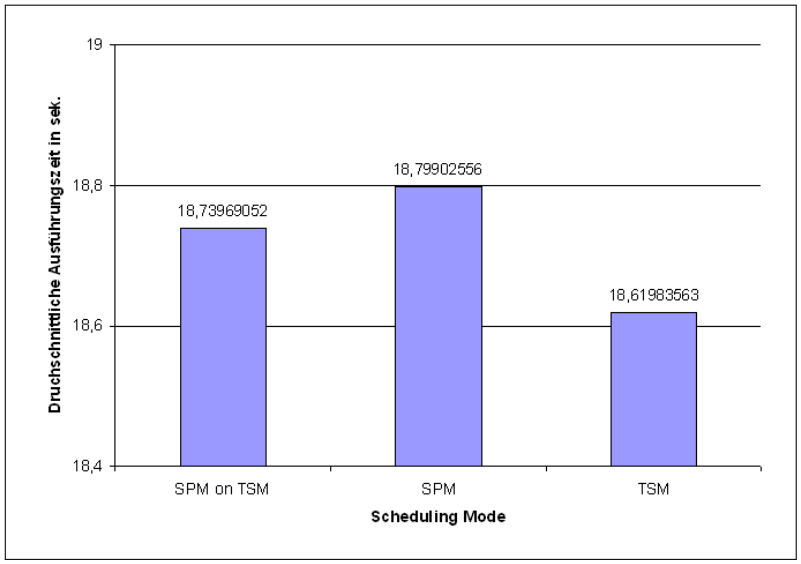

Abbildung 15: Durchschnittliche Ausführungszeiten in Sekunden. Bei den Scheduling Modi *SPM* und *SPM on TSM* wurde der Algorithmuss auf dem Kern 4 ausgeführt. Die Ausführungszeit für *TSM* ist aus dem vorhergehenden Test übertragen worden

Ermittlung der Kosten durch Migration Mit den bereits durchgefürhten Testreihen wurde gezeigt, dass sich mit Verwendung der MESI-Erweiterung DDI, Migrationen verkürzend auf die Ausführungszeit eines Tasks auswirken. Der SIFT-Algorithmus ist jedoch, verglichen mit einem Treiber oder einer Interrupt-Behandlung, ein sehr zeitaufwendiger Task. Zu klären war also, wieviele Kosten

[13] Da das Aufzeichnen der Migrationen nur mit EvenTreck - einer System-Monitor-Applikation zum Überwachen des Betriebssystems eT-Kernel - möglich ist und dieser die Ausführungszeiten beeinflusste, mussten die Zeitmessungen und Migrationsmessungen getrennt voneinander durch geführt werden

eine Migration trotz DDI verursacht und wie sie sich auf Tasks auswirken, die ähnlich wie Treiber eine kurze Ausführungsdauer haben.

Vorbereitungen Diese Testreihe wurde nicht wie die bisherigen Testreihen mit einen vollständigen Durchlauf des Algorithmus durchgeführt. Der Programmablauf wurde nach der Detektorphase abgebrochen, da die folgende Testreihe dazu verwendet werden soll, Vergleiche mit der parallel Variante zu ziehen, wurde beschlossen, diese Testreihe ebenfalls nach der Detektorphase abzubrechen.

Ähnlich wie in der Testreihe zuvor, wurde der EvenTreck verwendet um die Migrationen aufzuzeichnen. Problematisch an diesem Messwerkzeug ist, dass man eine Messung nicht über einen definierten Zeitraum hinweg durchführt, sondern erst dann ein Ergebnis geliefert wird, wenn der ereignisaufzeichnende Puffer des EvenTreks vollgeschrieben wurde. Dieser Umstand führte dazu, dass jede Messung unterschiedlich lang dauern konnte.

Eine weiter Vorbereitung lag darin, das Scheduling-Verfahren zuveranlassen, den Task zumigrieren. Denn ein Task wird nur dann von einem Kern migriert, wenn das Scheduling-Verfahren einen weiteren Task vorfindet, der auf diesem Kern ausgeführt werden soll und dieser eine höhere Scheduling-Priorität besitzt. Aus diesem Grund wurde ein Task implementiert, der sich das Prioritätengesteuerte Scheduling zu nutze macht und den Scheduler veranlässt, den Algorihmus auf einen anderen Kern zu migrieren.

Für die Zeitmessung der folgenden Testreihen wurden die bereits platzierten Messunkte aus den vorangegangen Testreihen verwendet.

Testaufbau Die Messung der Kosten durch Migration wurde daraufhin ausgelegt, über empirische Daten einen Mittelwert der Migrationen zu erhalten. Das folgende Experiment beinhaltet 10 Messungen, in denen ausschliesslich die durchgeführten Migrationen gezählt wurden. Diese Messungen wurden wie bereits erwähnt mit dem EvenTreck durchgeführt und beinhalten unterschiedlich lange Messzeiträume. Die Messdauer dieser Experimente variiert daher zwischen

- $10,3s$ und $13,0s$ bei der Messung ohne Stör-Task,
- $9,0s$ und $9,6s$ bei der Messung mit Stör-Task bei $33,33Hz$ Ausführungshäufigkeit und
- $7,5s$ und $7,9$ bei der Messung mit Stör-Task bei $50Hz$ Ausführungshäufigkeit.

Um vergleichbare Ergebnisse zu erhalten, wurde die Migrationsdichte pro Sekunde der einzelnen Testläufe berechnet.

Die Tatsache, dass nur die längsten Messungen ohne Stör-Task lang genug liefen, bis der Algorithmus vollständig fertig durchgeführte wurde, verändert die Qualität der Messung nicht. Denn anders als bei den vorangegangen Testreihen, werden die Kosten hier nicht durch den vollständigen Programmdurchlauf bewertet, sondern durch die Anzahl der Migrationen während des Messzeitraumes.

Der in den Vorbereitungen erwähnte Stör-Task wird bei seiner Kreierung mit einer hohen Scheduling-Priorität versehen. Das veranlasst den Scheduler dazu, den SIFT-Algorithmuss auf einen anderen Kern zu migrieren, wenn beide der folgenden Forderungen erfüllt sind:

1. Der Algorithmuss befindet sich in einer MCU (siehe 2.4.2) und gehört dem *TSM* an (Bei *SPM(on TSM)* ist Task-Migration nicht erlaubt).

2. Es existert ein Kern, der die Ausführung des Algorithmus übernehmen kann.

Der Stör-Task wurde viermal kreiert und je einem Kern zugewiesen. Dort starteteten diese Tasks ihre Ausführung (siehe 3.2.2) und liessen den SIFT-Algorithmus von Kern zu Kern migrieren.

Zusätzlich zu den Messungen der Migrationskosten wurden separat auch Zeitmessungen mit aktivem Stör-Task durchgeführt. Auch hier wurde die oben genannte Konfiguration bzgl. der Frequenz des Stör-Tasks verwendet. Die Zeitenmessungen wurden im Anschluß dieser Testreihen mit der Anzahl der Migrationen zusammengeführt und untersucht.

Testdurchführung Wie weiter oben bereits kurz angedeutet, werden alle Testreihen mit einem Stör-Task durchgeführt, der je nach Testreihe eine andere Ausführungsfrequenz besitzt. Die erste Testreihe wurde mit einer Frequenz von $33,33Hz$ durchgeführt. Die zweite Reihe mit einer Frequenz von $50Hz$. Wie ebenfalls bereits erwähnt, wurde der SIFT-Algorithmus bis einschliesslich Detektorphase durchgeführt und die Anzahl der Migrationen und im zweiten Testdurchgang die Ausführungszeit dokumentiert.

Resultate und Berwertung Die Testreihen zur Zeitmessung ergaben eine durchschnittliche Ausführungszeit von $\approx 12,24s$ ohne Stör-Task, $\approx 12,28s$ mit Stör-Task bei $33,33Hz$ und $\approx 12,39s$ mit Stör-Task bei $50Hz$.

Die Testreihen zur Zählung der Migrationen ergaben eine ohne Stör-Task ein Migrationsdichte im Durchschnitt $\approx 36,9$ Migrationen. Bei einer Ausführunsfrequens von $33,33Hz$ waren es $\approx 68,2s$ und $\approx 100,3$ bei einer Frequenz von $50Hz$.

Im nächsten Schritt wurde die Anzahl der Migrationen auf die jeweilige Ausführungszeit hochgerechnet, woraus sich folgenden Werte ergaben Ohne Stör-Task ≈ 451, Mit Stör-Task bei $33,33Hz \approx 837$ und Mit Stör-Task bei $50Hz \approx 1242$

In Abbildung 16 werden diese Ergebnisse noch einmal veranschaulicht. Aus Gründen der Übersicht wurden in der Abbildung nur die prozentualen Unterschiede dargestellt. Die Testreihe ohne Stör-Task bildet hierbei die Basis. Deutlich zu erkennen ist der Anstieg der Migrationen durch den Stör-Task. Bei $33,33Hz$ wurde ein Anstieg von $\approx 86\%$ und bei $50Hz$ ein Anstieg von $\approx 175\%$ gemessen. Im Vergleich zu den Migrationen steigt die Ausführungszeit nicht signifikant an. Nahmen die Migratioenen um den Faktor $1,75$ (Stör-Task bei $50Hz$) zu, stieg die Ausführungszeit lediglich um $\approx 1,2\%$.

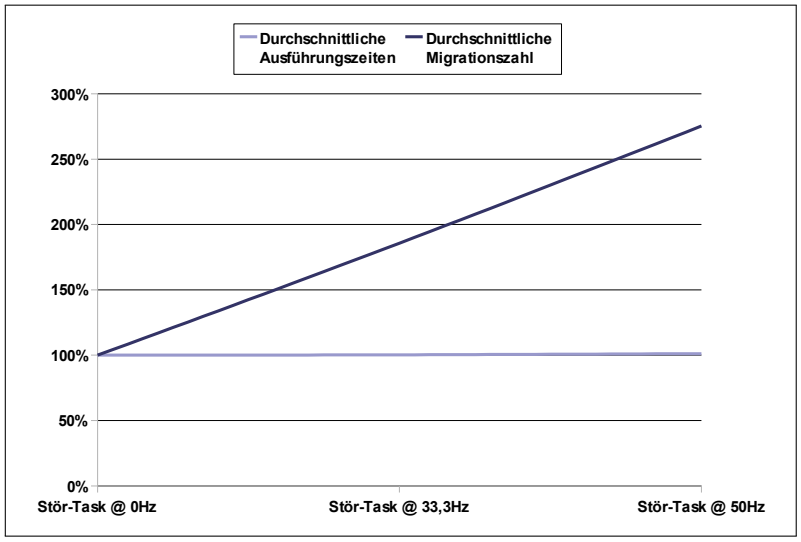

Abbildung 16: Durchschnittliche Ausführungszeiten nach 50 Testläufen in Sekunden

Daraus folgt, dass Migrationen mit der MESI-Erweiterung DDI bei zeitintensiven Rechenprozessen keine signifikante Verlängerung der Ausführungszeit bewirken. Die Einschätzung der Migration für zeitlich sehr kurz ablaufende Tasks, wie Treiber oder Interrupts, musste nun die reale Migrationszeit ermittelt werden. Jedoch ist diese mit den in dieser Arbeit zur Verfügung stehenden Mittelb nicht direkt messbar gewesen. Die Messung hätte vorausgesetzt, dass die Zeitmessung zuverlässig mit einer Auflösung im Nanosekundenbereich arbeitet. Diese Auflösung kann jedoch nicht mit denen im der Betriebssystem API zur Verfügung stehenden Funktionen gewährleistet werden. Für Messungen im Millisekundenbereich ist die Systemuhr jedoch hinreichend genau und daher nutzbar.

Um dennoch eine Einschätzen zu erhalten, wurden die bisherigen Resultate dazu verwendet, die Migrationszeit Z zu Berechnen, basierend darauf wie stark die laufzeit mit der Anzahl an Migrationen ansteigt. Z ist definiert als

$$Z(N, M, T) = \frac{\sum_{n=1}^{N} \frac{t_n - t_0}{m_n - m_0}}{N} \tag{6}$$

mit $N =$ Messreihen und T Ausführungszeit in Sekunden, $t \in T$, $M =$ Migrationen und $m \in M$.
Das Ergebnis dieser Berechnung ist eine Migrationszeit von durchschnittlich $\approx 90\mu s$

3.4 Vorbereitungen der Testreihen der Multi Thread Variante

Nachdem die Testreihen des im Single Thread laufenden Algorithmuss abgeschlossen waren, wurde die Implementierung des SIFT-Algorithmus dahingehend geändert, dass mehrere Threads parallel die Berechnung der Merkmalspunkte durchführen konnte. Der folgende Abschnitt beschreibt die nötigen Änderungen des Algorithmus.

Für die parallele Ausführung des SIFT-Algorithmus musste der Algorithmus

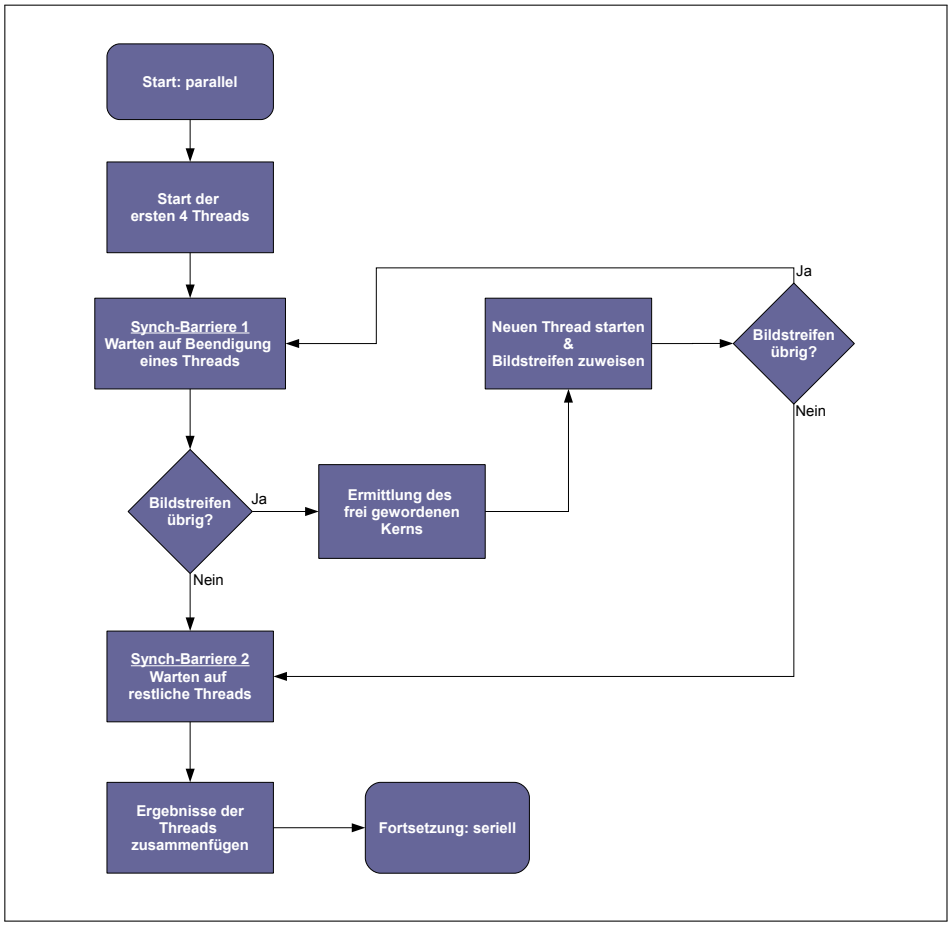

Abbildung 17: Algorithmus zur Verteilung und Verwaltung der Bildstreifen auf die CPU Kerne

zunächst angepasst werden. In diesem Zusammenhang wurde die Implementierung des Algorithmus in drei Teile zerlegt und um die nötigen Maßnahmen - z.B.: Zerlegen des Szenenbildes (siehe Add. 19 Seite 51 - erweitert. In folgenden Absätzen wird beschrieben welche Erweiterung des Algorithmus im Zuge dieser Arbeit nötig waren, um Teile des SIFT-Algorithmus parallel ausführen zu können.

Erster Teil - Vorbereitungen Das Bild wird in Quadranten oder Streifen (je nach Testszenario) zerlegt. Hierzu verwendet man die Image-Splitt-Funktion aus der *DiplUtils* Bibliothek und erhält eine Liste aus Bildquadranten/-streifen. Diese Bildquadranten/-streifen werden später auf Threads verteilt und (zum Teil) dynamisch auf die Kerne verteilt.

Die Implementierung der Threads ist in Abbildung 18 dargestellt. Zu erkennen ist, dass die Threads die selben Barrieren wie die Threadverwaltung besitzt. Damit wurde zum einen erreicht, dass ein Thread nachdem er das Bildsegment berechnet hat, automatisch der Verwaltung ermöglicht ihren nächsten Schritt einzuleiten

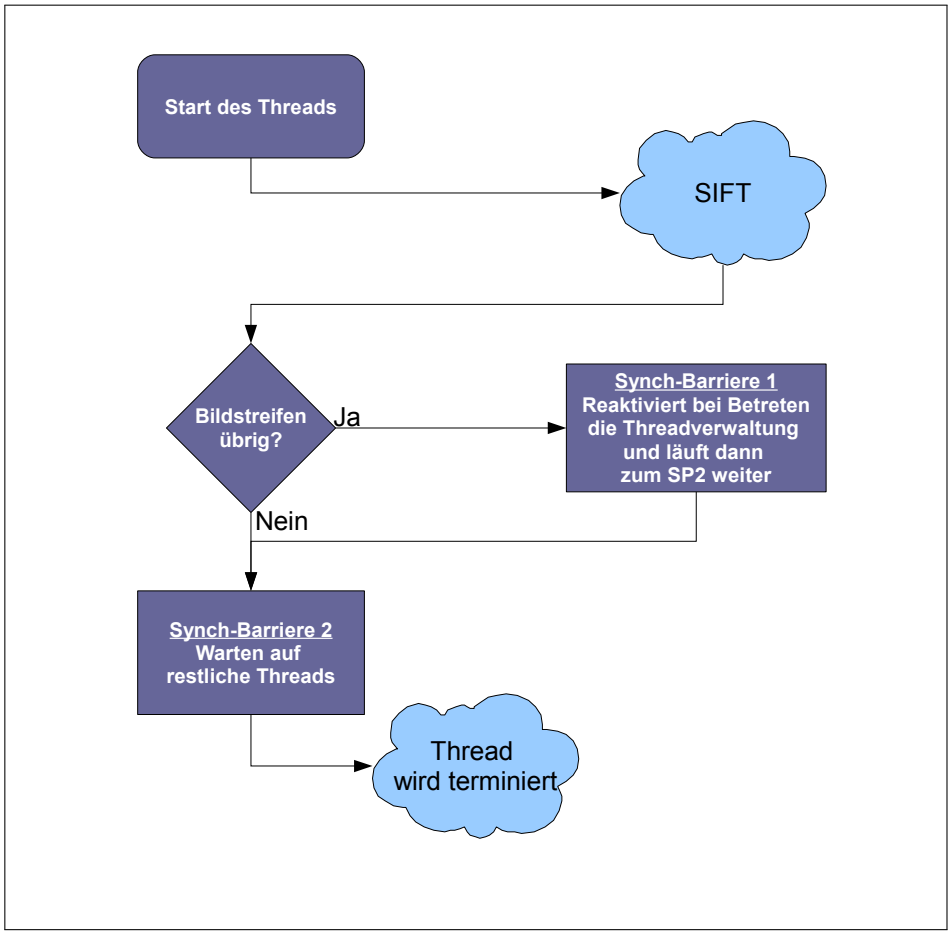

Abbildung 18: Algorithmus zur Verteilung und Verwaltung der Bildstreifen auf die CPU Kerne

(SP1). Zum anderen, wird der Thread nicht direkt nach seiner Berechnung wieder beendet (SP2). Dies hat den Vorteil, das die Verwaltung Informationen darüber benötigt, welcher Thread noch aktiv ist oder nicht. Erst wenn alle erzeugte Threads die Barriere betreten haben, öffnet sich diese wieder und die Threads werden beendet.

Nachdem die Bildsegmente vorbereitet wurden, wird eine weitere Liste erzeugt, in denen die Threads für parallele Ausführung bereitgestellte werden. Die Threads werden mit ihren Startparametern initialisiert. Im Folgenden werden nur die für die SIFT-Verarbeitung relevanten Parameter beschrieben.

- **Thread ID:**
  Dient zur Identifikation eines Threads.
- **CPU ID:**
  Bei *SPM (on TSM)* kann mit dieser ID ermittelt werden, auf welchem Kern der Threads ausgeführt wurde.

- **Bildsegment:**
  Für dieses Bildsegment werden die Merkmalspunkte berechnet.

Zweiter Teil - Starten und Verwalten der Threads In Abbildung 17 wird der Algorithmus der Threadverwaltung dargestellt. Im folgenden wird kurz die Arbeitsweise der Threadverwaltung beschrieben.

Die Threadverwaltung startet zunächst vier Threads (falls weniger Bildsegmente existieren, werden auch weniger Threads gestartet) in denen Bereits die ersten vier Bildsegmente enthalten sind. Hier läuft die Verwaltung in den Synchronisationspunkt 1 (SP1) und wartet auf das Fertig werden eines Threads. Ist ein Threads mit dem Berechnen eines Bildsegementes fertig, läuft auch er in den SP1 und öffnet somit die Barriere. Der Algorithmus läuft nun weiter und prüft ob noch weitere Bildsegmente vorhanden sind, falls ja, wird nun der im Leerlauf befindliche Prozessor-Kern ermittelt. Auf diesem Kern soll der neue Thread gestartet werden und das nächste Bildsegment bearbeiten. Danach wartet der Algorithmus wieder auf das Fertig werden eine Threads.

Sind alle Bildsegmente vergeben, läuft der Algorithmus in den Synchronisationspunkt 2 (SP2). Dort warten alle Threads zusammen mit der Verwaltung, bis auch der letzte Thread das letzte Bildsegment fertig berechnet hat. Betritt auch der letzte Thread den SP2, werden die Threads gestoppt, terminiert und der Algorithmus fährt fort zur Nachbereitung.

Dritter Teil - Nachbereitung In der Nachbereitung wird zunächst der nicht mehr benötigte Speicher freigeben. Anschließend werden die erzeugten Feature-Listen aus den Threads zusammengeführt und wieder sortiert. Danach ist die Ausführung identisch zur Single Threaded Variante.

3.5 Testreihen der parallelen Variante

Im folgenden Abschnitt werden die durchgeführten Tests zusammengetragen und kurz beschrieben. Es wird kurz auf die Testbedingungen eingegangen, Resultate gezeigt und Schlussfolgerungen genannt. Dies soll, wie im Abschnitt zuvor, die Übersicht der Arbeit erleichtern. Im weiteren Verlauf dieses Abschnitts werden dann die einzelnen Experimente genau beschrieben und die Ergebnisse diskutiert.

3.5.1 Statische Testkonfiguration

Ähnlich wie bei den Testreihen der seriellen Ausführung des SIFT-Algorithmus, wird auch hier kurz die statische Testkonfiguration beschrieben. Auch gilt für alle Testreihen der parallelen Ausführung die Testkonfiguration, sofern sie nicht anders für die Testreihe beschrieben wurde. In Tabelle 6 sind diese statischen Testparameter zusammengefasst.

Die Felder Trials, SignURI und SceneURI entsprechen den Feldern, die bereits aus den seriellen Testreihen bekannten sind.

Option	Wert	Bedeutung
Trials	100	Anzahl der Testdurchläufe
SignURI	1	Codierung des Pfad zum Zeichenbild (1 ist der Std.-Wert und entspricht Abb. 13)
SceneURI	1	Codierung des Pfad zum Szenenbild (1 ist der Std.-Wert und entspricht Abb. 13)
Number of threads	4	Anzahl der Threads
Number of cores	4	Anzahl der verwendeten CPU-Kerne
Excecute SIFT till	detector_phase	Definiert, wie weit der SIFT-Algorithmus ausgeführt werden soll.

Tabelle 6: Statische Testkonfigurationen für alle Tests der MultiThreaded Variante. Sie gelten, wenn nicht anders erwähnt, für alle Testreihen der parallelen Ausführung des SIFT-Algorithmus

Number of threads Entspricht der Zahl der zu verwendenden Threads. Anhand dieses Parameters wird ebenfalls die Anzahl der Bildquadranten bzw. -streifen festgelegt.

Number of cores Bestimmt wie viele Kerne des Systems für die Testreihe verwendet werden sollen. Grundsätzlich kann dieser Wert immer bei 4 belassen werden. Selbst dann, wenn weniger Threads zur Verfügung stehen.

Excecute SIFT till HIer wird definiert, wie weit der SIFT-Algorithmus ausgeführt werden soll. Dieser Parameter kann drei Werte annehmen: Bis zur

- Bis zur Detektorphase
  Kann eingesetzt werden, um bis zur Detektorphase (exklusiv) zu messen.
- Bis zur Deskriptorphase
  Kann eingesetzt werden, um bis zur Deskriptorphase (exklusiv) zu messen.
- SIFT Komplett
  Kann eingesetzt werden, um eine Messung mit einem vollständigem Durchlauf des Algorithmus durchzuführen.

3.5.2 Zusammenfassung der Experimente

Zunächst wurden, wie in der Single Threaded Variante, initiale Geschwindigkeitsmessungen vorgenommen. Dazu wurde zunächst das Szenenbild in vier gleich große Quadranten zerlegt, auf die vier Kerne verteilt und die Ausführungszeit gemessen. Es zeigte sich entgegen der seriellen Implementierung, dass die drei Scheduling Verfahren den Algorithmus ohne signifikanten Unterschied beschleunigten. Aufgefallen sind jedoch unterschiedliche Ausführungszeiten und Anzahl der möglichen Merkmalspunkte der einzelnen Threads.

Zur Klärung, ob die möglichen Merkmalspunkte mit der Ausführungszeit in Verbindung stehen, wurde eine weitere Testreihe durchgeführt. Dieses mal wurde ein einzelner Quadrant verwendet und auf allen Kernen gleichzeitig berechnet.

Scheduling Mode	Ausführungszeiten	Differenz
SPM on TSM	$7,20$	$60ms$
SPM	$7,37s$	$230ms$
TSM	$7,14s$	$0ms$

Tabelle 7: Resultat der initialen Testreihe des parallel ausgeführtem SIFT-Algorithmus. TSM Bildet die Basis der Differenzbildung und hat daher die Differenz $0ms$

Bildbereich	Ausführungszeiten	Merkmalspunkte
C_1	$3,82s$	86
C_2	$4,95s$	132
C_3	$3,77s$	77
C_4	$7,44s$	272
$Generiert$	$14,45s$	758

Tabelle 8: Zusammenfassung der Ergebnisse aus der Testreihe zum Bestimmung des Verhältnisses zwischen Ausführungszeit und Merkmalspunkten.

Die Resultate der Berechnungen auf den vier Kernen war identisch. Es ergab sich daraus der erster Hinweis auf eine direkte Verbindung zwischen den gefunden Merkmalspunkten und der Ausführungszeit. Dieses Experiment wurde noch drei mal mit den verbleibenden Quadranten wiederholt und mit denen bisherigen Resultaten verglichen. Zusätzlich wurde ein Bild von Hand erzeugt und ebenfalls auf allen Kernen berechnet. Dieses Bild erzeugte mehr Merkmalspunkte als das Szenenbild und zeigte die erwartete erhöhte Ausführungszeit.

Eine problematische Erkenntnis ergab sich bei der Betrachtung der Resultate bzgl. der wiedererkannten Merkmalspunkte. Lag die Anzahl in der Single Threaded Variante noch bei 22 wiedererkannten Merkmalen, so fiel diese Zahl bei der 4 Thread Variante auf 21. Es wurde erkannt, dass bei der Aufteilung der Bildbereiche Informationen über das Bild verloren gingen. Die Konsequenz aus dieser Erkenntnis war, dass man für die folgenden Testreihen den Algorithmus nach der Detektorphase abbrach. Damit wurde die Vergleichbarkeit der Ergebnisse aus der Single Threaded Variante gewährleistet.

Eine weitere Erkenntnis aus den anfänglichen Testreihen war, dass Kerne im Leerlauf sind, während die letzten Kerne noch mit der Berechnung eines Quadranten beschäftigt war. Mit Hilfe einer weiteren Aufteilung des Bildes wurden nun Testreihen mit 8, 12 und 16 Threads durchgeführt. Nach [Gustafson(1988)] würde man die Leistung des Systems mit kleineren Bildbereichen effektiver ausnutzen können. Die Resultate aus den Testreihen bestätigen einen Trend zur Optimierung der Arbeitsweise. Mit dem Faktor $3,95$ wurde die Ausführung mit 16 Threads am stärksten beschleunigt.

3.5.3 Experimente im Detail

Mit den Resultaten aus den seriell durchgeführten Testreihen folgten die Testreihen mit der parallelen Verarbeitung des SIFT-Algorithmus. Zunächst wurden auch hier initiale Tests durchgeführt, um die drei Scheduling-Verfahren miteinander zu vergleichen.

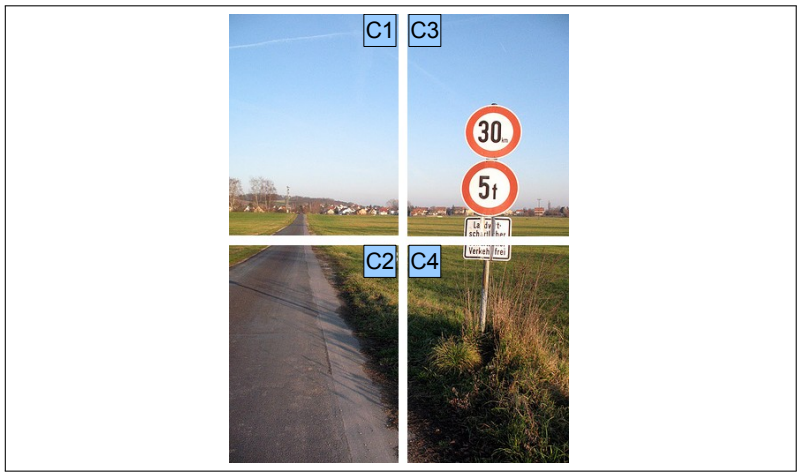

Abbildung 19: Aufteilung des Eingabebildes in die Quadranten C_1,...,C_4

Vorbereitungen Außer den Vorbereitungen, welche im Abschnitt 3.4 vorgestellt wurden, mussten für die erste Testreihe keine weiteren Vorbereitungen getroffen werden.

Testaufbau Bestandteil dieser Testreihen waren das Zeichen- und Szenenbild, welche bereits für die Single Threaded Variante verwendet wurden. Das Szenenbild wurde jedoch in vier gleich große Quadranten aufgeteilt (siehe Abb. 19). Je ein Quadrant ist auf einem anderem Kern berechnet worden.

Testdurchführung Gemessen werden wieder die Ausführungszeiten des SIFT-Algorithmus. Im Unterschied zur seriellen Ausführung jedoch, wurden hier auch die Zeiten der einzelnen Threads dokumentiert. Mit Hilfe dieser differenzierten Zeitmessung ließen sich in der Bewertung der Scheduling-Verfahren, die einzelnen Bildbereiche analysieren.

Resultate und Bewertung Zunächst werden die Ausführungszeiten der drei Scheduling Verfahren diskutiert. Hierbei wurden die einzelnen Zeiten der Threads zur Ausführungszeit D_{gesamt} zusammengefasst. D_{gesamt} beschreibt in diesem Zusammenhang die Differenz aus dem Startzeitpunkt des zu erst gestarteten Threads und dem Endzeitpunkt des zuletzt laufenden Threads. In Tabelle 9 sind die D_{gesamt} der einzelnen Scheduling Verfahren dargestellt. Aus der Abbildung wird ersichtlich, dass der Algorithmus im *TSM* Modus in kürzester durchlaufen wurde. Im Durchschnitt war dieser Mode 3% schneller als *SPM* und 1% schneller als *SPM on TSM*. Bezieht man die Standardabweichung σ (siehe Tab. 9) dieser Testreihe wieder mit

Scheduling Mode	D_{gesamt}	σ
SPM	$7,37s$	$0,1$
SPM on TSM	$7,2s$	$0,12$
TSM	$7,14s$	$0,08$

Tabelle 9: Messergebnisse aus der initialen Testreihe der parallelen Variante

in die Bewertung ein, so sieht man, dass 3% in der Ausführungsgeschwindigkeit keinen signifikanten Unterschied darstellt.

Nachdem D_{gesamt} betrachtet und bewertet wurde, folgte die Analyse auf der Threadebene. Die Ausführungszeiten D_x ($x = ThreadID$) der Threads wurden in Abbildung 20 und 21 veranschaulicht und zeigen, dass die einzelnen Bildbereiche unterschiedliche D_x erzeugten. Dies erklärt sich, indem man die Anzahl der berechneten möglichen Merkmalspunkte mit in die Betrachtung der Ergebnisse einbezieht. Eine unterschiedliche Anzahl der Merkmalspunkte ist nicht ungewöhnlich, da die Inhalte der Bildsegmente unterschiedlich sind. So ist beispielsweise im Bildsegment C1 viel Himmel zusehen. Dadurch sind hier im Verhältnis zu Bildsegment C4 weniger Objekte zu finden. Wie bereits im Kapitel 2.4.4 erklärt, sind Kontrastwerte mit entscheidend dafür, ob ein Bildpunkt in die Liste der Merkmalspunkte aufgenommen wird. Diese Aussage wird durch fünf weitere Testreihen untermauert, die auf Seite 53 näher beschrieben sind.

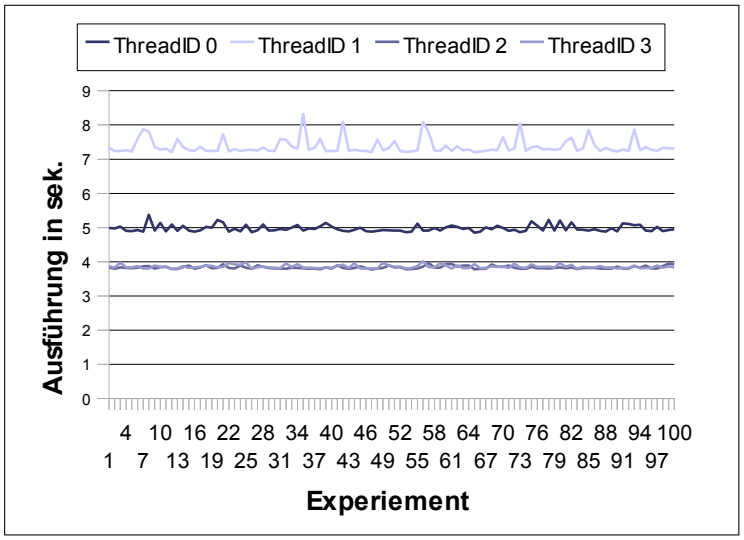

Abbildung 20: Ausführungszeiten der vier einzelnen Threads, ausgeführt im *SPM* Verfahren

Aus den Ergebnissen zeigen sich die problematische Charakteristiken der vereinfachten Implementierung des parallelen ausgeführten Algorithmus. Wurden bei den Testreihen im Single Thread 22 Merkmalspaare[14] im Szenen- und Zeichenbild

[14] Ein Merkmalspaar besteht aus einem Merkmal im Szenenbild und einem Merkmal im Zeichenbild

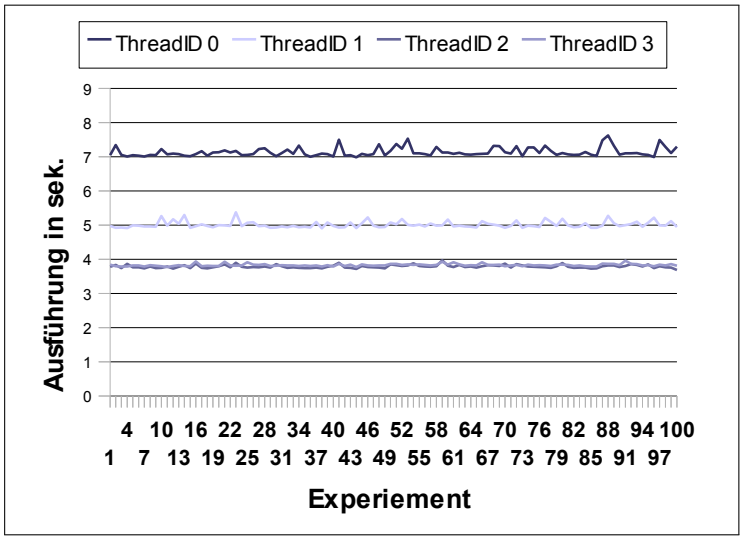

Abbildung 21: Ausführungszeiten der vier einzelnen Threads, ausgeführt im *TSM* Verfahren

Parameter	Wert
Korreliertes Rauschen	ausgewählt
Unabhängige RGB-Kanäle	ausgewählt
Rot	$0,20$
Grün	$0,20$
Blau	$0,20$

Tabelle 10: Parameter zur Erstellung des generierten Referenzbildes

ermittelt worden, konnten bei der Multi Thread Version nur noch 21. dieser Paare registriert werden. Durch die Aufteilung des Eingabebildes verändern sich auch die Bildpyramiden (Siehe 2.4.4). Hier kann es zu Randproblemen kommen, wenn ein Merkmalspunkt (im original Bild) zu nahe am Rand eines Quadranten liegt.

Wie bereits weiter oben beschrieben, folgen nun die Testreihen zum Klären der These, dass Merkmalspunkte direkt mit der Ausführungszeit in Verbindung stehen. Dazu werden die vier Quadranten einmal auf jedem Kern berechnet und die Ausführungszeiten verglichen. Zusätzlich wird ein von Hand erzeugtes Bild berechnet. Dieses Bild ist so beschaffen, dass mehr Merkmalspunkte errechnet werden und die gemessene Ausführungszeit damit länger werden sollte.

Vorbereitung Für einen der fünf Testreihen ist ein von Hand erzeugtes Bild vorgesehen. Daher wurde ein weiteres Eingabebild mit Hilfe des Bildbearbeitungsprogramms GIMP erzeugt (siehe Abb. 22). Es wurde zunächst ein weißes Bild mit den gleichen Außenmaßen eines Quadranten des Szenenbildes kreiert und anschließend mit dem Verzerrungsfilter *Rauschen* bearbeitet. Die Rausch-Parameter sind der Tabelle 10 zu entnehmen.

Testaufbau In vier der fünf Testreihen, wurde pro Testreihe genau ein Quadrant ausgewählt, dupliziert und auf die vier Kerne verteilt. Das erzeugte Eingabebild hatte bereits die Ausmaße eines Quadranten und wurde deshalb nur noch Dupliziert und auf die vier Kerne verteilt. Sollte die Anzahl der Merkmalspunkte weiter steigern, was auch auf die Ausführungszeit wirken sollte.

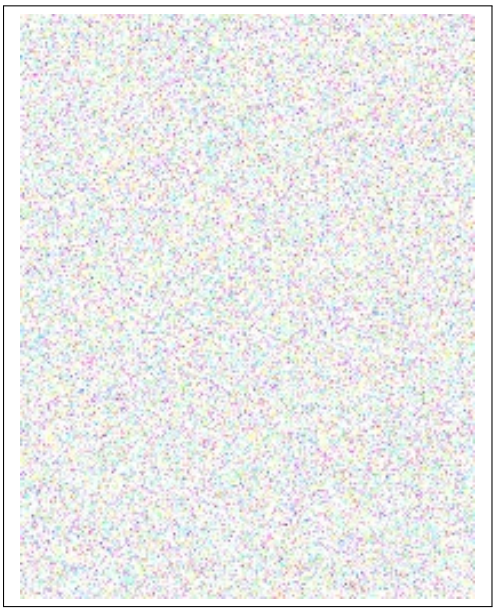

Abbildung 22: Das mit GIMP erzeugte Test-Eingabebild. Es diente dazu die Anzahl der möglichen Merkmalspunkte und somit die Ausführungszeit zu erhöhen

Resultate und Bewertung Die Resultate aus den vier Testreihen (Abb. 23) zeigen, dass alle vier Threads bei jeweils gleichem Bildsegment die gleiche Rechenzeit benötigt haben.Damit konnte gezeigt werden, dass die Ausführung nicht durch etwaige Stör-Tasks behindert wurden und das die Anzahl der Merkmalspunkte und die Rechenzeiten miteinander in Verbindung stehen.

Um jedoch diese Aussage allgemeingültig zu validieren, wurde ebenfalls eine Testreihe mit einem vollständig anderem Eingabebild (Abb. 22) durchgeführt. Das von Hand erzeugte Eingabebild zeigte einen Anstieg bei der Anzahl der möglichen Merkmalspunkte und eine ebenfalls verlängerte Ausführungszeit. Abbildung 24 stellt diesen Zusammenhang nochmals dar. Auf der linken Seite befindet sich die Kurve die die Ausführungszeiten der einzelnen Bildbereiche beschreibt. Auf der rechten Seite stellt die Kurve die möglichen Merkmalspunkte dar, die in dem jeweiligem Bildbereich berechnet wurden. Beide Kurven zeigen für einen gemeinsamen Bildbereich das gleich Verhalten.

Wie gerade gezeigt, ist der Grund für den Anstieg der Ausführungszeit durch die Anzahl der Merkmalspunkte bedingt und liegt an der Implementierung des SIFT-Algorithmus selbst. Da während der Detektorphase jedes potenzielle Merkmal

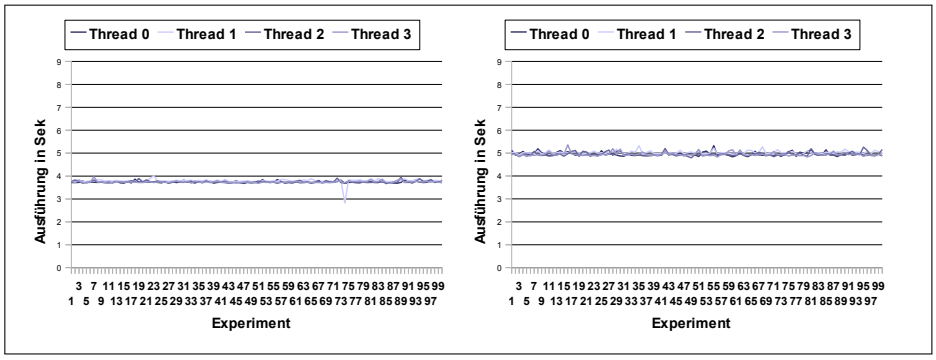

Abbildung 23: Ergebnisse aus dem Test zur Bestimmung der Verbindung zwischen Merkmalspunkten und der Ausführungszeit. Links sind die Ausführungszeiten des Bildbereiches C_1 zu sehen. Rechts die Ausführungszeiten aus C_2. Gut zu erkennen sind nun die gleichmäßigen Threadlaufzeiten $0 - 3$

gespeichert wird, kann es passieren, dass diese Merkmale im ganzen Bild und somit im Hauptspeicher verteilt sind. Wenn das passiert, muss während der Deskriptorphase oft der Speicher neu geladen werden, um die nötigen Daten zur Berechnung der Deskriptoren zu erhalten. Wie bereits im Kapitel 2.2.2 beschrieben sind diese Zugriffe auf den Hauptspeicher teuer und verursachen so eine längere Ausführungszeit. Hinzu kommen noch die zusätzlichen Speicherallokationen in der Detektorphase. Denn jeder neue Merkmalspunkt erzeugt eine weitere Allokation, bei der im Betriebssystem zunächst freier Speicher gesucht werden muss. Diese Suche kann im Idealfall mit der ersten Suchiteration vorbei sein, schlimmsten falls jedoch erst mit der letzten Iteration. Problematisch daran ist, dass diese Suche nicht deterministisch ist, und somit nicht vorhersagbar, wie lang eine Speicherallokation dauern wird.

Die These, dass Merkmalspunkte durch Sprünge durch den Speicherbereich die Ausführung des Algorithmus beeinträchtigen, soll in den folgenden Testreihen bewiesen werden. Dazu werden neben den Ausführungszeiten diesmal auch die Chachemisses und Cachehits aufgezeichnet.

Vorbereitung Da für diese Testreihe das Messen der Cachemisses und -hits voraussetzt, musste im Vorfeld eine Funktion implementiert werden, die diese Messung vor nimmt. Das Aufzeichnen der Cachemisses und -hits geschieht durch das Auslesen der Register CP15-c15 des ARM11 MPCore Dieses Register kontrolliert das *Count Register 0* (PNM0), das *Count Register 1* (PNM1) sowie das *Cycle Counter Register* (CCNT). Weitere Informationen zum c15 Register können im Abschnitt 3.2.1 oder der ARM-11 Refernz [ARM Ltd.(2009)ARM-11MPCore] entnommen werden.

Testaufbau Das Register c15 wurde vor Beginn der Testreihen so initialisiert, dass die Zählregister PMN0 und PMN1 jedes Auftreten eines Cachemiss oder -hits mitzählen. Der SIFT-Algorithmus wurde in dieser Testreihe vollständig (entgegen

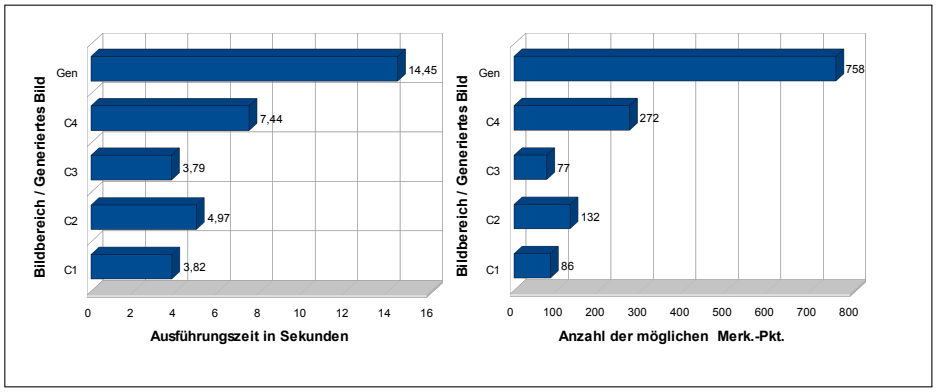

Abbildung 24: Gegenüberstellung des Verhältnisses aus Ausführungszeit und Anzahl der möglichen Merkmalspunkte. $C_1, \ldots, C_4$ sind die Quadranten des Szenen-Eingabebildes. Gen steht für das künstliche Eingabebild
Links: Ausführungszeit
Rechts: Anzahl der möglichen Merkmalspunkte

der statischen Testkonfiguration!) durchgeführt, da hier durch die Deskriptorphase das „Springen“ durch den Speicher besonders oft stattfinden sollte.

Testdurchführung Getestet wurden die Scheduling Verfahren *SPM*, *TSM* und *SPM on TSM*.

Resultate und Bewertung Die Testresultate der drei Testreihen müssen in zwei Gruppen zusammengefasst werden. Zum einen die Ergebnisse aus den Testreihen mit *SPM* und *SPM on TSM*. Zum anderen die Testreihe mit *TSM*. Die Unterscheidung der Ergebnisse begründet sich auf das Verhalten der drei Scheduling Verfahren. Da während der *SPM(on TSM)* gestützten Tests keine Migrationen durch geführt werden, sind hier die Ergebnisse genauer als jene, bei denen die Bildbereiche durch *TSM* nicht auf einen Kern festgelegt sind. Wie später in Abbildung 27 zu erkennen sein wird, weicht das Ergebnis für einen Bildbereich stark ab. Daher werden im folgenden zunächst die Ergebnisse aus den im *SPM* und *SPM on TSM* laufenden Testreihen präsentiert.

Der Übersicht halber wurden alle Ergebnisse von absoluten Werten mit

$$C_n(x) = \frac{c_x}{MAX(c_1, \ldots, c_4)} \tag{7}$$

mit C = Bildbereiche, $c \in C$ und $x = 1, \ldots, 4$ BildbereichID in eine auf 1 normierte Form gebracht. Die daraus entstandenen Kurven sind in Abbildung 25 (*SPM*) und Abbildung 26 (*SPM on TSM*) abgebildet. Wie aus den vorangegangen Testreihen bekannt, hat der Bildbereich C_2 die geringste Anzahl möglicher Merkmalspunkte. In den Grafiken lässt sich auch hier erkennen, dass hier sowohl die wenigsten Daten-Cache-Misses entstanden, als auch die wenigsten Zeit gebraucht wurde um den Bildbereich zu berechnen. Beide Kurven beschrieben eine ähnliche Gesamtkurve. Es konnte also gezeigt werden, dass die möglichen Merkmalspunkte im Speicher verteilt sind. Durch den Zugriff auf diese Punkte muss also häufiger

der Cache neu geladen werden, was dazu führt, dass das System auf diesem Kern langsamer wird. Ist die Anzahl der verwendeten Threads also ≤ 4 entstehen so Leerlaufphasen den Kernen, die die Berechnung ihres Bildbereiches abschließen konnten.

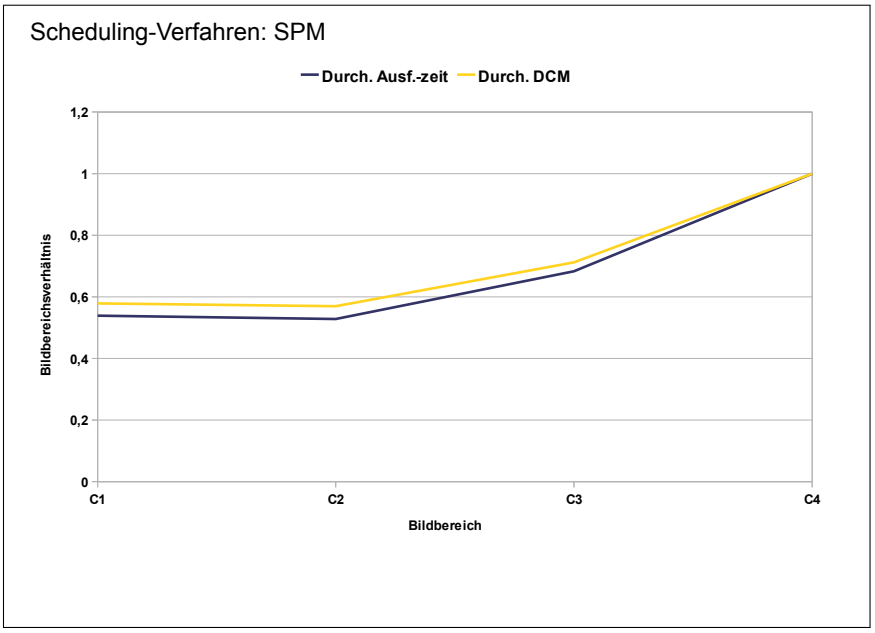

Abbildung 25: Testreihe im SPM Verfahren. Auf 1 Normierte Testresultate. Auch hier läst sich gut erkennen, das die Merkmalspunkte dafür sorgen, das Daten aus dem Hauptspeicher nachgelden werden müssen.

Die Testreihe im *TSM* Mode ist schwierig zu bewerten, da hier nicht vorhergesagt werden kann, in welchem Zustand das System zu einem bestimmten Zeitpunkt eines Testlaufs ist. So ist eine eindeutige Erklärung für den Kurvenverlauf in Abbildung 27 nicht möglich. Es ist nicht klar, ob im Bereich C_3 (und somit auf Kern 3) evt. noch weitere Bildbereiche berechnet wurden, oder ob die Kerne 3 und 4 möglicherweise ihre Bereiche durch den Scheduler ausgetauscht bekommen haben. Um diese Thesen validieren zu können, ist eine spezialisierte Hardware nötig. Sie müsste ermöglichen, sowohl Daten-Cache-Misses als auch Migrationen im selben Testlauf zu registrieren, ohne dabei Einfluss auf das System selbst auszuüben. Insbesondere das zählen der Migrationen ist ohne Einflussnahme auf das System nicht möglich, weshalb die Zählung der Migration immer von den Zeitmessungen entkoppelt waren und durch empirische Daten in Verbindung gebracht wurden.

Optimierung Die bisherigen Testresultate aus der Multi Threaded Variante zeigten, dass die Ausführungszeiten der einzelnen Bildbereiche bei allen Scheduling Verfahren unterschiedlich lang sind und damit sog. Idle-Zeiten[15] bei den Kernen gezeigt haben. Diese Idle-Zeit kann durch Verkleinerung der zu berechnenden Bildausschnitte (in diesem Fall schmalere Bildstreifen) und aktive Rechenzeit aufgefüllt werden [Gustafson(1988)]. Durch die damit verbesserte Parallelisierung erhält man

[15] Zeit in der eine CPU im Leerlauf ist

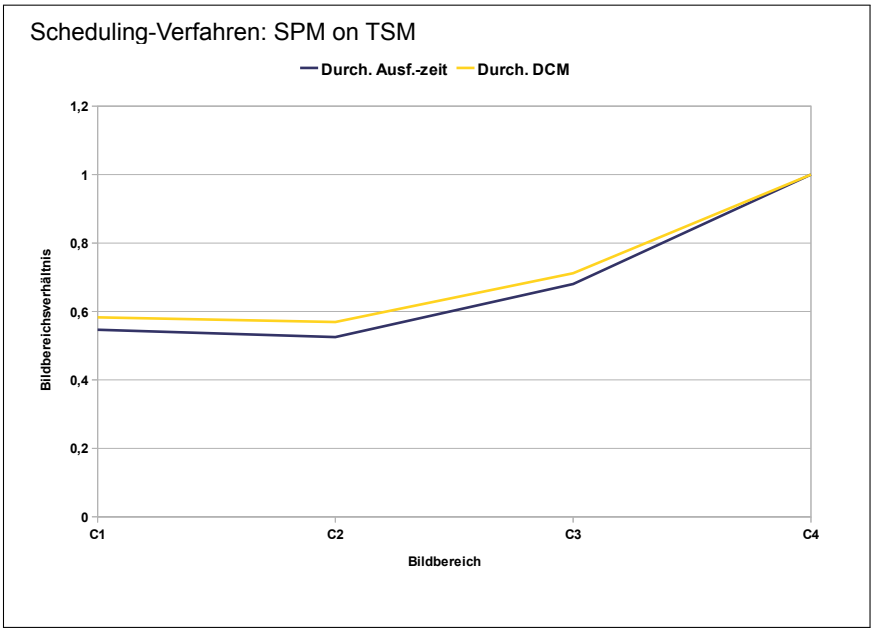

Abbildung 26: Testreihe im SPM on TSM Verfahren. Auf 1 Normierte Testresultate. Analog zur Ausführung im SPM Verfahren sind auch hier die steigenden Cachemisses mit mit den gefundenen Merkmalspunkten verbunden.

eine bessere Auslastung der Ressourcen und damit eine weitere Verkürzung der Ausführungszeit.

Vorbereitung Außer die im Abschnitt 3.4 beschriebenen Vorbereitungen mussten für diese Testreihe keine weiteren Vorbereitungen getroffen werden.

Testaufbau In den Folgenden Testreihen wurde untersucht, wie groß der Leistungszuwachs mit verschieden vielen Threads ist. Dazu wurden Testreihen mit unterschiedlicher Threadanzahl (1,4,8,12 und 16 Threads) erzeugt und durchgeführt.

Testdurchführung Getestet wurden die Scheduling Verfahren *SPM* und *TSM*. Die Aufteilung der Bilder wurde für diese Test dahingehend geändert, dass nun nicht mehr Quadranten aus dem Szenenbild erzeugt wurden, sondern horizontale Bildstreifen. Die Testreihen mit mehr als 4 Threads wurden noch dahingehend angepasst, so das die Threadzuweisung auf einen Kern automatisiert durchgeführt wurde (siehe Abschnitt 3.4). Zum Vergleich wurden auch die Ausführungszeiten eines Single Threaded Tests aufgezeichnet. Dieser Test wurde mit der gleichen Konfiguration durchgeführt, wie sie bei den Multi Threaded Tests eingesetzt wurde. D.h. die oben angesprochene Problematik mit den Merkmalspunkten wurde auch hier umgangen, indem der Algorithmus bis einschließlich der Detektorphase

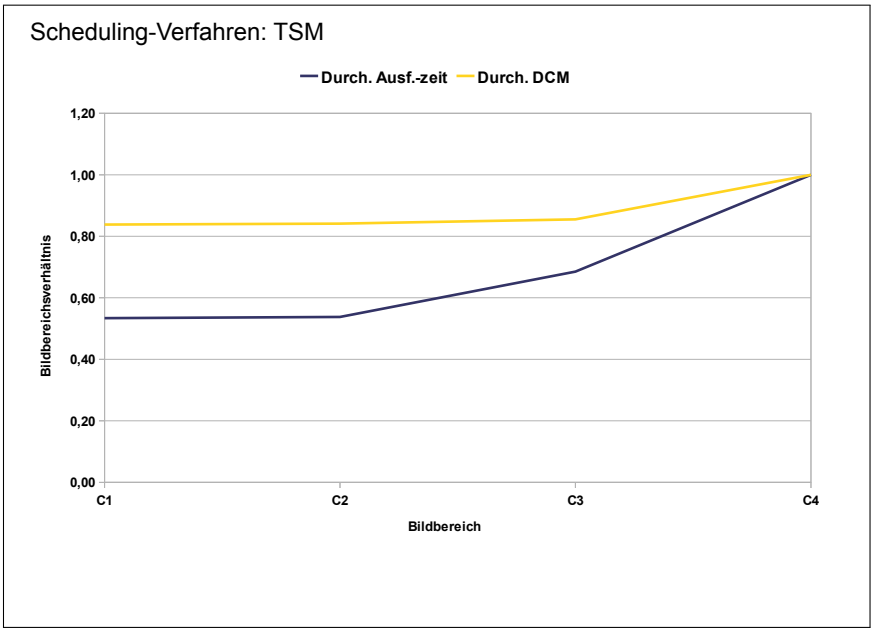

Abbildung 27: Testreihe im TSM Verfahren. Auf 1 Normierte Testresultate. Hier läst sich nur schwer sagen, was genau die Abweichung in den Segmenten C_3 und C_4 verursacht hat. Dennoch ist der Trend der steigenden Cachmisses im Bezug auf die Merkmalspunkte zu erkennen.

ausgeführt wurde.

Resultate und Bewertung In Abbildung 28 sieht man die Ausführungszeiten der Testreihen mit unterschiedlicher Anzahl der Threads. Die Grafik zeigt für beide getesteten Scheduling Verfahren deutlich eine Leistungssteigerung durch den Einsatz mehrerer Threads. Der Leistungszuwachs bei 16 Threads liegt bei etwa 380% im *SPM* Mode und 395% im *TSM* Mode. Dies entspricht dem, was durch [Gustafson(1988)] und [Hill und Marty(2008)] zu erwarten war. Dieses Ergebniss zeigt auch, dass bereits ein hohes Maß an Optmierung allein durch die Aufteilung des Szenenbildes erreicht wurde. Weitere Optimierungen sind zwar möglich, würden sich dann aber sowohl auf die serielle als auch auf die parallele Variante auswirken. Zudem sind Geschwindigkeitszuwächse jenseits des Faktors 4 bei einem MupltiCore Systems mit vier Kernen nicht möglich.

Ein weiterer Schluss kann an dieser Stelle bereits getroffen werden. Der Scheduling Mode *TSM* ist in allen Testreihen als der hervorgegangen, der den SIFT-Algorithmus in jeweils der kürzesten Zeit beschleunigt hat.

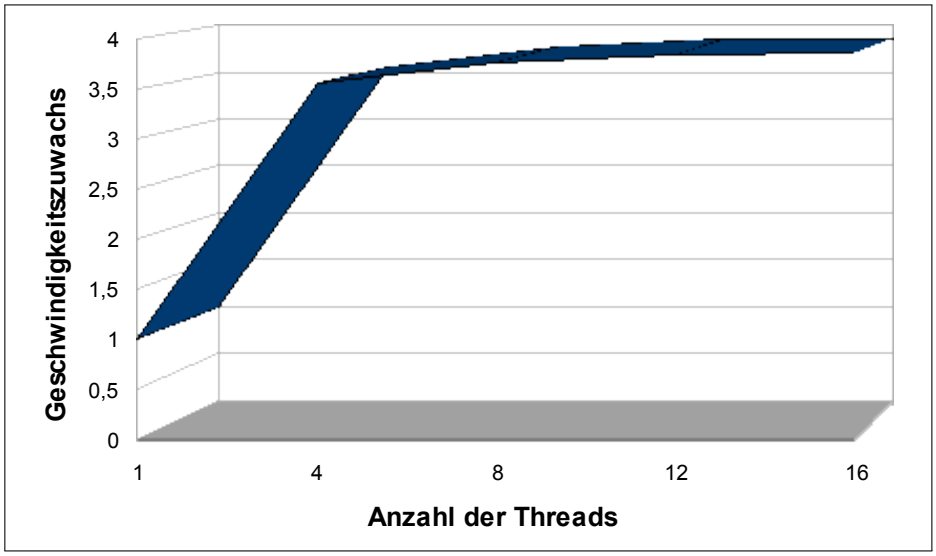

Abbildung 28: Zeigt den Leistungszuwachs des Systems durch die Nutzung von mehreren Threads. Der Versuch mit 16 Thread zeigt ein Leistungszuwachs vom $\approx 3,8$ für den *SPM* und $\approx 3,9$ für den *TSM* Mode.

4 Zusammenfassung

Für diese Arbeit wurden Scheduling-Verfahren des Echtzeit-Betriebssystem eT-Kernel und dem Mehrkernsystems ARM11 MPCore untersucht, deren Aufgabe darin besteht, Tasks oder Teile dieser Tasks auf alle CPU-Kerne verteilt ausführen zu lassen. Hierfür wurde eine bereits bestehende Implementierung des SIFT-Algorithmus als Last-Erzeuger verwendet und für die Testreihen der verteilten Ausführung angepasst, beziehungsweise um die Multithreading-Fähigkeit erweitert. Zunächst wurde theoretische Konzeptionierung zur Implementierung eines parallel ausführbaren SIFT-Algorithmus gegeben. Anschließend wurde aus zeitlichen Gründen eine vereinfachte Variante dieses Konzeptes Implementiert.

Die für diese Analyse maßgeblichen Kennzahlen wurden aus den Ausführungszeiten, der Anzahl an Task-Migrationen des SIFT-Algorithmus, sowie dem Cachingverhalten der einzelnen CPU-Kerne ermittelt. Zusätzlich wurden verschiedene Stör-Tasks implementiert, die den Ablauf des Algorithmus unterbrachen oder migrieren ließen.

Das Ergebnis dieser Arbeit ist eine Aussage über die Effizienz der drei Scheduling-Verfahren. Zunächst wurden dazu die Scheduling-Verfahren untereinander anhand der Ausführungszeiten des SIFT-Algorithmus in serieller Ausführung verglichen. Die dabei gewonnen Erkenntnisse führten zu der Untersuchung des Migrationsverhaltens des SIFT-Algorithmus. Gegenstand dieser Untersuchung waren speziell ausgewählte Scheduling-Verfahren, welche in den vorhergehenden Testreihen unterschiedliche Ergebnisse bezüglich der Ausführungszeiten lieferten. Aus den Resultaten dieser Testreihen ergab sich die Frage nach den Kosten der Migrationen. Eine weitere Testreihe, in der die gemessene Ausführungszeit und die Anzahl an Migrationen empirisch ermittelt wurden, wurde zur Bewertung der Task-Migration verwendet.

Im Zweiten Teil der Untersuchung der Scheduling Verfahren wurden Testreihen mit dem parallel ausgeführten SIFT-Algorithmus durchgeführt. Zunächst wurden auch hier die Scheduling Verfahren miteinander anhand der Ausführungszeiten des SIFT-Algorithmus verglichen. Die Resultate dieser Testreihen warfen Fragen bezüglich Merkmalspunkte und Ausführungszeit auf. Um diese Fragen zu klären, wurden die Merkmalspunkte hinsichtlich ihrer Auswirkung auf die Ausführungszeit des SIFT-Algorithmus hin untersucht. Dazu wurden in weiteren Testreihen auch das Caching-Verhalten in die Untersuchung mit einbezogen.

4.1 Fazit

Wie aus den Resultaten der Untersuchung des seriell ausgeführtem SIFT-Algorithmus aus Abschnitt 3.3.2 zur erkennen ist, haben Migrationen mit MESI-Erweiterung DDI (Abschnitt 2.2.2) einen nicht erwarteten Einfluss auf die Ausführungszeiten. Das Scheduling-Verfahren *TSM* des eT-Kernels (Abschnitt 2.4.2) ließ den SIFT-Algorithmus im Durchschnitt in $18,62s$ vollständig durchlaufen. Damit war dieses Scheduling-Verfahren $340ms$ bzw. $450ms$ schneller als die beiden weiteren getesteten Scheduling-Verfahren *SPM on TSM* und *SPM*.
Eine mögliche Erklärung für diese Resultate war, dass das Migrieren auf einen anderen (im Leerlauf befindlichem) Kern verkürzend auf die Ausführungszeit des

SIFT-Algorithmus wirkt. Eine weitere Testreihe sollte diese These untersuchen. Das Ergebnis war mit durchschnittlich 524 Migrationen eindeutig. Die Frage nach den Migrationen war damit beantwortet, jedoch stellte sich nun die Frage der tatsächlichen Migrationskosten. Bis dahin wurden Migrationskosten als mit entscheidende Menge der Ausführungskosten betrachtet.
Hieraus entstand die Frage nach den tatsächlichen Kosten durch Task-Migrationen und damit eine weitere Testreihe zur Beantwortung dieser Frage. Die nachfolgende Untersuchung der Kosten ergab, hoch gerechnet auf auf die Ausführungszeit von $12,24s$, im Durchschnitt 451 Migrationen ohne aktivem Stör-Task. Im Vergleich dazu, stieg die Ausführungszeit mit aktivem Stör-Task des SIFT-Algorithmus auf $12,39s$ und 1242 Migrationen. Für die Ausführungszeit ergibt sich daraus ein Zuwachs um den Faktor $1,012$, während die Anzahl der Migrationen mit dem Faktor $2,722$ zunahm. Bereits jetzt wurde klar, dass die Migrationen keinen signifikanten Einfluss auf die Ausführungszeit eines zeitaufwendigen Algorithmus haben.
Offen blieb jedoch noch die tatsächlichen Kosten der Migrationen. Durch Messungen konnten diese Kosten nicht bestimmt werden, da die nötige Hardware in dieser Arbeit nicht zur Verfügung stand. Jedoch konnte mit den bereits ermittelten empirischen Daten aus Ausführungszeit und Anzahl der Migrationen ein Richtwert von $130ns$ (Gleichung 6 Seite 45) berechnet werden. Hieraus folgte also, dass Migrationen für Tasks mit exterm kurzer Ausführungszeit (im Millisekundenbereich), wie z.B. Treiber oder Interrupt-Behandlung kein adiquates Mittel ist, um die Ausführung solcher Tasks zu beschleunigen.

Mit der Implementierung der parallelen Verarbeitung des SIFT-Algorithmus konnten nun die Scheduling-Verfahren dahingehend Untersucht werden, wie sie den Algorithmus parallel ausführen ließen. Auch hier erwies sich das Scheduling-Verfahren *TSM* des eT-Kernel mit einer Ausführungszeit von $7,14s$ am effektivsten. Die beiden verbliebenen Scheduling Verfahren beschleunigten den Algorithmus auf $7,20$ (Ausführung im *SPM on TSM* Verfahren) und $7,37s$ (Ausführung im *SPM* Verfahren). Auch hier ließen sich $\approx 0,8\%$ bis $\approx 3,2\%$ Leistungsunterschied feststellen.
Bei der Untersuchung der drei Scheduling Verfahren viel jedoch ein weiterer Umstand auf. Die Threads des SIFT-Algorithmus hatte bei allen Scheduling Verfahren eine unterschiedliche Laufzeit.
Analysen des Cachingverhaltens haben gezeigt, dass die Last der Bildsegmentberechnungen nicht gleichermaßen auf alle Kerne verteilt war. Dies führte zu einem Leerlauf der CPU-Kerne, deren Last geringer war und somit schneller das ihnen zugewiesene Bildsegment berechnen konnten.
Daraufhin wurde die Implementierung des Threadsmanagments dahingehend erweitert, mehr als vier Threads dynamisch verwalten zu können. Mit dieser Funktionalität wurden im Anschluss die Testreihe mit 1,8, 12 und 16 Threads wiederholt.
Bei 16 Threads und somit 16 Bildbereichen erreichte das Scheduling Verfahren *TSM* ein SpeedUp von $3,95$, wobei das theoretische Maximum bei $4,0$ liegt. Das Scheduling Verfahren *SPM* erreichte ein SpeedUp von $3,80$.

Die Untersuchung der drei Scheduling Verfahren hat ergeben, das alle drei Scheduling Verfahren die nahe zu gleichermaßen in der Lage sind den SIFT-Algorithmus zu beschleunigen. Hierbei zeigte sich, dass das Scheduling Verfahren *TSM* im Durch-

schnitt 3% schneller war, als die beiden verbleibenden Verfahren. Es wurde gezeigt, dass das *TSM* Verfahren durch die Migrationskosten von $\approx 130ns$ des bei Zeitaufwendigen Anwendungen von keiner signifikanten Größe sind. Dadurch können Migrationen die Ausführung solcher Algorithmen von Vorteil sein, falls die Berechnung auf einem CPU-Kern mit weiteren Tasks ausgeführt wird.
Durch die Aufteilung des Eingabebildes und die Verteilung dieser Bildsegmente auf Threads konnte ein Leistungszuwachs vom Faktor $3,95$ erreicht werden. Bei vier CPU-Kernen liegt der theoretische Zuwachs bei 4. Es konnte also gezeigt werden, dass bereits das Aufteilen und verteilte parallele Berechnen des Eingabebildes den Leistungszuwachs nahe an das theoretische Maximum beschleunigte. Problematisch jedoch, ist bislang die Umsetzung des SIFT-Algorithmus für den Mehrkernbetrieb. Durch die Aufteilung des Bildes zeigte sich, das weniger bis keine Merkmalspunkte der Bilder einander zugeordnet werden konnten. Wie dieses Problem zu beheben sein könnte, wird im folgenden Abschnitt, dem Ausblick, diskutiert.

4.2 Ausblick

Die Messung der Anzahl der Migrationen wirkt nachhaltig auf die Ausführungszeit des Algorithmus. Daher mussten diese Messungen stets getrennt voneinander durchgeführt werden. Daraus resultierte, dass nur durch empirische Daten eine Verbindung zwischen den Migrationen und den Ausführungszeiten hergestellt werden konnte. Gleiches gilt für die Berechnung der Dauer einer Migration.
Um jedoch noch genauere Ergebnisse zu erhalten, währe eine Untersuchung der Migrationen und Ausführungszeiten durch eine geeignete Hardware erforderlich. Diese Hardware müsste dann, ohne Einflussnahme auf das System, die Ereignisse dokumentieren können. Denkbar währen hier Mechanismen die ähnlich wie die Snoop Control Unit auf der designierten Datenleitung lauscht und alle Ereignisse erfasst.

Wie bereits in der Arbeit angesprochen, geht der Zerlegung des Szenenbildes Informationen über den Bildinhalt an den Schnitträndern verloren. Eine Idee ist, dass Bild zwar weiterhin zu zerlegen und die einzelnen Pyramiden erzeugen zulassen, danach aber die Pyramiden einzelnen Pyramiden wieder zu einer großen zu vereinen.

Hiernach erzeugt man neue Threads, welche mit Start- und Endkordinaten initiiert werden. Dies Koordinaten dienen als Bildsegment, in dem der Thread die Merkmalspunkte sucht. Gefundene Merkmalspunkte werden in eine globale Liste eingetragen, damit weitere Threads beginnen können, die Merkmalspunkte mit denen aus dem Referenzbild zu vergleichen. Merkmalspaare werden dann wieder in eine gemeinsame Liste zusammengefasst. Dieses Konzept wird das Problem mit den fehlenden Merkmalspaaren lösen können.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[Lowe99(1999)] Lowe99 1999 : *Object recognition from local scale-invariant features*. Bd. *2*. URL http://dx.doi.org/10.1109/ICCV.1999.790410, 1999. – 1150–1157 vol.2 S

[Intel(2009)] Intel 2009 *Product Brief: Dual-Core Intel Xeon Processor 500 Series*, 2009

[ARM11(2009)] ARM11 2009 : *Products and Solutions*. Website. 2009. – URL http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html

[Agarwal u. a.(1988)Agarwal, Simoni, Hennessy und Horowitz] Agarwal u. a. 1988 Agarwal, Anant ; Simoni, Richard ; Hennessy, John ; Horowitz, Mark: An Evaluation of Directory Schemes for Cache Coherence. In: *In Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, S. 280–289

[Allan u. a.(2002)Allan, Edenfeld, Joyner, Kahng, Rodgers und Zorian] Allan u. a. 2002 Allan, Alan ; Edenfeld, Don ; Joyner, William H. ; Kahng, Andrew B. ; Rodgers, Mike ; Zorian, Yervant: 2001 Technology Roadmap for Semiconductors. In: *Computer* 35 (2002), Nr. 1

[ARM Ltd.(2009)ARM-11MPCore] ARM Ltd. 2009 ARM Ltd. (Veranst.): *ARM11 MPCore Processor Revision: r1p0*. 2009

[Benz(2008)] Benz 2008 Benz, Benjamin: Speicherfix: Aufbau und Synchronisation von Caches in Mehrkernprozessoren. In: *CT* 2008 (2008), Nr. 13, S. 220–225

[zu Bexten und Hiltner(1998)] zu Bexten und Hiltner 1998 Bexten, Erdmuthe M. zu ; Hiltner, Jens: Medizinische Bildverarbeitung: Aktueller Stand und Zukunftsperspektiven. In: *Bildverarbeitung fi¿ $\frac{1}{2}$ r die Medizin*, 1998

[eSoL(2009)] eSoL 2009 eSoL, Ltd: *Car navigation systems*. 2009. – URL http://www.esol.co.jp/english/embedded/successstory.html

[Flynn(1972)] Flynn 1972 Flynn, M.: Some Computer Organizations and Their Effectiveness. In: *IEEE Trans. Comput.* C-21 (1972), S. 948+

[Gondo(2006)] Gondo 2006 Gondo, Masaki: Blending Asymmetric and Symmetric Multiprocessing with a Single OS on ARM11 MPCore. In: *Inform. Quar. 5* 4 (2006)

[Gustafson(1988)] Gustafson 1988 Gustafson, John L.: Reevaluating Amdahl's law. In: *Commun. ACM* 31 (1988), Nr. 5, S. 532–533. – URL http://dx.doi.org/10.1145/42411.42415. – ISSN 0001-0782

[Hill und Marty(2008)] Hill und Marty 2008 Hill, M. D. ; Marty, M. R.: Amdahl's Law in the Multicore Era. In: *Computer* 41 (2008), Nr. 7, S. 33–38. – URL http://dx.doi.org/10.1109/MC.2008.209

[Keltcher u. a.(2003)Keltcher, McGrath, Ahmed und Conway] Keltcher u. a. 2003 KELTCHER, Chetana N. ; MCGRATH, Kevin J. ; AHMED, Ardsher ; CONWAY, Pat: The AMD Opteron Processor for Multiprocessor Servers. In: *IEEE Micro* 23 (2003), Nr. 2, S. 66–76. – ISSN 0272-1732

[Lowe(2004)] Lowe 2004 LOWE, David G.: Distinctive Image Features from Scale-Invariant Keypoints. In: *Int. J. Comput. Vision* 60 (2004), Nr. 2, S. 91–110. – ISSN 0920-5691

[Molnos u. a.(2006)Molnos, Heijligers, Cotofana und van Eijndhoven] Molnos u. a. 2006 MOLNOS, A. M. ; HEIJLIGERS, M. J. M. ; COTOFANA, S. D. ; EIJNDHOVEN, J. T. J. van: Compositional, efficient caches for a chip multi-processor. In: *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium : European Design and Automation Association, 2006, S. 345–350. – ISBN 3-9810801-0-6

[QNX(2009)] QNX 2009 QNX: *QNX Neutrino RTOS*. Website. 2009. – Online gefunden bei http://www.qnx.com/; besucht am 26. April 2009.

[Stensland u. a.(2008)Stensland, Griwodz und Halvorsen] Stensland u. a. 2008 STENSLAND, Håkon K. ; GRIWODZ, Carsten ; HALVORSEN, Pål: Evaluation of multi-core scheduling mechanisms for heterogeneous processing architectures. In: *NOSSDAV '08: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. New York, NY, USA : ACM, 2008, S. 33–38. – ISBN 978-1-60558-157-6