Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Lars Kleen

Parallelisiertes Echtzeit-Audio-Processing auf Mehrkernsystemen mit Hyper-Threading-Technologie

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Lars Kleen

Parallelisiertes Echtzeit-Audio-Processing auf Mehrkernsystemen mit Hyper-Threading-Technologie

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Wolfgang Fohl
Zweitgutachter : Prof. Dr. Friedrich Esser

Abgegeben am 17. Juni 2009

Danksagung

An meine Eltern, die mich dazu ermutigt haben zu studieren.

Professor Dr. Fohl, für die gute Betreuung während meines Praxissemesters und der Anfertigung dieser Arbeit.

Frank Sauerland, der mich im Rahmen meines Praxissemesters bei Steinberg hervorragend betreut hat.

An die Firma Steinberg und die Mitarbeiter der Entwicklungsabteilung, die es mir ermöglicht haben diese Arbeit fertig zu stellen.

Lars Kleen

Thema der Bachelorarbeit

Parallelisiertes Echtzeit-Audio-Processing auf Mehrkernsystemen mit Hyper-Threading-Technologie

Stichworte

Hyper-Threading, Mehrkernsysteme, Echtzeit, Audio-Processing, Core i7, Nehalem, NetBurst, Multithreading, Nuendo 5, Steinberg, Intel

Kurzzusammenfassung

Diese Arbeit befasst sich mit den Auswirkungen von Hyper-Threading auf das Audio-Processing der Nuendo 5 Audio-Engine. Zur Anlayse an speziell entwickelten Prototypen und der Audio-Engine werden unterschiedliche Profiling-Werkzeuge verwendet, um abschließend eine fundierte Aussage über die Eignung von Hyper-Threading für das Echtzeit-Audio-Prozessing auf Microsoft-Betriebssystemen treffen zu können.

Lars Kleen

Title of the paper

Parallelized Realtime-Audio-Processing on Multicore-Platforms with Hyper-Threading-Technology

Keywords

Hyper-Threading, Multicore, realtime, Audio-Processing, Core i7, Nehalem, NetBurst, Multithreading, Nuendo 5, Steinberg, Intel

Abstract

This paper covers the effects of Hyper-Threading on the realtime-audio-processing of the Nuendo 5 audio-engine. Different profiling-tools are utilized for the analysis of specifically developed prototypes and the audio-engine, to close the paper with a knowledgeable statement about the applicability of Hyper-Threading for realtime-audio-processing on Microsoft operating-systems.

Inhaltsverzeichnis

1 Einleitung

1.1 Vorwort

Die Weiterentwicklung von Mikroprozessoren war lange Zeit davon gekennzeichnet, dass stetig neue Rekorde bei der Erhöhung der Taktrate aufgestellt wurden. Seit Ende 2003 geht diese Entwicklung nun in eine andere Richtung[11]. Während die höchsten Taktraten bei etwa 3 GHz stagnieren, wird die Zahl der verfügbaren Prozessorkerne auf einem Chip immer weiter aufgestockt. Aktuelle CPUs erlauben die parallele Ausführung von bis zu 8 *Thread*s. Anwendungen, die ihre Rechenlast geschickt auf mehrere CPUs verteilen, können dadurch erheblich an Performance gewinnen.

Der Begriff Hyperthreading wurde erstmals Anfang 2002 mit der Einführung des 2.20 GHz *Xeon* Prozessors verwendet und bezeichnet die simultane Ausführung mehrerer Threads auf einem physikalischen CPU-Kern, häufig wird diese Technik auch als Simultaneous MultiThreading (SMT) bezeichnet.

Während bei multicore-Architekturen ohne Hyperthreading alle Ressourcen doppelt ausgelegt sind, sind bei einem CPU-Kern, der HT beherrscht, nur die Register für den Daten- und Informationsfluss doppelt ausgelegt. Dem Betriebssystem gegenüber werden für jeden physikalischen CPU-Kern zwei logische CPU-Kerne dargestellt. Die *Execution Units* für *Integer* und *Floating-Point*-Operationen werden von beiden logischen Kernen gemeinsam genutzt.

Gerade bei der Entwicklung von *Echtzeit*anwendungen, die auf mehreren Prozessorkernen parallel arbeiten muss den Besonderheiten, die bei der Verwendung gemeinsam genutzter Ressourcen eines CPU-Kernes auftreten, erhöhte Aufmerksamkeit geschenkt werden.

Die Auswirkungen und Ursachen dieses Verhaltens sollen in dieser Arbeit anhand des aktuellen Entwicklungsstandes von Nuendo 5 im Zusammenspiel mit einem Core i7 Extreme Prozessor genauer betrachtet werden.

1.2 Zielsetzung

Die Zielsetztung dieser Arbeit besteht darin, eine fundierte Aussage darüber treffen zu können, welche Auswirkungen die Hyper-Threading-Technologie aktueller und kommender Prozessorgenerationen auf das Echtzeit-Audio-Processing auf Microsoft-Betriebssystemen hat.

Aufbauend auf dieser Fragestellung soll untersucht werden, ob durch Modifizierungen am derzeitigen Entwicklungsstand der Audio-Engine von Nuendo 5 eine spürbare Leistungsverbesserung durch Hyper-Threading erreicht werden kann.

1.3 Aufbau dieser Arbeit

Im Laufe dieser Arbeit werden nach und nach eine Reihe von Begriffen eingeführt, die insbesondere für nicht-Informatiker, nicht immer selbsterklärend sind. Entsprechende Begriffe werden bei der ersten Verwendung kursiv dargestellt, was als Hinweis darauf anzusehen ist, dass zu dem jeweiligen Begriff eine kurze Erklärung im Glossar zu finden ist.

Der Aufbau dieser Arbeit ist so konzipiert, dass man beim Lesen vom ersten bis zum letzten Kapitel alle durchgeführten Anlaysen und Arbeitsschritte nachvollziehen kann. Begonnen wird in Kapitel 2 mit einem einleitenden Teil, der sich mit den Grundlagen des Audio-Processings befasst. In Kapitel 3 und 4 wird auf die technischen Hintergründe eingegangen, die im Zusammenhang mit den anschließenden Kapiteln von Bedeutung sind.

Kapitel 5 befasst sich mit den Werkzeugen, die dazu verwendet wurden, Performance-Messungen an den Prototypen und der Nuendo-Implementierung durchzuführen, und gibt einen Überblick über die in den folgenden Kapiteln ermittelten Messwerte und die verwendeten Messverfahren. In Kapitel 6 beginnt die Analyse der Hyper-Threading-Technologie. Zunächst gibt ein Vergleich der Hyper-Threading-Implementierung auf dem ersten Xeon-Prozessor aus dem Jahr 2002, mit der Hyper-Threading-Implementierung auf einem aktuellen Core i7-Prozessor, einen Einblick in die technische Entwicklung seit der Einführung dieser Technologie.

Die Ergebnisse und Versuchsprojekte für die bestehende Nuendo 5 Implementierung werden im ersten Teil des 7. Kapitels dargestellt. Der zweite Teil des Kapitels beschäftigt sich mit der Auswertung von Messungen an unterschiedlichen Prototypen die speziell für die Analyse von Hyper-Threading konzipiert wurden.

Im 8. Kapitel wird schließlich auf Möglichkeiten zur Anpassung der Audio-Engine eingegangen, die daraufhin ausgelegt sind, den speziellen Begebenheiten der Hyper-Threading-Technologie gerecht zu werden.

2 Audio-Processing und Audio-Scheduling

2.1 Abtastrate

Eine hohe Abtastrate ist notwendig, um das analoge Audiosignal möglichst originalgetreu wiedergeben zu können.

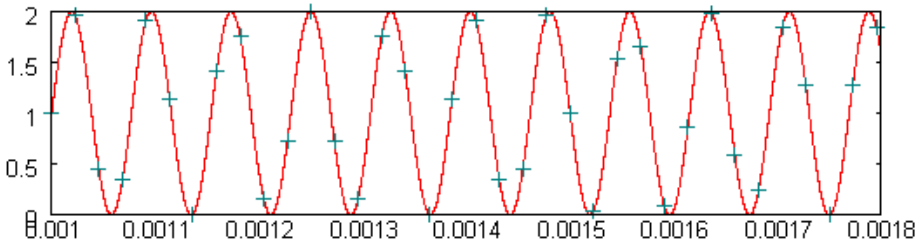

Abb. 1: Abgetastete Punkte eines 13kHz Sinus bei 44 kHz Abtastrate

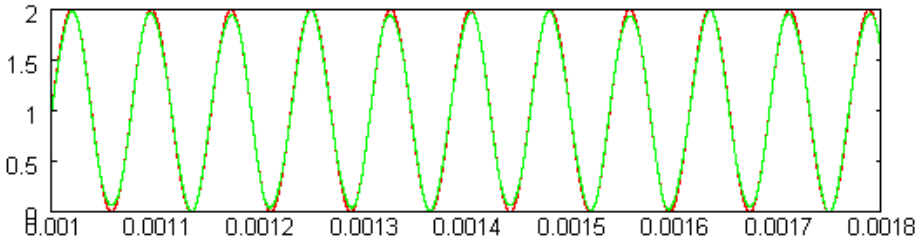

Abb. 2: Interpoliertes Signal bei 44 kHz Abtastrate

Im obigen Beispiel ist zu erkennen, wie sich aus einem bei 44 kHz abgetasteten Sinussignal mit einer Frequenz von 13kHz wieder ein nahezu identisches Signal interpolieren lässt. Höhere Frequenzen bei der Abtastrate lassen hier eine entsprechend genauere Interpolation zu.

Bei dem Beispiel in Abbildung 3 und 4 ist zu sehen, dass das Ergebnis einer Interpolation aus einer Abtastung mit nur 22kHz das Signal stark verfälscht wiedergibt. Obwohl auf den ersten Blick die Vermutung naheliegt, dass man mit einer Abtastrate von 22kHz auch Signale bis zu 22kHz wiedergeben kann, tritt hier ein Problem auf, dass im Zusammenhang mit digitaler Singnalverarbeitung als Aliasing bezeichnet wird.

Bei der Wandlung von digitalen in analoge Signale macht sich dieser Effekt bemerkbar, sobald die Frequenz des wiederzugebenden Signals oberhalb der Hälfte der Abtastfrequenz liegt. Die wiedergegebene Frequenz wird dann nicht mehr höher, sondern beginnt wieder tiefer zu werden.

Um ein Signal nach der Wandlung verlustfrei wiedergeben zu können, muss die Abtastfrequenz demnach mindestens doppelt so hoch sein wie die Frequenz des wiederzugebenden Signals.

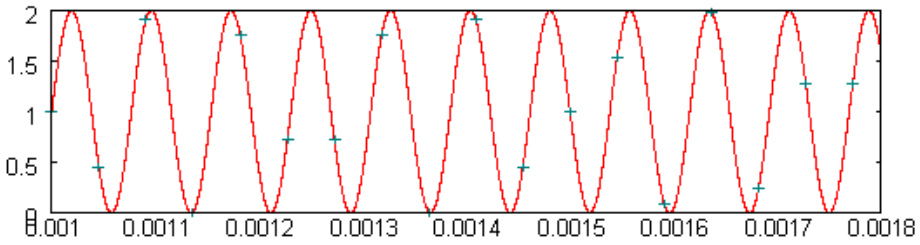

Abb. 3: Abgetastete Punkte eines 13kHz Sinus bei 22 kHz Abtastrate

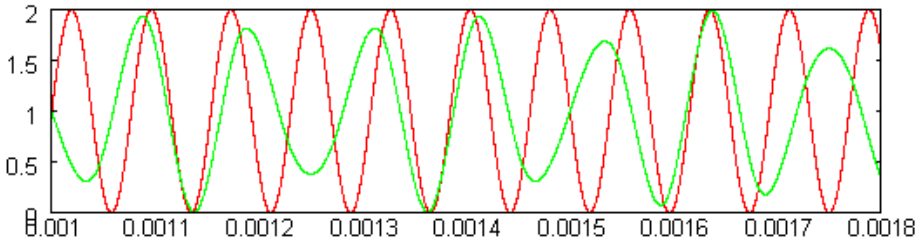

Abb. 4: Interpoliertes Signal bei 22 kHz Abtastrate

Die Frequenz des abzutastenden Signals wird im Bezug zur Abtastfrequenz auch als Nyquist-Frequenz bezeichnet und entspricht genau der Hälfte der Abtastfrequenz.

2.2 Latenz

Mit der Latenz wird die Verzögerung beschrieben, die vom Eingang eines Signals, wie beispielsweise eines Audio-Eingangssignals oder eines MIDI-Events, bis zu dessen Ausgabe am Audio-Ausgang entsteht.

Die Latenz, die der Spieler eines akustischen Instrumentes wahrnimmt, lässt sich mit folgender Formel einfach errechnen:

$$\frac{Schallgeschwindigkeit}{Entfernung} = Latenz$$

Ein Musiker, dessen Ohren sich in 40 cm Abstand von der Schallquelle befinden, hört sein Instrument also bereits mit einer Latenz von:

$$\frac{340\,\mathrm{m}/1\,\mathrm{s}}{0{,}4\,\mathrm{m}} = 1{,}18 \cdot 10^{-3}\,\mathrm{s}$$

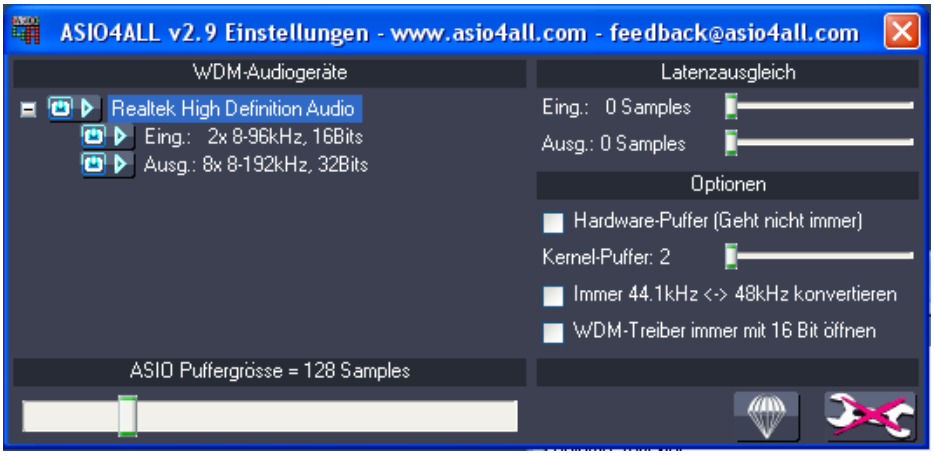

Abb. 5: Einstellungen des ASIO-Treibers

Die Zeit, die die Berechnung eines *ASIO-Block*s maximal in Anspruch nehmen darf, errechnet sich aus der Abtastrate und der Größe des *Sample*-Buffers.

$$\frac{AnzahlSamples}{Abtastrate} = maximale\ Processingzeit$$

Die in Abbildung 5 eingestellten Werte ergeben demnach ein Zeitfenster von

$$\frac{128\,\text{samples}}{44\,100\,\text{samples/s}} = 2{,}90 \cdot 10^{-3}\,\text{s}$$

für das Audio-Processing.

Zu der gesamten Latenz addieren sich schließlich noch die Latenz für das Füllen des Eingangs-Buffers und die Verzögerungen, die sich am Eingang des Audio-Interfaces bei der A/D-Wandlung beziehungsweise der D/A-Wandlung am Audio-Ausgang ergeben (Abbildung 6). In Abhängigkeit von dem Audio-Interface (z.B: PCI, USB, Firewire) addieren sich zusätzlich noch Latenzen für die Übertragung zwischen Interface und Buffer beziehungsweise Buffer und Interface hinzu.

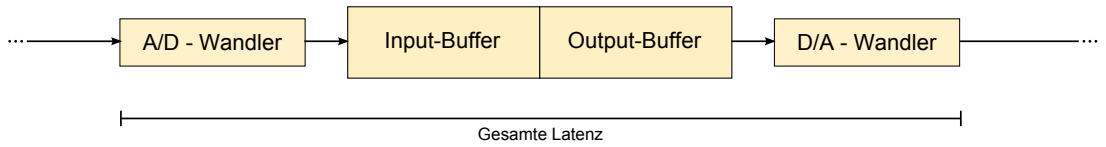

Abb. 6: Vereinfachte Darstellung der für die Latenz maßgeblichen Komponenten des Audio-Interfaces und dessen Treibers.

Bei einem Sample-Buffer von 128 Samples ergibt sich demnach eine Latenz von mindestens 5,8ms (Latenz Eingangs-Buffer + Ausgangs-Buffer).

2.3 ASIO-Schnittstelle

Das Audio-Processing ist in Abbildung 6 nicht mit dargestellt, da es auf die Latenz keinen direkten Einfluss hat. Viel mehr bestimmt die Größe des Ausgangs-Buffers wieviel Zeit für das Audio-Processing zur Verfügung steht. Ein Ausgangs-Buffer mit einer Größe von 128 Samples bedeutet somit, dass das Audio-Processing unter keinen Umständen mehr als 2,9 ms in Anspruch nehmen darf.

Eingangs- und Ausgangs-Buffer sind als Doublebuffer ausgelegt, wobei jeweils einer der beiden Buffer gelesen wird, während der andere beschrieben wird.

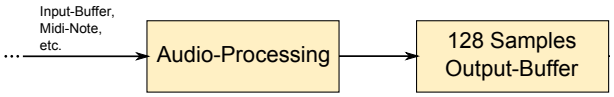

Abb. 7: Position des Audio-Processings im Bezug zum Audio-Interface. Der Eingangs-Puffer ist hier nicht mit dargestellt, da er, beispielsweise bei MIDI-Events für virtuelle Instrumente, nicht zwangsläufig die Quelle für das Audio-Processing sein muss.

Für die unterbrechungsfreie Audiowiedergabe wird vom Audio-Treiber ein Bufferswitch-Aufruf ausgelöst, sobald von der Hardware weitere Audio-Daten benötigt werden. Abbildung 8 zeigt die Situation vor einem Bufferswitch.

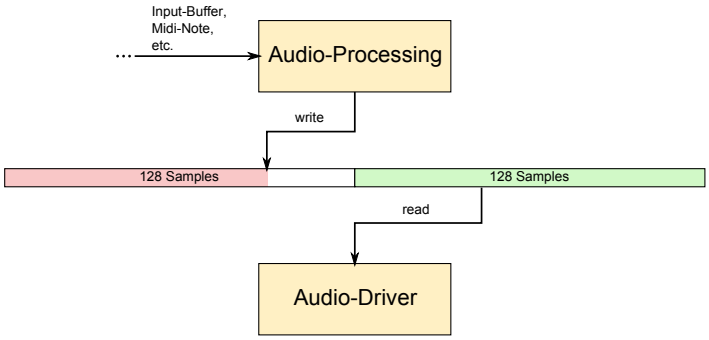

Abb. 8: Zustand des Ausgangs-Buffers vor einem Bufferswitch. Der rot dargestellte Bereich zeigt den Buffer, der von der Audio-Anwendung beschrieben wird, der grün dargestellte Bereich den Buffer, der von dem Audio-Treiber ausgelesen wird.

Die Darstellung zeigt in der Mitte den Ausgangs-Buffer bei einer Samplegröße von 128 Samples. Während die Hardware damit beschäftigt ist, den Ausgangs-Buffer auszulesen, wird von der Audio Anwendung der zweite Buffer mit neuen Audio-Samples beschrieben.

Damit bei der Audio-Ausgabe keine Aussetzer entstehen, muss das Beschreiben des Buffers durch die Audio-Anwendung abgeschlossen sein bevor der Treiber einen erneuten Bufferswitch auslöst. (Abbildung 9)

Hat der Treiber nun seinen Buffer ausgelesen, wird auf den vorher von der Anwendung beschriebenen Buffer umgeschaltet, während die Anwendung damit beginnt, den zuvor vom Trei-

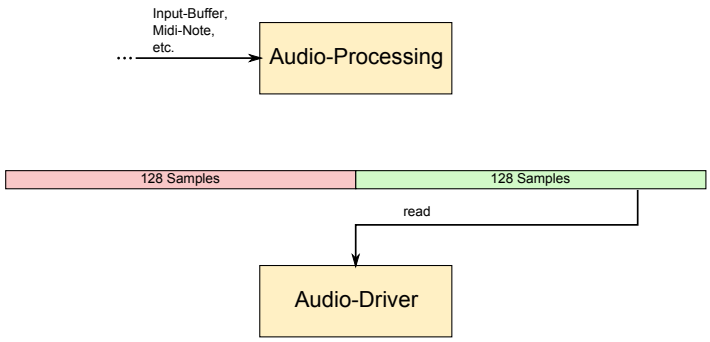

Abb. 9: Zustand des Ausgangs-Buffers vor einem Bufferswitch, nachdem die Audio-Anwendung ihren Buffer fertig beschrieben hat

ber ausgelesenen Buffer wieder neu zu beschreiben. (Abbildung 10)

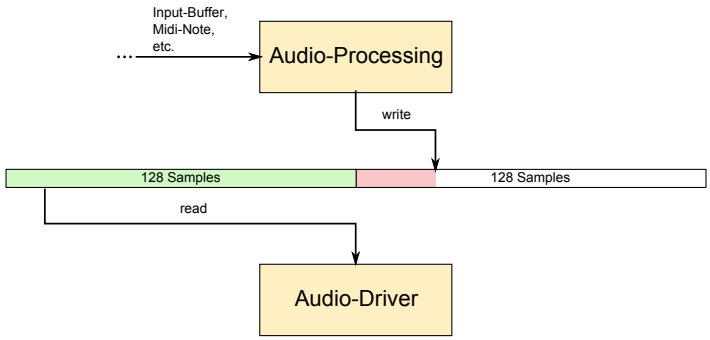

Abb. 10: Ausgangs-Buffer nach dem Bufferswitch. Der vorher von der Anwedung beschriebene Buffer wird vom Audio-Treiber ausgelesen, die Audio-Anwendung beginnt nun damit den zuvor ausgelesenen Buffer neu zu beschreiben.

Sobald der Treiber den Buffer ausgegeben hat, erfolgt erneut ein Bufferswitch, und der Vorgang beginnt wieder bei Abbildung 8[3].

2.4 Routing und Synchronisation

Bei der Verteilung des Audio-Processings auf verschiedene Prozessorkerne stellt sich das Problem, dass Abhängigkeiten, die beim *Routing* der Audiokanäle entstehen, aufgelöst und synchronisiert werden müssen.

Abbildung 11 zeigt den Aufbau eines einfachen Demo-Projektes. Auf der linken Seite sind 3 voneinander unabhängige Audiokanäle zu sehen, für die jeweils der darunter abgebildete Effekt (MonoDelay, AmpSimulator, Reverb) berechnet wird. Die Signale aus den Audiokanälen senden wiederum in die Gruppe auf der rechten Seite, die ihrerseits mit einigen Effekten versehen ist.

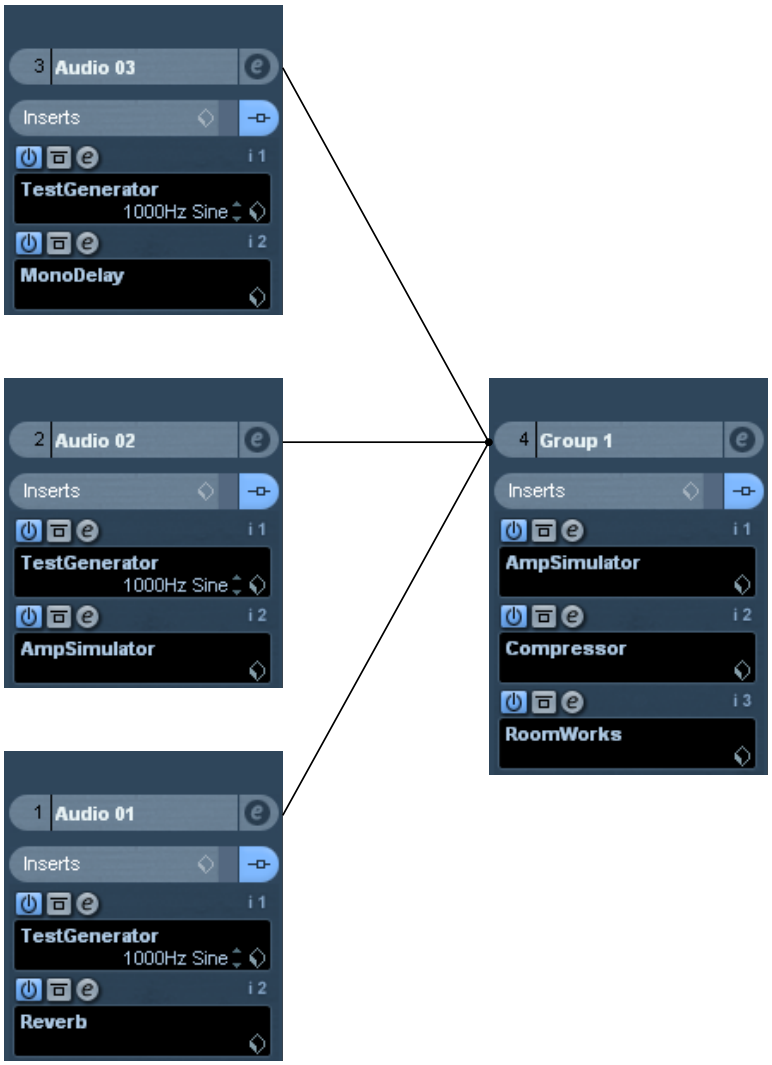

Abb. 11: Beispiel für ein einfaches Routing. 3 Audio-Kanäle routen in Gruppe 1. Jeder der Audio-Kanäle ist mit einem unterschiedlichen Effekt versehen. Audio 1 hat den Effekt mit der längsten Berechnungszeit, Audio 3 den mit der schnellsten.

Die unterschiedlichen Effekte der Audio-Kanäle wurden bewusst so ausgewählt, dass sich Berechnungszeiten ergeben, die sich deutlich voneinander unterscheiden.

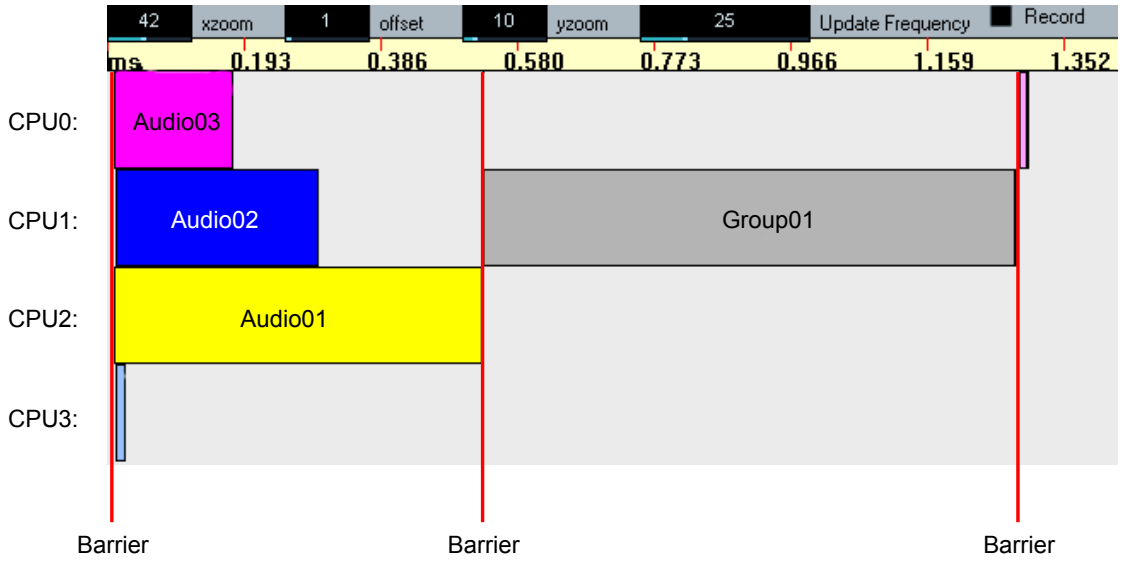

Abb. 12: Resultierendes Processing. Jeder farbige Block stellt eine Process-Unit dar. Die Gesamtlänge bis zum letzten Block enspricht der Rechenzeit für einen ASIO-Block.

In Abbildung 12 ist das daraus resultierende Processing und die Synchronisierung auf einer Quad-core CPU zu sehen. Die unterschiedlichen Balken zeigen von links nach rechts den Zeitverlauf und von oben nach unten die jeweilige CPU auf der die *Process-Unit* berechnet wird. Deutlich ist zu sehen, dass der Effekt für Gruppe 1 erst dann berechnet wird, nachdem die Berechnung aller Audio-Spuren abgeschlossen ist.

Für jede CPU wird dafür eine Liste mit Process-Units erstellt, die der Reihe nach abgearbeitet wird. Zwischen den Process-Units für das Audio-Processing sind jeweils Barriern eingefügt, die die Audio-Threads miteinander synchronisieren. Die so berechneten Samples werden anschließend in den Sample-Buffer geschrieben.

Anhand der unterschiedlichen Verarbeitungszeiten der drei Audio-Kanäle lässt sich ein Problem erkennen, das die Grenzen der Parallelisierbarkeit des Audio-Processings aufzeigt. Die Effekte, die in Gruppe 1 berechnet werden, werden für die Summe der Audiosignale aus den drei Audiokanälen berechnet. Mit der Berechnung kann folglich erst dann begonnen werden, wenn der letzte Audiokanal fertig berechnet wurde, CPU3 bleibt während dessen ungenutzt. Zusätzlich ergibt sich aus den unterschiedlichen Laufzeiten der drei Audiokanäle das Problem, dass auch CPU0 und CPU1 so lange unbeschäftigt bleiben, bis CPU2 das Signal für Audio 3 fertig berechnet hat.

3 Mikroprozessor-Architektur

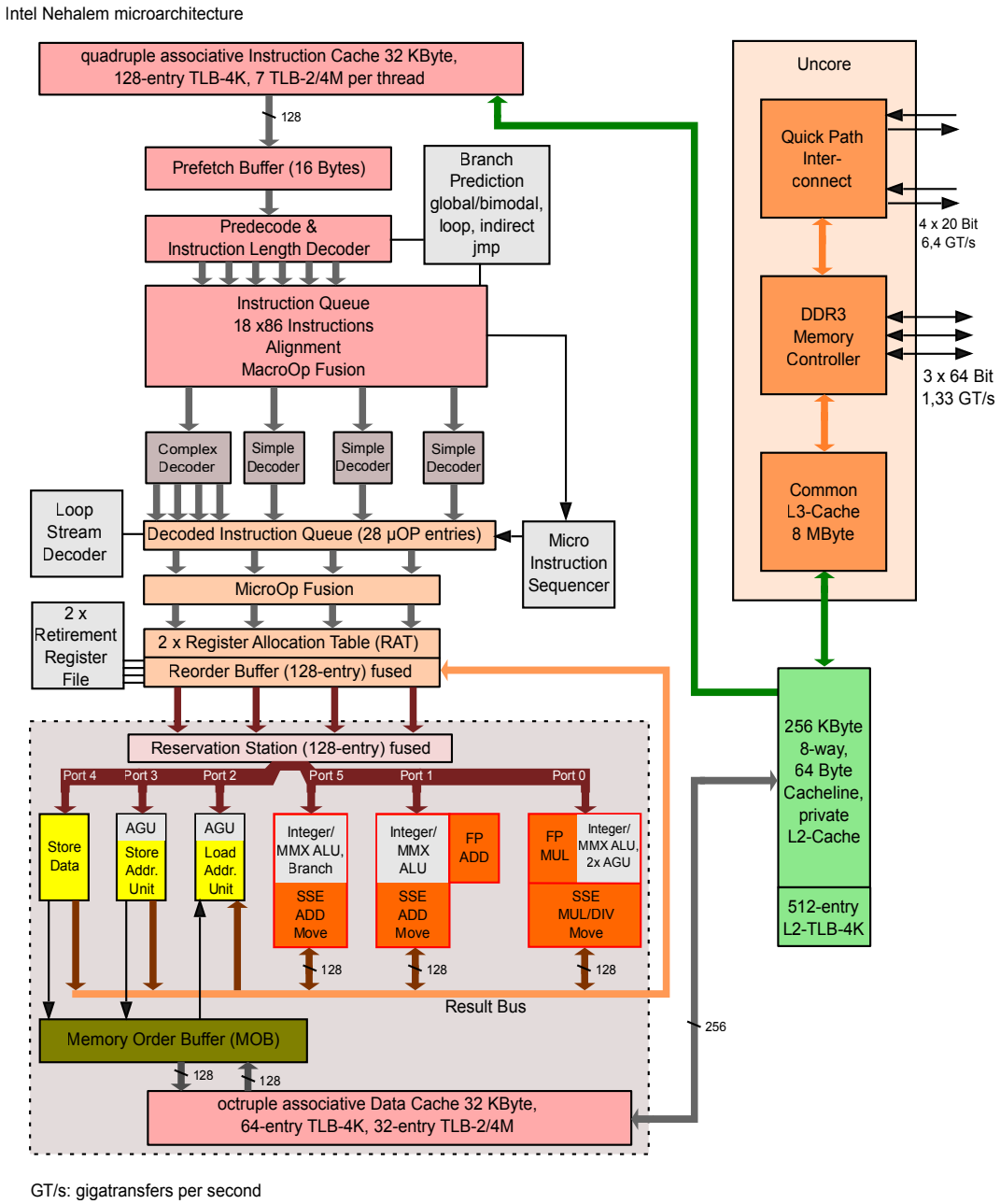

Abb. 13: Nehalem-Mikroarchitektur

Abbildung 13 stellt den grundlegenden Aufbau der Nehalem-Mikroarchitektur dar. Von oben nach unten findet sich die Reihenfolge für das Einlesen (Kapitel 3.3), Dekodieren (3.4) und Ausführen (3.5) wieder. Der grau umrandete Kasten zeigt die Bestandteile eines CPU-Kerns. Im folgenden Kapitel wird genauer auf alle Bestandteile dieser Grafik eingegangen, die in den folgenden Kapiteln von Bedeutung sein werden.

Die Beschreibung sämtlicher Vorgänge innerhalb des Prozessors würde den Rahmen dieser Arbeit sprengen. Für eine grundlegende Einführung zu der Funktionsweise der hier darge-

stellten Komponenten des Prozessors kann ich den Artikels im Intel Technology Journal[6] empfehlen. Der Artikel befasst sich mit der Funktionsweise der NetBurst-Architektur des Pentium 4, auf der auch die Nehalem-Architektur aufbaut. Erläuternd dazu wird der Artikel in Kapitel 4.6 aus dem Buch über Computerarchitektur von Tanenbaum[16] gut zusammengefasst.

3.1 Cache und Speicherzugriff

Selbst der schnellste Arbeitsspeicher (*RAM*) ist nicht in der Lage, Daten in der Geschwindigkeit zu liefern, die eine moderne CPU verarbeiten kann. Würde eine CPU all ihre Daten direkt aus dem Hauptspeicher abrufen, müsste sie für jeden Zugriff in Abhängigkeit von Speichertyp und Zugriffsmuster weit über hundert Takte warten[1]. Der größte Teil der Leistungsfähigkeit einer CPU würde in *Stall-Cycles* ungenutzt bleiben.

Um dieses Problem zu umgehen, befindet sich zwischen der CPU und dem Hauptspeicher ein meist als *SRAM* implementierter Pufferspeicher (Cache), der die Daten zwischenspeichert. Im günstigsten Fall können die Daten innerhalb eines CPU-Taktes geliefert werden, so dass die CPU unterbrechungsfrei arbeiten kann. In Abbildung 14 ist der grundlegende Aufbau einer solchen Speicherarchitektur dargestellt[17].

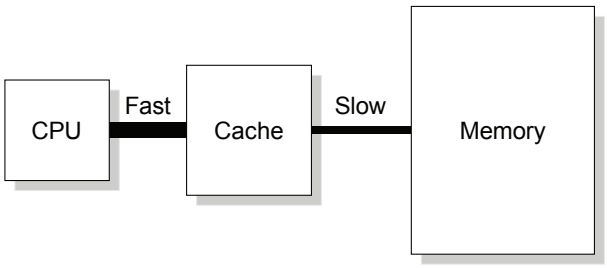

Abb. 14: Grundlegendes Prinzip der Speicherarchitektur

Um den Aufwand eines Datentransfers aus dem Hauptspeicher möglichst gering zu halten, werden Daten in Cache-Blöcken eingelesen. Die Größe dieser Blöcke wird üblicherweise als Cache Line Size angegeben. Das Beispiel in Abbildung 13 weist hier eine Größe von 64 Byte aus, die mit Hilfe des Tools cpuz[25] auch bestätigt werden konnte[2].

3.1.1 Mittlere Zugriffszeit

Die mittlere Zugriffszeit auf den Speicher ist maßgeblich für die Gesamtgeschwindigkeit des Systems. Wenn eine CPU wiederholt auf einen Speicherbereich zugreifen muss, werden die entsprechenden Daten[3] zunächst in den Cache-Speicher geladen. Bei darauf folgenden Zugriffen

[1] Beispielsweise *DDR*2-1066 Speicher mit einem Timing von 5-5-5-15-2T. Im ungünstigsten Fall addieren sich die Werte zu 32 Bustakten. Bei einem Bustakt von 533 MHz und einem CPU-Takt von 2.8GHz würde das bedeuten, dass die CPU

$$\frac{2800}{533} * 32T = 168T$$

Takte warten muss. [18] [22]

[2] In Kapitel 6.3 Ist die Größe der Cache-Blöcke ausschlaggebend für die unterschiedlichen Messergebnisse der Prototypen M1 und M2.

[3] Beginnend mit dem ersten Daten*Wort* wird immer ein Block von der Größe der *Cache-Line-Size* eingelesen

kann die CPU anschließend wesentlich schneller auf die Daten zugreifen. Wird also *n* Mal wiederholt auf einen bestimmten Speicherbereich zugegriffen, muss nur ein einziges Mal auf den Hauptspeicher und *n-1* Male auf den Cache-Speicher zugegriffen werden.

Die Hit *Ratio h* (Anteil der Cache-Treffer) lässt sich für den einfachen Fall im obigen Beispiel mit der Formel *h = (n - 1) / n* errechnen. Um diesen Zusammenhang in einer Formel auszudrücken, muss die Cache-Zugriffszeit *c*, die Hauptspeicher-Zugriffszeit *m* und die Miss Ratio *(1 - h)* (Fehlerquotient) in die Berechnung mit einbezogen werden. Daraus ergibt sich die Formel:

$$\textit{Mittlere Zugriffszeit} = c + (1 - h)m$$

In Kapitel 6.3 wird diese Formel wieder interessant, da sich die unterschiedlichen Messergebnisse für die Prototypen M1 und M2 mit dieser Formel erklären lassen.

3.1.2 Speicherhierarchie

Bei der Speicherarchitektur moderner Prozessoren ist der Cache in unterschiedliche Ebenen aufgeteilt. Der L1 Cache befindet sich meist mit auf dem *Die* und wird mit vollem Prozessortakt betrieben. Der Platz ist daher begrenzt, so dass dieser Speicher nur wenige Kilobyte aufnehmen kann. Der L2 Cache ist etwas größer und verfügt nicht über eine direkte Anbindung zum CPU-Kern, die Zugriffszeiten sind hier entsprechend länger (etwa 5 Takte). Bei der Nehalem-Architektur in Abbildung 13 befindet sich zwischen L2 Cache und Arbeitsspeicher zusätzlich noch ein L3 Cache

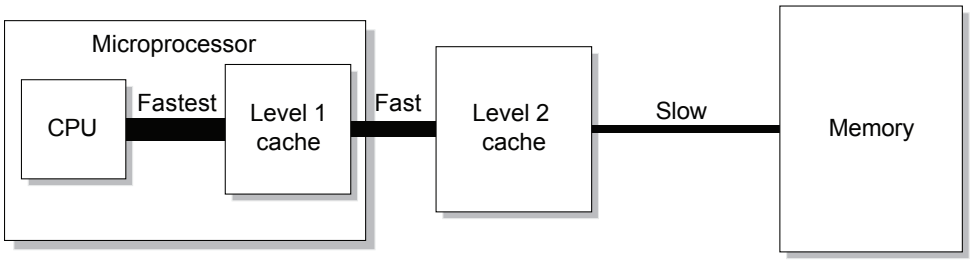

Abb. 15: Speicherhierarchie

Abbildung 15 zeigt den Aufbau der Speicherhierarchie eines Microprozessors mit einem CPU-Kern. In Kapitel 4 wird auf die Zuordnug der unterschiedlichen Cache-Levels noch genauer eingegangen.

3.1.3 Virtueller Speicher

Vom System vergebene Speicherardressen können über den tatsächlich verfügbaren Arbeitsspeicher hinausreichen. Um zu ermöglichen auch dann noch Speicher zur Verfügung zu stellen, wenn der physikalisch vorhandene Speicher bereits voll belegt ist, können Teile des Arbeitsspeichers auf die Festplatte ausgelagert werden.

Um diesen Auslagerungsvorgang für die CPU transparent zu halten, wird auf den Speicher anhand von virtuellen Speicheradressen zugegriffen. Da die Abbildung von virtuellen auf physikalische Speicheradressen sehr zeitaufwändig ist, wird ein zusätzlicher Cache genutzt, der sogenannte Translation Lookaside Buffer (TLB). Im TLB werden Abbildungen von virtuellen auf physikalische Adressen gecached und wiederverwendet, damit der Aufwand für die Erstellung eines Eintrages nach Möglichkeit amortisiert wird.

3.1.4 Cache Funktionalitäten

Der L1 Cache besteht bei der Nehalem Architektur aus zwei verschiedenen Speichern mit unterschiedlichen Aufgaben. Der **Instruction Cache** wird benutzt, um Befehle zur Ausführung zwischenzuspeichern. Im **Data Cache** werden Daten, wie zum Beispiel Variablenwerte, gepuffert. Cache Speicher, der sowohl Instruktionen als auch Daten enthalten kann, wird als **Unified Cache** bezeichnet.

3.1.5 Translation Lookaside Buffer TLB

Virtuelle Speicheradressen könne Speicher adressieren, der auch ausserhalb des physikalisch vorhanden Speichers liegen kann.

Damit die CPU auf virtuellen Speicheradressen arbeiten kann, müssen die entsprechenden Abbildungen auf die physikalischen Adressen vom System erstellt werden. Dieser Übersetzungsvorgang nimmt relativ viel Zeit in Anspruch. Um diesen Effekt etwas abzumildern, werden Adressen, auf die in der Vegangenheit bereits zugegriffen wurde, im TLB zwischengespeichert.

Für Nehalem Prozessoren gibt es jeweils einen eigenen TLB für Instruktionen (ITLB) und für Daten (DTLB).

3.1.6 Zusammenfassung

Setzt man die beschriebenen Komponenten zusammen, erhält man eine Architektur, die sich so oder ähnlich nicht nur bei Nehalem, sondern auch in den meisten anderen modernen Microprozessor-Architekturen wiederfindet. (Abbildung 16)

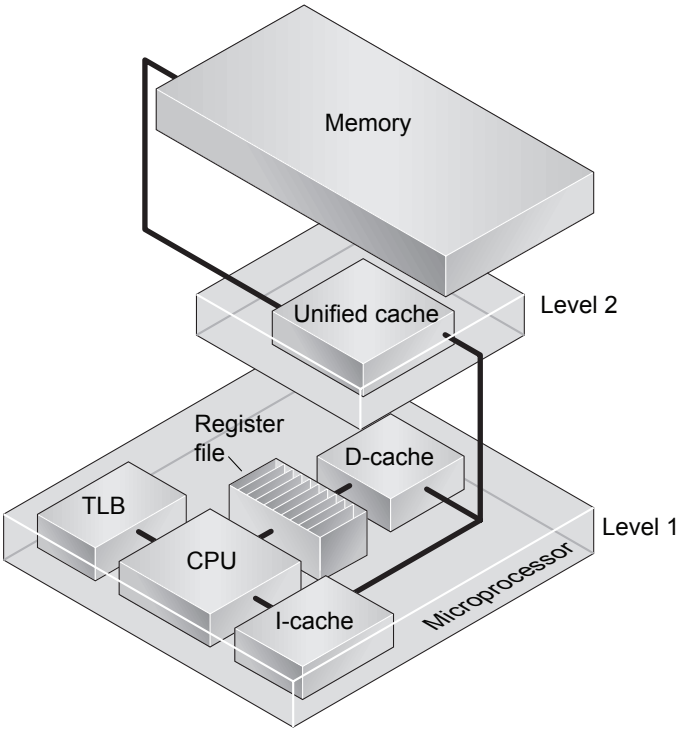

Abb. 16: Dual-Prozessor-Architektur

Die Bezeichung ’unified Cache’ bezieht sich hier auf die Funktion des L2 Caches, der sowohl Instruktionen als auch Daten enthalten kann.

3.2 Instruction Pipeline

Jede moderne (und auch nicht mehr ganz so moderne) Microarchitektur macht sich das Prinzip des Pipelining zunutze. Das Ausführen einer einzelnen Instruktion erfordert mehrere Schritte und ist somit nicht in einem einzigen Takt möglich. Die Ausführung teilt sich typischerweise in folgende Schritte auf:

- Instruction Fetch
  Das Einlesen der Instruktion aus dem Instruction Cache
- Instruction Decode
  Das Decodieren der Instruction in MicroOperations die von den Execution Units ausgeführt werden können
- Execute
  Die eigentliche Ausführung der Operation
- Write Back
  Das Zurückschreiben des Ergebnisses in den Speicher (L1 Cache)

Vom Einlesen der Instruktion bis zum Zurückschreiben des Ergebnisses vergehen also 4 Takte. Es wäre dementsprechend ineffektiv, jedes mal zu warten bis eine Instruktion die komplette Pipeline durchlaufen hat; bevor der nächste Befehl ausgeführt wird.

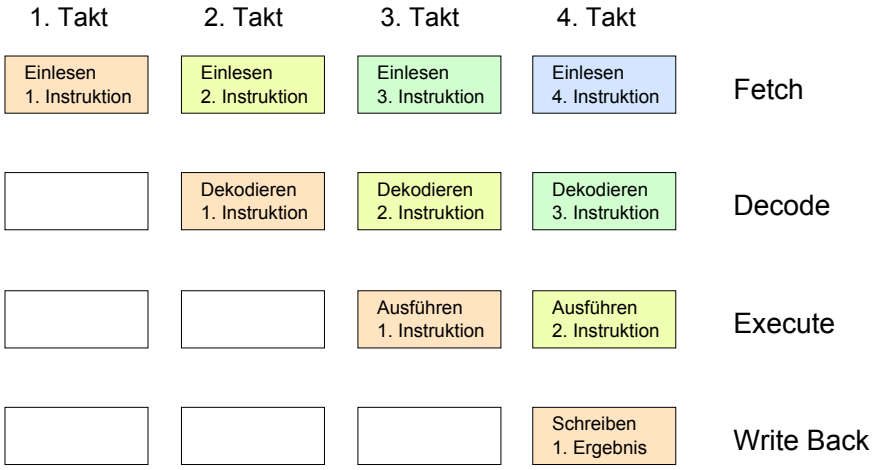

Abb. 17: Instruction Pipelining

Um das zu verhindern, wird nach jeder Stufe in der Pipeline bereits die nächste Instruktion nachgereicht. Während die 1. Intruktion dekodiert wird, wird also schon die 2. Instruktion eingelesen (Abb. 17). Auf diese Weise benötigt eine Instruktion im Idealfall nur einen Takt für die Verarbeitung.

Die Implementierung einer Pipeline in der NetBurst-Architektur, auf der der Pentium 4 basiert, ist etwas komplexer und erstreckt sich über 20 Stufen. Abbildung 18 zeigt den Vergleich zwischen der Instruction-Pipeline eines Pentium III mit der eines Pentium 4 Prozessors.

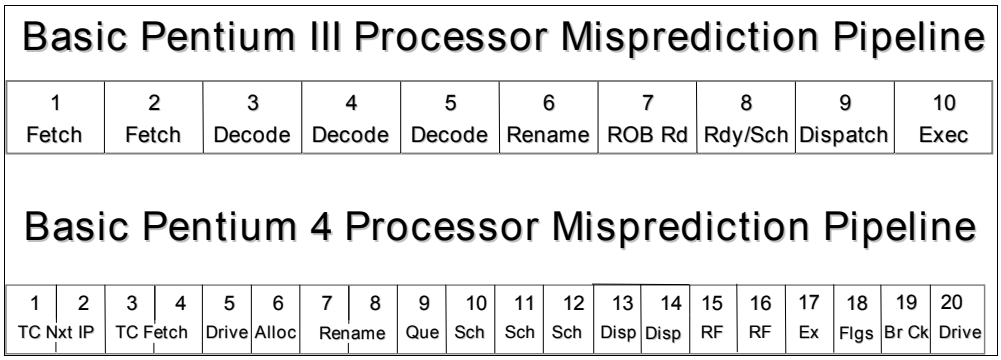

Abb. 18: Instruction Pipeline eines Pentium III und eines Pentium 4 im Vergleich.

Die Pipeline wurde bei der neueren Architektur in kleinere Zwischenschritte aufgeteilt, um innerhalb der einzelnen Pipelinestufen eine einfachere Schaltungslogik verwenden zu können und damit höhere Taktfrequenzen zu ermöglichen[12].

3.3 Einlesen

Zwischen dem Instruction Cache und der Instruction Queue findet das Einlesen der Instruktionen zur Ausführung statt. Hier greifen die ersten prozessorinternen Optimierungen wie die Verzweigungsvorhersage[4] (Branch Prediction) oder das Zusammenfassen mehrer Instruktionen zu einer einzelnen Instruktion (MacroOp Fusion).

Bei Multicore Architekturen gibt es einen Instruction Cache für jeden physikalisch vorhandenen CPU-Kern, so dass die Instruktionen auf dem CPU-Kern ausgeführt werden in dessen Cache sie vom Betriebssystem geschrieben wurden. Bei Hyper-Threading CPUs gibt es einen Instruction Cache für zwei logische CPU-Kerne die auf einem physikalischen CPU-Kern ausgeführt werden.

Wie das Verhalten des Betriebssystems, Threads auf bestimmten CPU-Kernen auszuführen, beeinflusst werden kann, wird in Kapitel 4.4 genauer erläutert.

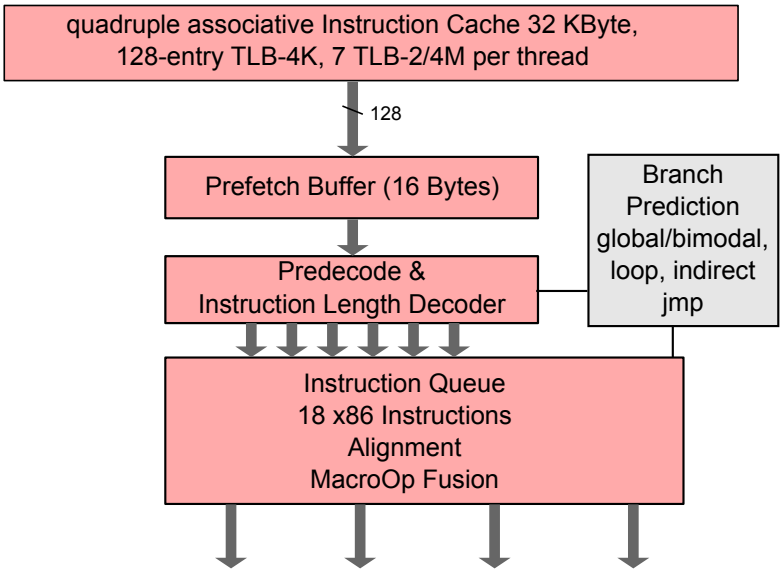

Abb. 19: Instruction fetch

3.4 Dekodieren

In diesem Schritt werden die *x86* Instruktionen in maschinenlesbare Micro-Operations (uOps) oder auch opCodes dekodiert. In Abbildung 20 sind ein Complex Decoder und drei Simple

[4] Verändern beispielsweise zwei direkt aufeinanderfolgende Instruktionen dieselbe Variable, müsste die zweite Instruktion warten bis das Ergebnis in den Speicher zurückgeschrieben wurde. Das kann verhindert werden, indem das Ergebnis der Operation direkt aus dem Ergebnisregister der ersten Instruktion ausgelesen wird. Darüber hinaus gibt es auch die Möglichkeit der Vorhersage von Sprüngen und Schleifen. Die genaue Funktionsweise der Vorhersagealgorithmen wird von Intel jedoch geheim gehalten.

Decoder zu sehen. Intel definiert den Unterschied darin, dass ein Simple Decoder eine Instruktion dekodieren kann, aus der genau eine Micro-Operation resultiert, während ein Complex Decoder Instruktionen dekodiert, die zu mehreren Micro-Operations dekodiert werden ([9] S. 27). Mit einen Takt können in diesem Schritt also bis zu 4 Instruktionen dekodiert werden.

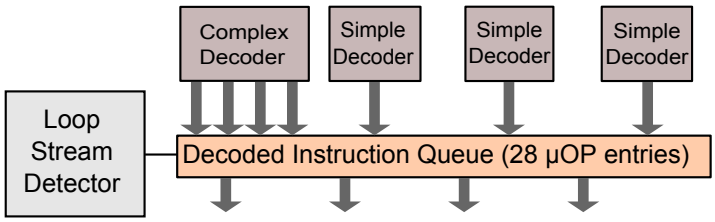

Abb. 20: Decode

3.4.1 Loop Stream Detector

Der LSD erkennt Wiederholungen im *Instruction Stream* und fungiert als Zwischenspeicher für die entsprechenden Anweisungen. Die CPU kann für den Zeitraum der Ausführung alle Elemente vor dem LSD deaktivieren und die Befehle zur Ausführung direkt abarbeiten. Kommen im Code viele Schleifen vor, kann das die Ausführungsgeschwindigkeit erheblich beschleunigen und sorgt zusätzlich für erhöhte Energieeffizienz ([9] S. 25).

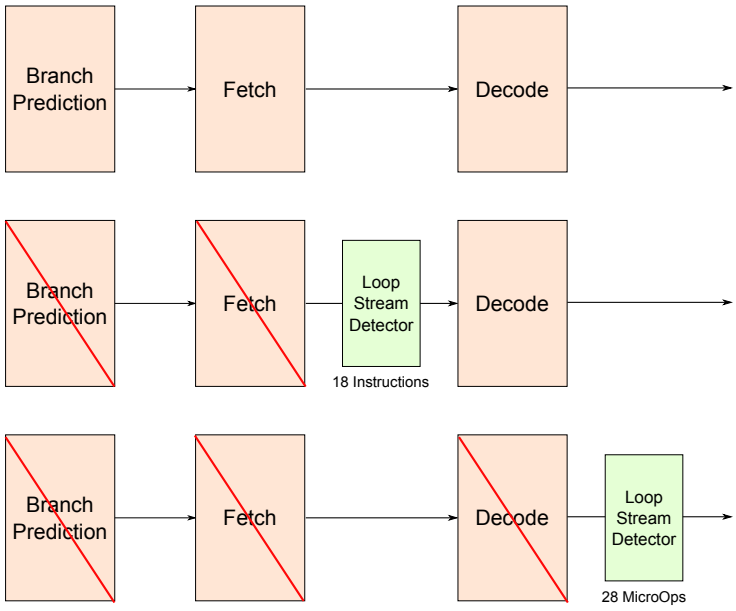

Abb. 21: Loop Stream Detection beim Core 2 Duo (mitte) und beim Core i7 (unten)

In Abbildung 21 ist zu erkennen dass sich die Position des LSD bei der Nehalem-Architektur gegenüber älteren Architekturen verändert hat. Während er bei Core2 Prozessoren noch vor der

Dekodierung eingriff, sitzt er bei Nehalem-Prozessoren dahinter und puffert dekodierte Micro-Operations. Dadurch, dass mit dem Dekodieren ein weiterer Arbeitsschritt eingespart werden kann, wird noch etwas Performance gewonnen und der Prozessor arbeitet etwas energieeffizienter. Beim Profiling ergeben sich daraus mitunter etwas verwirrende Ergebnisse, wenn die Anzahl der gezählten Instruktionen nicht mit der Anzahl der gezählten Micro-Operationen korelliert.

3.5 Ausführen

In diesem Schritt findet die eigentliche Operation statt. In Abhängigkeit von der Operation wird die Aufgabe an eine der in Abbildung 22 dargestellten Functional Units verteilt. Jede Functional Unit ist für bestimmte Befehle optimiert, so dass nur selten alle Functional Units voll ausgelastet sind. Hyper-Threading setzt an genau dieser Stelle an und versucht die Auslastung zu maximieren, indem die Threads aus zwei Instruction Streams gleichzeitig bearbeitet werden. Die Idee mehrere Functional Units gleichzeitig zu beschäftigen, wird auch als superskalare Architektur bezeichnet und reicht schon sehr viel weiter zurück[19].

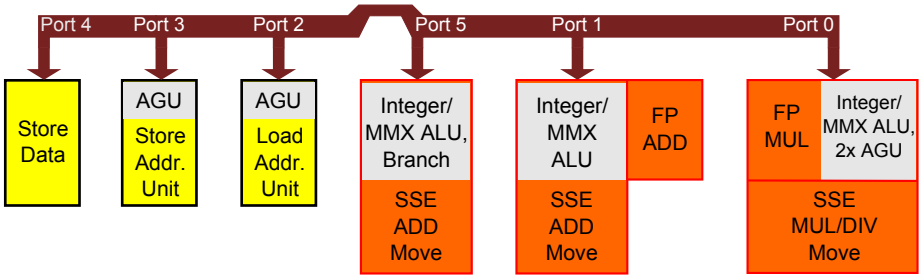

Abb. 22: functional units in der Nehalem Architektur

- *ALU* - Arithmetic Logical Unit, Integer / MMX ALU:
  Ganzzahlige Berechnungen und logische Operationen wie AND, XOR usw.

- *FPU* - Floating Point Unit, orange dargestellt:
  Berechnungen von *Gleitkommazahlen*

- LOAD/STORE - gelb dargestellt:
  Lesen und Schreiben von Daten

3.6 Probleme von Intels 32-Bit Architekturen (IA-32)

Setzt man sich genauer mit der Architektur von 32-Bit Prozessoren auseinander, wird sehr deutlich, dass das gesamte System sehr aufwändig ist und sehr viele komplexe Steuerungen notwendig sind, um einen reibungslosen Betrieb eines in IA-32-Architektur gefertigten Prozessors zu gewährleisten.

Im Prozessor wird die Reihenfolge der Befehlsausführung verändert, es müssen Sprungvorhersagen gemacht werden, fehlgeschlagene Vorhersagen werden zwischengespeichert und verworfen, um nur einige Beispiele zu nennen. Die *Out-of-Order*-Ausführungs-Logik eines Pentium 4 kann bis zu 126 Instruktionen gleichzeitig bereithalten, von denen wiederum 48 Load- und 24 Store-Instruktionen sein können ([6] Out-of-Order Execution Logic). Diese Instruktionen sind teilweise untereinander abhängig, so dass mit steigender Komplexität des Prozessors die Kosten für Chipfläche und Ausführungsgeschwindigkeit immer weiter ansteigen([16] Kapitel 5.8.1).

Grund für diesen großen Komplexitäts-Overhead ist die notwendige Abwärtskompatibilität dieser Architektur, die bis zum *8086* bzw. *8088* aus den späten 70er Jahren zurückreicht. Damals wurde von Intel der Ansatz verfolgt, mit einem möglichst großen Befelssatz dafür zu sorgen, die Diskrepanz zwischen *Hochsprache*n und deren Übersetzung zu maschinenlesbarem *Binärcode* zu verringern. Mit dem ersten 32-Bit Prozessor von Intel, dem *80386*, wurde die IA-32 Architektur eingeführt, die bis heute von allen Nachfolgemodellem im wesentlichen übernommen wurde. Diese auf den frühreren Modellen aufbauende CISC[5]-Architektur hat den Nachteil, dass der Prozessor, neben der eigentlichen Ausführung der Befehle, im Gegensatzt zu einer einfacheren RISC[6]-Architektur einen sehr viel höheren 'Verwaltungsaufwand' hat.

Langfristig betrachtet könnte die Lösung dafür in der von Intel und HP gemeinsam entwickelten IA-64-Architektur[7] liegen, die zurzeit ausschließlich in Intels *Itanium*-Reihe zum Einsatz kommt. Der grundlegende Unterschied dieser Architektur liegt darin, dass nicht der Prozessor während der Ausführung die Reihenfolge der Instruktionen und die parallele Ausführung auf unterschiedlichen Functional Units festlegt, sondern dass bereits die vom *Compiler* erzeugten Instruktionen so angeordnet werden, dass voneinander abhängige Instruktionen nicht genau hintereinander ausgeführt werden und die gesamte komplexe Logik im Prozessor entfallen kann.

Dem Prozessor liegt die RISC-Architektur zugrunde. Er kennt nur einen deutlich kleineren Befehlssatz als ein CISC-Prozessor und muss die Befehle nicht selbst in mehrere Mikro-Operationen

[5] Complex Instruction Set Computer

[6] Reduced Instruction Set Computer - gegenüber CISC-Prozessoren vereinfachte Prozessorarchitektur, erstmals diskutiert von Patterson und Ditzel in [5]

[7] Nicht zu verwechseln mit Intel 64 oder AMD64 bzw. x86-64 oder x64 die lediglich 64-Bit Erweiterungen für IA-32 sind

zerlegen. Stattdessen werden vom Compiler mehrere voneinander unabhängige Befehle in ein *Maschinenwort* fester Länge kodiert, die von dem Prozessor, ohne weiter darüber 'nachdenken' (Sprungvorhersage, Tracing etc.) zu müssen, übersetzt und ausgeführt werden können. Dieses Verfahren wird auch VLIW[8]-Machine genannt und kommt in den Itanium-Prozessoren in einer speziellen Variante zum Einsatz, in der immer genau drei Maschinenworte zusammengefasst werden. Von Intel und HP wird dieses Verfahren als EPIC[9] bezeichnet[4].

[8] Very Long Instruction Word
[9] Explicit Parallel Instruction Computing

4 Unterschiedliche Mehrkern-Architekturen

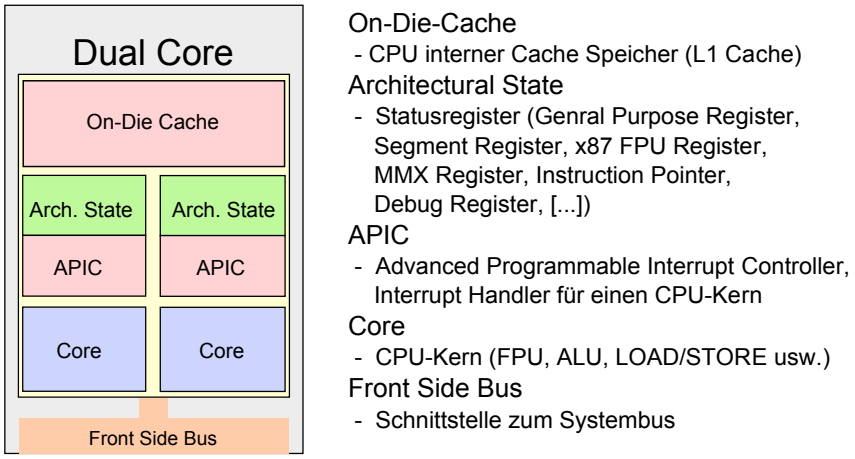

Abb. 23: Legende am Beispiel einer Dual-Core Architektur

4.1 Dual-Prozessor

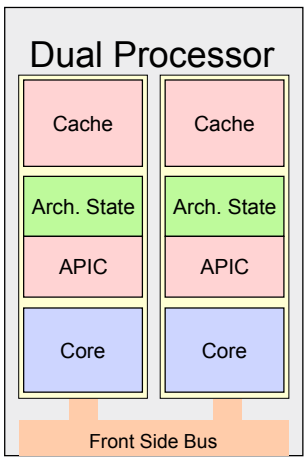

Abb. 24: Dual-Prozessor-Architektur

Die früheste Version von Mehrkern-Systemen verfügte noch über mehrere CPUs, die sich auf unterschiedlichen Sockeln befanden. Jede CPU verfügte über eine eigene Anbindung an den Front-Side Bus und einen seperaten L1 Cache. Das kann insbesondere bei häufigen Speicherzugriffen sehr problematisch werden. Will CPU A auf Daten zugreifen, die im L1 Cache von CPU B liegen, muss zunächst der Weg über den viel langsameren Front-Side Bus gegangen werden.

4.2 Dual-Core

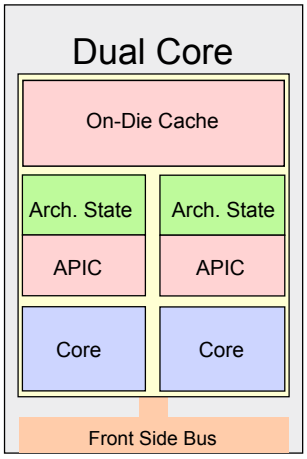

Abb. 25: Dual-Core-Architektur

Üblicherweise sind bei Dual-Core CPUs 2 physikalische Prozessorkerne auf einem Die untergebracht. Die Verwendung des L1-Cache kann dabei auf unterschiedliche Arten erfolgen:

- eigener L1-Cache für jeden CPU-Kern
- ein gemeinsam genutzter L1 Cache
- L1 Cache aufgeteilt in gemeinsam und exklusiv genutzte Speicherbereiche

Die einzelnen Kerne haben die Möglichkeit untereinander zu kommunizieren und machen damit die Verwendung des Front-Side Busses für Zugriffe auf den Speicher eines anderen Kernes überflüssig. Im Gegensatz zu einem Dual-Prozessor System bedeutet das einen erheblichen Performancegewinn bei Aufgaben, für die zwei Kerne auf den gleichen Speicherbereich zugreifen müssen.

4.3 Hyper-Threading

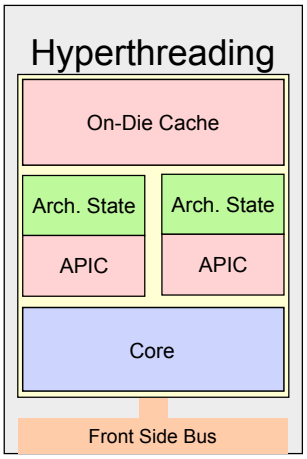

Abb. 26: Hyper-Threading

Mit Hyper-Threading wird versucht einen CPU-Kern besser auszulasten, indem zwei Threads gleichzeitig auf einem Kern ausgeführt werden. Die CPU verfügt dafür über doppelt ausgelegte Status-Register und *Instruction Pointer*. Sie entscheidet nach außen hin transparent, wann ein Befehl aus einem der beiden Instruction-Streams bearbeitet wird. Dem Betriebssystem gegenüber wird 1 CPU-Kern als 2 Kerne dargestellt.

Der *Scheduler* innerhalb des CPU-Kerns muss dafür Sorge tragen, dass die unterschiedlichen Instruction-Streams der beiden Threads möglichst gleichmäßig auf die zur Verfügung stehenden Execution-Units verteilt werden. Dabei werden jeweils ein paar Instruktionen des einen und dann ein paar Instruktionen des anderen Streams ausgeführt. Das Umschalten zwischen den beiden Intruction-Streams geschieht basierend auf einem Algorithmus, der versucht, die Zugriffe auf gemeinsam genutzte Ressourcen (zB. Execution-Units und *On-Die-Cache*) möglichst effizient zu verteilen.

Im Ergebnis kann das zu einer besseren Ausnutzung der vorhanden Ressourcen und damit zu besserer Performance führen. Unglücklicherweise kann die Performance aber auch beeinträchtigt, werden sobald zwei Instructions-Streams Zugriff auf dieselbe Ressource benötigen.

Um die Ausführung von zwei Threads auf einem Kern zu ermöglichen, müssen einige Register für jeden logischen Kern doppelt ausgelegt werden: [8]

- General Purpose Register
- Segment-Register
- EFLAGS und EIP-Register, Instruction- und Code-Segment Pointer zeigen hier auf den jeweiligen Instruction-Stream für einen logischen Prozessor

- x86 FPU-Register
- weitere Status-Register zb. MMX, DEBUG, Time Stamp Counter, etc.

Da ein Kern einer Core i7 CPU über 6 unterschiedliche Execution-Units verfügt, wäre es bei voller Auslastung aller Units theoretisch möglich 6 micro-Ops[10] während eines Taktes auszuführen. Die optimale Ausnutzung der Execution-Units wird allerdings durch mehrere Faktoren limitiert. In der folgenden Abbildung ist der Aufbau eines Core i7 Prozessorkerns schematisch dargestellt.

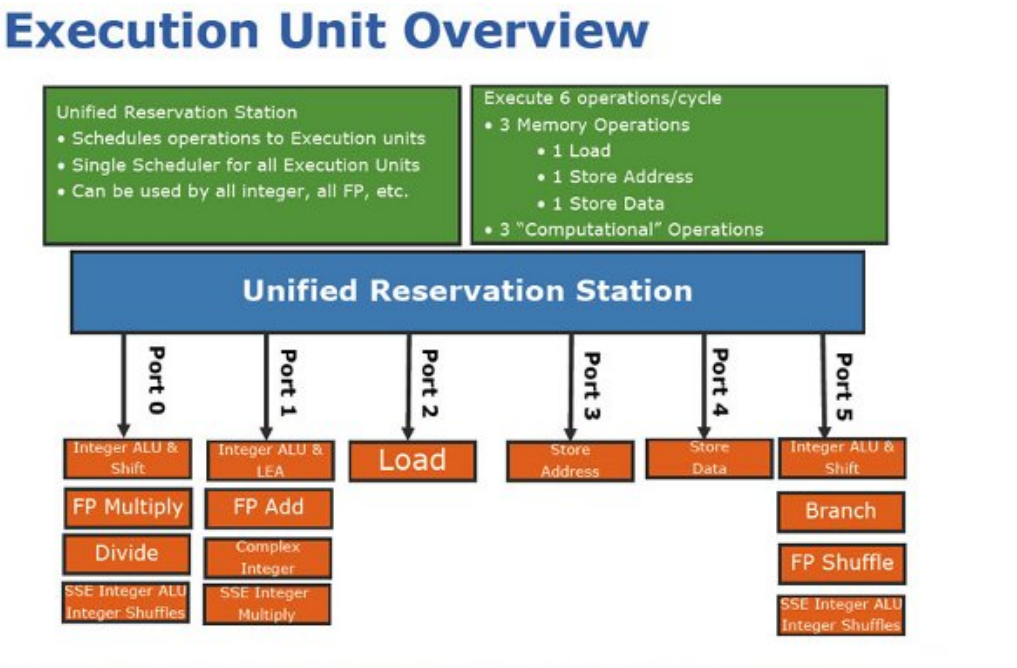

Abb. 27: Execution Units eines Nehalem (Core i7) Prozessors. [21]

Wie in Abbildung 27 zu sehen ist, unterscheiden sich die Execution-Units in ihren Aufgaben. Hyper-Threading macht sich diese Eigenschaft zunutze, indem es unterschiedliche Micro-Ops möglichst gleichmäßig auf die unterschiedlichen Execution-Units verteilt, um so eine höhere Auslastung des gesamten CPU-Kernes zu erreichen.

Folgende Sequenz von Micro-Ops könnte also theoretisch in einem Taktzyklus ausgeführt werden.

- Load
- Store
- Store Address

[10] In opcodes übersetzte *Assembler*-Befehle

- 3 Berechnungen auf ALU oder FPU

Die Wahrscheinlichkeit, dass genau dieser Instruktions-Mix auftritt, ist relativ gering. Beim Audio-Processing liegt der Schwerpunkt auf *Floating-Point*-Berechnungen, so dass sich das bestmögliche Verhältnis von Takt zu Micro-Ops auf 3 Micro-ops pro Takt reduziert.

4.3.1 Besonderheiten

Im Zusammenhang mit Hyper-Threading treten bei der parallelen Verarbeitung von 2 Threads auf einem CPU-Kern Probleme auf, denen es bei Echtzeit-Anwendungen besondere Aufmerksamkeit zu widmen gilt.

physisch versus logisch: Unglücklicherweise ist es weder unter Windows XP noch unter Windows Vista möglich herauszufinden, welche logischen CPU-Kerne sich gemeinsam auf einem physikalischen Kern befinden. Es gibt zwar eine Empfehlung von Intel, die vorgibt in welcher Art und Weise die CPU-*Affinität* (4.4) gesetzt werden sollte, es bleibt aber abhängig von den BIOS-Entwicklern, ob diese Vorgaben auch umgesetzt werden. Vom BIOS falsch ergebene CPU-Affinitäten können die Performance damit im ungünstigsten Fall auf die Hälfte reduzieren.

Thread-Prioritäten: Besonders kritisch ist die Tatsache, dass alle Informationen über *Thread-Prioritäten* verloren gehen. Der Scheduler vom Betriebssystem kennt keinen Unterschied zwischen logischer und physikalischer CPU und delegiert Instruktionen eines niedrig priorisierten Threads möglicherweise an einen (physikalischen) CPU-Kern, der bereits mit der Abarbeitung eines zeitkritischen Threads beschäftigt ist. Der Scheduler innerhalb der CPU, der die decodierten Micro-Ops schließlich an die Execution-Units verteilt, weiß nicht mehr, ob er gerade für einen Echtzeit- oder einen *Idle-Thread* arbeitet.

4.4 CPU-Affinität

Normalerweise entscheidet der Scheduler des Betriebssystems auf welchem CPU-Kern ein Thread ausgeführt wird. Entscheidungsgrundlage für die Verteilung der Threads in einem multicore-System sind unter anderem:

- Thread Priorität
- Verfügbarkeit von CPU-Kernen
- Speicherarchitektur[11]
- 'Wartezeit' eines Threads

Unter manchen Umständen (zum Beispiel beim Audio-Processing) ist es sinnvoll dem Scheduler Vorgaben für die Verteilung eines Threads auf einen bestimmten CPU-Kern zu machen. Dies kann durch die Vergabe einer CPU-Affinität für einen Thread erreicht werden.

Die CPU-Affinität für einen Thread kann unter Windows mit dem Aufruf der folgenden API-Funktion gesetzt werden:

```
DWORD_PTR WINAPI SetThreadAffinityMask(
  __in  HANDLE hThread,
  __in  DWORD_PTR dwThreadAffinityMask
);
```

Die Affinität wird in Form einer Bitmaske übergeben. Für jeden CPU-Kern kann jeweils ein Bit gesetzt werden. Mit einem ähnlichen Aufruf (Get- oder SetProcessAffinity) kann auch die Affinität für einen *Prozess* gesetzt oder abgefragt werden.

Bei aktiviertem Hyper-Threading wird jeder logischen CPU eine eigene Nummer zugewiesen. Auf einem System mit 4 logischen Kernen würde uns der Aufruf von GetSystemAffinity folglich 1+2+4+8 = 15 liefern.

In welcher Reihenfolge die logischen CPU-Kerne in der Affinitätsmaske auftauchen, ist abhängig von der Reihenfolge in der sie beim Hochfahren des Systems vom BIOS initialisiert werden. Einer Empfehlung von Intel folgend, sollten dabei die ersten logischen CPU-Kerne jeweils vor den zweiten logischen Kernen in der Affinitätsmaske auftauchen. Für einen Prozessor mit 2 physikalischen CPU-Kernen [A,B] und 4 logischen Kernen [0,1] würde das bedeuten,

[11] In Mehrkern-Systemen kann die Ausführungsgeschwindigkeit auf unterschiedlichen CPU-Kernen variieren. Soll CPU A beispielsweise eine Instruktion ausführen, für die sie auf Daten zugreifen muss, die im L1-Cache von CPU B liegen, wäre CPU B mit der Ausführung vorraussichtlich schneller. http://msdn.microsoft.com/en-us/library/aa363804(VS.85).aspx

dass die Affinitätsmaske die Kerne in der Folge [A(0), B(0), A(1), B(1)] ansprechen sollte. [20]

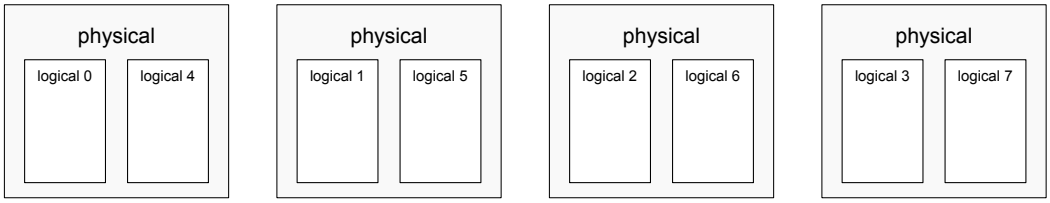

Abb. 28: empfohlene Reihenfolge für die Vergabe der CPU-Affinität bei 4 physikalischen und 8 logischen CPU-Kernen

Diese Verteilung ist insbesondere für Betriebssysteme von Bedeutung, deren Scheduler vor dem ersten Auftauchen von Hyper-Threading entwickelt wurde. Windows 2000 verteilt die Threads der Reihe nach auf die frei werdenden CPU-Kerne. Angenommen, das System arbeitet auf einer CPU mit 2 physikalischen [A,B] und 4 logischen Kernen [0,1], würden nun die Kerne A[0] und A[1] als erstes in der Affinitätsmaske auftauchen. Das hätte zur Folge, dass CPU B praktisch ungenutzt bliebe. [2]

4.5 Intel Hyper-Threading-Prozessoren

4.5.1 Intel32 - Processoren

Intel Processor	Einführungs-datum	Microarchitecture	Frequenz bei Einführung	Transistoren
Intel Xeon Processor	2002	Intel NetBurst Microarchitecture, Hyper-Threading Technology	2.20 GHz	55 M
Intel Pentium 4 Processor Supporting Hyper-Threading Technology at 90 nm process	2004	Intel NetBurst Microarchitecture, Hyper-Threading Technology	3.40 GHz	125 M

4.5.2 Intel64 - Processoren

Intel Processor	Einführungs-datum	Microarchitecture	Frequenz bei Einführung	Transistoren
64-bit Intel Xeon Processor with 800 MHz System Bus	2004	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.60 GHz	125 M
64-bit Intel Xeon Processor MP with 8MB L3	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.33 GHz	675 M
Intel Pentium 4 Processor Extreme Edition Supporting Hyper-Threading Technology	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.73 GHz	164 M
Intel Pentium Processor Extreme Edition 840	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Dual-core	3.2 GHz	230 M
Dual-Core Intel Xeon Processor 7041	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Dual-core	3.00 GHz	321 M
Intel Pentium 4 Processor 672	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Intel Virtualization Technology.	3.80 GHz	164 M
Intel Core i7-965 Processor Extreme Edition	2008	Intel microarchitecture (Nehalem); Quadcore; Hyper-Threading Technology; Intel QPI; Intel 64 Architecture; Intel Virtualization Technology.	3.20 GHz	731 M

5 Zur Performance-Analyse verwendete Profiling-Tools

Um einen Überblick darüber zu geben, wie die gemessenen Werte zustande gekommen sind gibt es in diesem Abschnitt zunächst eine kurze Einführung zu den verwendeten Profiling-Tools. Auf die verwendeten Werkzeuge wird hier sehr genau eingegangen, um die im weitern Verlauf dieser Arbeit ermittelten Messwerte möglichst genau nachvollziehbar zu machen.

Beide in der Arbeit verwendeten Tool unterscheiden sich grundsätzlich in der Art der Performance-Analyse. Während VTune[23] sich darauf konzentriert, die Effizienz und Ausführungsgeschwindigkeit eines Programmes zu messen liegt der Schwerpunkt des Thread-Profilers[24] darin, eine Ablaufverfolgung parallelisierter Anwendungen darzustellen. VTune bietet, neben den hier dargestellten Möglichkeiten zur Performance-Analyse, noch eine Reihe weiterer Möglichkeiten auf die hier nicht näher eingegangen wird, da sich diese Analysemethoden haupsächlich mit der Optimierung gesamter Anwendung und dem Aufspüren von Flaschenhälsen widmen, was nicht Gegenstand dieser Arbeit ist.

5.1 VTune

VTune bietet vielfältige Möglichkeiten, Flaschenhälse in einer Anwendung oder einer bestimmten CPU aufzuspüren[12]. Für die Analyse von Hyper-Threading ist hier insbesondere das Sampling Profiling interessant, mit dem es ermöglicht wird prozessorbezogene Events aufzuzeichnen. Die Sampling-Daten umfassen sehr allgemein gehaltene Informationen wie die Anzahl der CPU-Takte oder verarbeiteter Instruktionen bis hin zu Events, die nur für spezifische Prozessortypen existieren. Für einen Core i7 wären das beispielsweise eine Reihe von Events, die Informationen zu dem On-Die Memory-Controller liefern können, der bei vorausgegangen Prozessorarchitekturen noch nicht in den Chip integriert war.

Auf Basis der gesammelten Sampling-Daten können Verhältnisse zwischen bestimmten Events frei definiert werden. Eine standardmäßig verwendete *Ratio* wäre beispielsweise die durchschnittliche Anzahl von CPU-Takte pro Instruktion.

Die gesammelten Daten können anschließend für jeden aktiven Prozess oder Thread, bis hin zur *disassembliert*en Ansicht, aufgeschlüsselt werden.

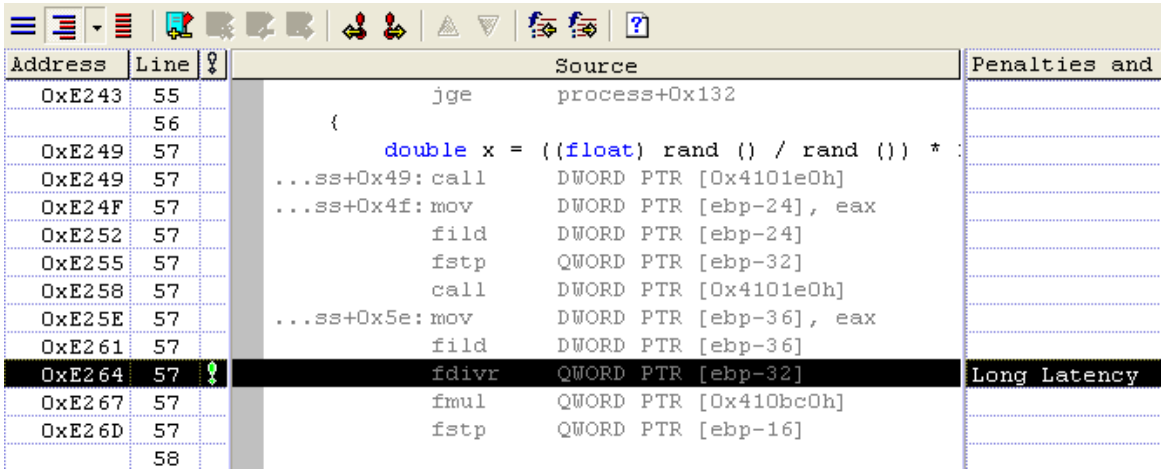

Abb. 29: disassembly-view

[12] Gemeint ist damit beispielsweise ineffiziente Speichernutzung wie zum Beispiel *false sharing*

5.2 Profiling Messwerte

Für die Berechnung der Ratios auf der nächsten Seite wurden folgende **Events** aufgezeichnet

- CPU_CLK_UNHALTED.THREAD
  Zählt die Anzahl der Clock-Cycles für einen Thread, während der sich der ausführende CPU-Kern nicht in einem *halt*-Status befindet nachdem eine hlt-Instruktion ausgeführt wurde. Im Ergebnis werden schließlich die Clock-Cycles aller ausgeführten Threads aufaddiert.

  Bei aktiviertem Hyper-Threading kann das mitunter zu sehr verwirrenden Ergebnissen führen, da die Cycles Threadbasiert gezählt werden. In der Folge werden für jeden physikalischen CPU-Kern, auf dem gleichzeitig 2 Threads ausgeführt werden, die Cycles doppelt gezählt. Bei der Auswertung der Profiling-Ergebnisse muss daher beachtet werden, welche beobachteten Module parallelisiert berechnet werden und auf welchen CPU-Kernen die entsprechenden Threads laufen.

- INST_RETIRED.ANY
  Die Anzahl aller Instruktionen, die während des Samplings ausgeführt wurden. Bei einer Instruktion, die aus mehreren *Micro-Operation*en besteht wird nur die Ausführung der letzten Micro-Operation gezählt.

- UOPS_RETIRED.ANY Zählt alle ausgeführten Micro-Operationen. Micro-Operationen sind decodierte Instruktionen, die von dem Prozessor ohne weitere Aufbereitung direkt ausgeführt werden können. Die meisten Instruktionen werden in ein bis zwei Micro-Operationen dekodiert. In Ausnahmefällen, wie Wiederholungsanweisungen –>(Loop Stream Detection) oder komplexen Floating-Point-Instruktionen, können auch längere Sequenzen auftreten.

- L1D_CACHE_LD.I_STATE L1-Cache Loads für Daten (Werte für Variablen oder Ähnliches), die nicht im L1-Cache gefunden werden konnten. (Invalid-State)

- L1D_CACHE_LD.MESI Alle L1-Cache Loads für Daten, egal ob Treffer oder nicht. (Modified[13] + Exclusive[14] + Shared[15] + Invalid-State[16])

- L1I.CYCLES_STALLED Zählt die Takte, in denen die CPU darauf warten muss den nächsten Befehl ausführen zu können, weil Instruktionen nicht direkt aus dem L1 Instruktions-Cache gelesen werden konnten.

[13] Die Daten befinden sich nur im angefragten L1-Cache und wurden verändert. Sie müssen noch mit dem Arbeitsspeicher abgeglichen werden (können aber problemlos weiter bearbeitet werden)
[14] Die Daten befinden sich nur im angefragten L1-Cache und ensprechen den Daten im Arbeitsspeicher
[15] Die Daten befinden sich eventuell auch noch in einem anderen Cache und sind mit dem Arbeitsspeicher synchron
[16] kein Treffer im L1-Cache

- UOPS_RETIRED.CORE_STALL_CYCLES Takte, in denen keine bereits an eine Execution-Unit zugeteilte Micro-Operations ausgeführt wurde.

- UOPS_RETIRED.CORE_ACTIVE_CYCLES Takte, in denen mindestens eine bereits an eine Execution-Unit zugeteilte Micro-Operation ausgeführt wurde.

Die **Ratios** in den nachstehenden Analysen wurden aus folgenden Berechnungen gebildet:

- **CPI - Clock Cycles per Instruction retired**: Anzahl der Takte, die im Mittel für die Abarbeitung einer Instruktion benötigt wird.

$$CPI = \frac{INST\_RETIRED.ANY}{CPU\_CLK\_UNHALTED.THREAD}$$

- **CPu - Clock Cycles per micro-op retired**: Anzahl der Takte für die Abarbeitung einer dekodierten micro-Operation.

$$CPu = \frac{UOPS\_RETIRED.ANY}{CPU\_CLK\_UNHALTED.THREAD}$$

- **FP% - Floating Point Unit utilization**: Anteil von floating-point-instructions.

$$FP\% = \frac{INST\_RETIRED.X87}{INST\_RETIRED.ANY}$$

- **L1DMISS - L1 Data Cache Miss**: Anteil der Zugriffe auf Daten, die nicht im L1-Cache gefunden werden konnten.

$$L1DMISS = \frac{L1D\_CACHE\_LD.I\_STATE}{L1D\_CACHE\_LD.MESI}$$

- **L1ISTALLS - L1 Instruction Cache Stall Cycles**: Anteil von stall-cycles, bei der Wartezeit auf Instruktionen, die sich nicht im L1-Cache befanden.

$$L1ISTALLS = \frac{L1I.CYCLES\_STALLED}{CPU\_CLK\_UNHALTED.THREAD}$$

- **EXEC_STALLS - Execution Stall Cycles**: Anzahl der stall-cycles in der micro-ops auf die Ausführung warten.

$$EXEC\_STALLS = \frac{CORE\_STALL\_CYCLES}{CORE\_STALL\_CYCLES + CORE\_ACTIVE\_CYCLES}$$

5.3 Thread Profiler

Der Thread Profiler bietet eine komfortable Möglichkeit, parallelisierte Anwendungen zu *debuggen* und Thread Synchronisierungen auf ihre Ausführungsgeschwindigkeit hin zu optimieren. Das Beispiel in Abbildung 30 zeigt ein einfaches Producer/Consumer Programm. Die dunkelgrünen Balken kennzeichnen einen aktiven Thread, während sich die Threads in den hellgrünen Bereichen in einem wait-state befinden.

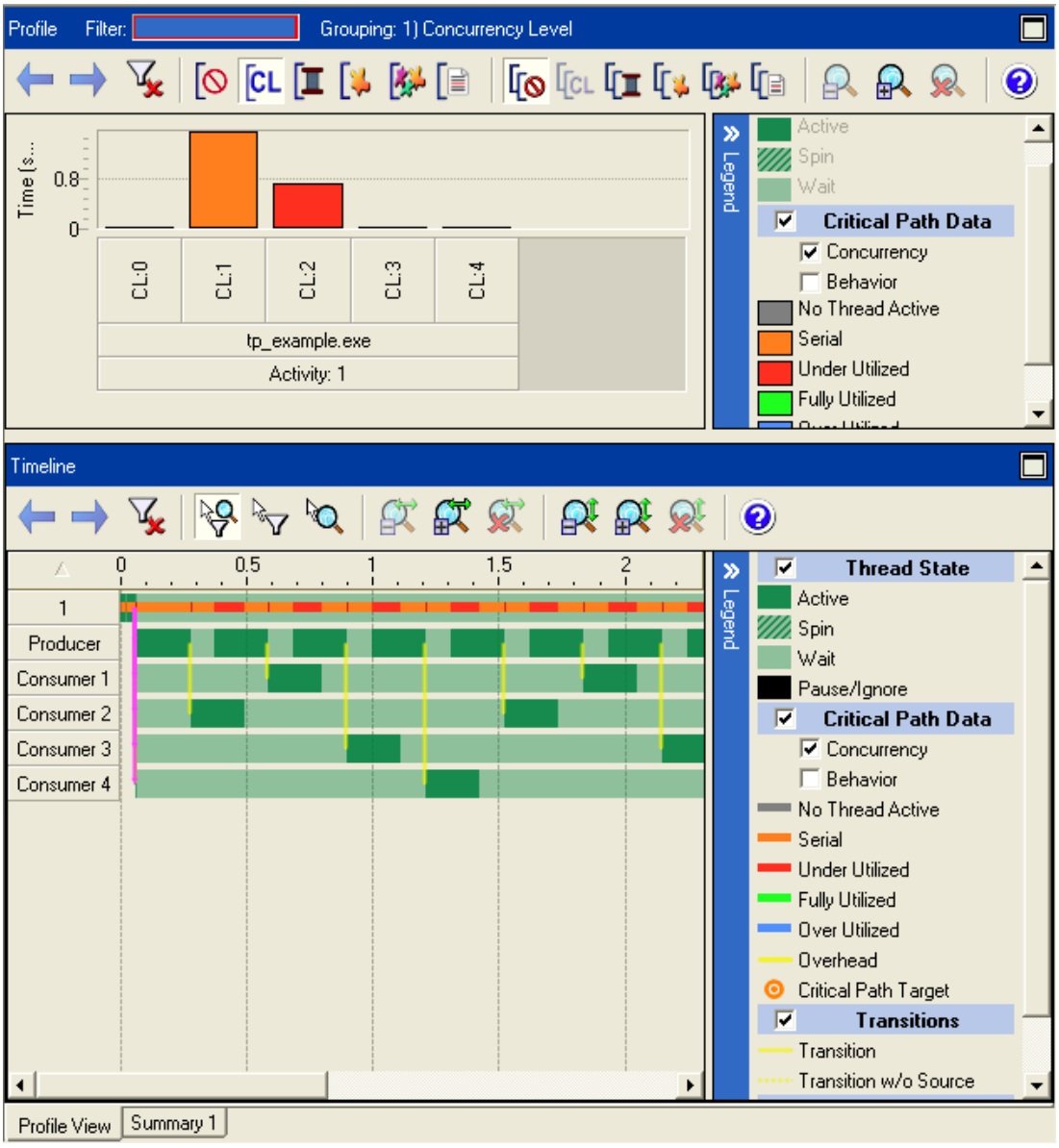

Abb. 30: Ausgabe des Thread-Profilers für das Producer/Consumer Beispielprogramm

Die Threads führen for-Schleifen aus, die eingefügt wurden, um die Thread-Aktivität in der Ausgabe besser sichtbar zu machen. Im Producer-Thread wurde zusätzlich noch eine 'Sleep'-Anweisung eingefügt, die für die regelmäßigen Unterbrechungen sorgt, während der wait-state bei den Consumer-Threads durch das Warten auf die Semephore ausgelöst wird. In der Ausgabe ist gut zu ekennen, dass der Scheduler die Threads nicht zufällig bedient, sondern sie immer wieder in der gleichen Reihenfolge zur Ausführung kommen lässt.

```
//Producer-Threads
while ( true )
    {

    for ( int32 i = 0; i < 100000000; i++ ) {}
    sema->release ();
    Sleep ( 100 );
}

//Consumer-Threads
while ( true )
{
    sema->acquire ();
    for ( int32 i = 0; i < 10000000; i++ ) {}
}
```

6 Vergleich von Hyper-Threading auf einem Xeon-Prozessor mit dem Core i7

6.1 Testsystem

CPU:	Intel Core i7 Extreme 965
Chipsatz:	X58
BIOS-Version:	SOX5810J.86A.2786.2008.1112.1500
Speicher:	4x 1024MB Kingston *DDR*3, 667MHz
Betriebssystem:	Windows XP SP2

6.2 Performance-Analyse anhand von Prototypen

Für den Core i7 waren keine Informationen zu finden, die Aufschluss darüber gegeben hätten, welchen Einfluss Hyper-Threading auf die Ausführungsgeschwindigkeit von parallelisiertem Code hat. Lediglich für den ersten 2002 eingeführten Xeon Processor mit Hyper-Threading-Technologie war ein Artikel [7] zu finden, der sich mit der Ausführungsgeschwindigkeit verschiedener *Benchmark*s beschäftigt. Einige dieser Benchmarks wurden nachgestellt um einen Überblick über die potentiellen Stärken und Schwächen des Core i7 zu erhalten.

Alle Tests wurden jeweils mit aktiviertem und deaktiviertem Hyper-Threading durchgeführt. Für die Tests mit aktiviertem Hyper-Threading wurde die Berechnung auf 8 Threads verteilt, die per CPU-Affinität jeweils an einen logischen Kern gebunden wurden. Bei deaktiviertem HT wurde die gleiche Arbeit auf 4 Threads verteilt, die Anzahl der Wiederhohlungen der Berechnung wurden dementsprechend verdoppelt.

6.3 Messergebnisse von Xeon (2002) und Core i7

Tab. 1: Messergebnisse Xeon [7]

Code	Benchmark	CPI	CPuops	FP%	HT Speed-up
M1	int_mem (load=32)	1.99	0.94	0.0 %	1.08
M2	int_mem (load=4)	6.91	3.61	0.0 %	1.36
M3	dbl_mem (load=32)	1.81	1.47	23.2 %	1.90
M4	int_dbl (load=32)	3.72	1.63	9.8 %	1.76

Tab. 2: Messergebnisse Core i7

Code	Benchmark	CPI	CPuops	FP%	L1DMISS	L1ISTALLS	exec-time (ms)
M1	int_mem (load=32)	0.68	0.56	0.00%	1%	0%	2073
M1 HT	int_mem (load=32)	1.18	1.08	0.00%	1%	0%	2009
HT Speedup		**1.15**	**1.04**				**1.03**
M2	int_mem (load=4)	0.73	0.61	0%	8%	0%	2184
M2 HT	int_mem (load=4)	1.27	1.14	0%	5%	0%	2065
HT Speedup		**1.15**	**1.07**				**1.06**
M3	dbl_mem (load=32)	0.69	0.53	14%	1%	0.00%	2053
M3 HT	dbl_mem (load=32)	1.19	1.01	14%	1%	0.00%	1961
HT Speedup		**1.16**	**1.05**				**1.05**
M4	int_dbl	0.76	0.79	4%	0%	0%	2062
M4 HT	int_dbl	1.42	1.44	5%	0%	0%	1594
HT Speedup		**1.07**	**1.10**				**1.29**

6.3.1 Erläuterung zu den Messergebnissen

In Tabelle 2 werden jeweils für CPI, CPuops und Ausführungsgeschwindigkeit (exec-time) Verhältnisse errechnet, an denen erkennbar wird, wieviel Geschwindigkeitsgewinn der jeweilige Benchmark durch die Aktivierung von HT erfährt. Ausschlaggebend ist letzendlich der Wert für die Ausführungszeit, der angibt, wie lange die Berechnung insgesamt benötigt hat. Die Werte für CPI (Cycles per Instruction) und CPuops (Cycles per Micro-Instruction) lassen eine Aussage darüber zu, mit welcher Effizienz der Code ausgeführt wurde. Die hervorgehobenen Werte für HT-Speedup geben an, in welchem Verhältnis der Code mit aktiviertem HT schneller ausgeführt wurde.

Für Hyper-Threading ist hierbei eine Besonderheit für die Errechnung des Geschwindigkeitsverhältnisses zu beachten. Während sich der Wert für die Ausführungszeit leicht ersichtlich mit

$$HTSpeedup_{execTime} = \frac{exectime_{HTdisabled}}{exectime_{HTenabled}}$$

errechnen lässt, werden die Werte für CPI und CPuops mit der Formel

$$HTSpeedup_{CPI} = \frac{CPI_{HTdisabled} * 2}{CPI_{HTenabled}}$$

beziehungsweise

$$HTSpeedup_{CPuops} = \frac{CPuops_{HTdisabled} * 2}{CPuops_{HTenabled}}$$

errechnet. Grund für den Multiplikator bei deaktiviertem Hyper-Threading ist die unterschiedliche Zählweise der Takte mit aktiviertem bzw. deaktiviertem Hyper-Threading. Bei der Berechnung der Verhältnisse für CPI und CPu fließt jeweils der Wert für CPU_CLK_UNHALTED.THREAD aus Kapitel 5.1 mit in die Berechnung ein. Dieser Wert bezieht sich, wie der Name bereits andeutet, auf die Anzahl der Clock Ticks für einen Thread. Bei aktiviertem Hyper-Threading hat das zur Folge, dass dadurch, dass 2 Threads auf einem physikalischen CPU-Kern gleichzeitig ausgeführt werden, die Clock Ticks doppelt gezählt werden.

6.4 Auswertung

6.4.1 int_mem (M1, M2)

```
for ( int32 i = 0; i < repeat; ++i ) // A
{

    j = index[ i % bufferSize ];

    for ( int32 k = 0; k < load; ++k ) // B
    {
        buffer[ j ][ 0 ] += input;
        buffer[ j ][ 1 ] += input;
        buffer[ j ][ 2 ] += input;
        buffer[ j ][ 3 ] += input;

        buffer[ j ][ 4 ] += input;
        buffer[ j ][ 5 ] += input;
        buffer[ j ][ 6 ] += input;
        buffer[ j ][ 7 ] += input;
    }

}
```

Der **int_mem** Benchmark ist darauf ausgelegt, den Effekt auf die Ausführungsgeschwindigkeit bei unterschiedlicher Auslastung der ALU zu untersuchen. Dafür wird mit zufällig verteilten Indizies wiederholt auf eine 32Mb große Buffer-Variable zugegriffen. Die Buffer-Variable wurde so groß angelegt, um zu verhindern, dass der Speicherbereich vollständig im L3-Cache abgebildet werden kann.

Durch die zufällig verteilten Zugriffe wird erreicht, dass die Wahrscheinlichkeit beim ersten Zugriff auf die Buffer-Variable in Schleife B im L1-Cache fündig zu werden, sehr gering ist. In der Folge sind einige *Stall-Cycles* notwendig, um auf den langsameren Hauptspeicherzugriff zu warten. Bei den nachfolgenden Zugriffen auf die Buffer-Variable über die Indizes 0-7 ist hingegen sichergestellt, dass der Speicherbereich bereits geladen wurde, da die 'Gesamtgröße' eines Zugriffs mit 8*32 Bit (32Byte) unterhalb der Größe eines *Cache-Block*s liegt, der beim ersten Zugriff geladen wurde.

Die Durchführung des Tests erfolgte anschließend in 2 unterschiedlichen Varianten mit den Werten 32 und 4 (M1 und M2) für die Variable 'load'. Durch den höheren Wert für M1 wird erreicht, dass der Prozessor wiederholt auf den selben Speicherbereich zugreifen kann und nur selten Pausen einlegen muss, in denen er auf den Hauptspeicher warten muss. M2 hingegen wird im L1-Cache nicht so oft fündig und muss verhältnismäßig viele Stall-Cycles einlegen.

In Tabelle 2 ist zu sehen, dass sich die L1-Cache-Miss-Rate (L1DMISS) bei einem kleineren Wert für 'load' merklich erhöht. In der Folge erhöhen sich bei M2 die Stall-Cycles und die Möglichkeit mit Hyper-Threading die Auslastung der execution-units zu erhöhen führt zu einer

Reduzierung der Ausführungsgeschwindigkeit.

Leider gibt es in dem Artikel[7] aus dem Tabelle 7.2.1 stammt, keine Informationen über die entsprechenden Werte für den Xeon-Prozessor, so dass hier kein direkter Vergleich darüber anstellt werden konnte, ob eine Optimierung der Zugriffe auf den L1-Cache beim Core i7 für den verringerten Geschwindigkeitsgewinn verantwortlich ist.

6.4.2 dbl_mem (M3)

dbl_mem entspricht dem int_mem-Benchmark, mit dem Unterschied, dass der Datentyp für die Buffer-Variable hier von int auf double geändert wurde und die innere Schleife entsprechend verkürzt wurde um bei der Block-Größe von 32Byte zu bleiben. Daraus resultierend erhöht sich das Optimierungspotenzial für HT noch weiter; Durch die Mischung von integer und floating-point Instruktionen können die unterschiedlichen Execution-Units besonders Effektiv genutzt werden.

6.4.3 int_dbl (M4)

```
for ( int i = 0; i< npoints; i++ )
{
    double x, y;

    x = (((float) rand ()) / RAND_MAX * 2) - 1;
    y = (((float) rand ()) / RAND_MAX * 2) - 1;

    if ( sqrt (x*x + y*y) <= 1 )
    {
        inner++;
    }
}
```

Bei diesem Benchmark wird eine Annährung an Pi mit Hilfe eines Monte-Carlo Algorithmus errechnet. Für x und y werden zufällige Werte zwischen -1 und 1 ermittelt. Mit der Formel

$$\sqrt{x^2+y^2} <= 1$$

wird anschließend errechnet, ob sich der Punkt innerhalb oder ausserhalb eines Kreises mit dem Radius 1 befindet. Teilt man anschließend die Zahl der Treffer durch die Gesamtzahl der errechneten Punkte erhält man damit eine Annährung an Pi/4.

6.5 Vergleich von Hyper-Threading in der Nehalem- und NetBurst-Architektur

Subsystem	Nehalem (Core i7)	NetBurst (Xeon, P4)
Ports, Funcrional-Units	Drei Ports für FPU, ALU und SIMD-Instruktionen (Port 0, 1, 5)	Alle ALU, FPU und SIMD-Instruktionen auf demselben Port (Port 1)
Buffer	Mehr Einträge in *Reorder Buffer* und *Reservation Station*, geringere Pipeline-Tiefe	schlechtere Balance zwischen Pipelinetiefe und Anzahl der Buffereinträge
Sprungvorhersage und nicht-zusammenhängende Speicherzugriffe	Zuverlässigere Sprungvorhersage mit sofortiger Rücknahme der Instruktion bei Fehlvorhersagen, weniger vollständige Pipeline-Leerungen	Mehr Pipeline-Hazards mit anschließender Leerung der Pipeline
Cache Hierarchy	Größer und effizinter	Größere Anzahl Micro-Architektur bedingter Schwachstellen
Speicher und Speicherzugriff	*NUMA*, drei Kanäle pro Socket zu *DDR*3-Speicher, bis zu 32GB/s pro Socket	*SMP*, *FSB* oder dual FSB, bis zu 12.8GB/s pro FSB

6.6 Zusammenfassung

Auffällig bei dem Vergleich der Messergebnisse zwischen Core i7 und Xeon ist, dass alle Benchmarks auf dem Core i7 wesentlich effizienter ausgeführt werden und der mit Hyper-Threading zu erreichende Geschwindigkeitsgewinn dementsprechend niedriger ausfällt.

Offensichtlich ist das Optimierungspotential durch Hyper-Threading bei der Nehalem-Architektur gegenüber der NetBurst-Architektur deutlich zurückgegangen. Während die älteren Prozessoren dazu neigten rechenintensive CPU-Lasten mit relativ hohen CPI-Werten zu bearbeiten, wird dieselbe Berechnung auf einem Nehalem-Prozessor per se schon wesentlich effizienter ausgeführt. Von Intel selbst gibt es zu dem Thema recht unterschiedliche Aussagen.

Während in [6] (aus dem Jahr 2002) hohe Geschwindigkeitsgewinne mit Hyper-Threading demonstriert werden, wird in Kapitel 8.9.1 in [14] (aus dem Jahr 2009) genau beschrieben warum mit Hyper-Threading damals aufgrund des schlechteren Desings der Micro-Architektur weniger Optimierungspotential vorhanden gewesen sei.

Die Messergebnisse der hier nachgestellten Benchmarks zeigen hingegen, dass die Benchmarks, die aufgrund ineffizienter Speicherzugriffe auf einem älteren Xeon-Prozessor noch besonders langsam ausgeführt wurden, noch recht gut von Hyper-Threading profitiert haben, während die Berechnungen auf einem aktuellen Nehalem-Prozessor auch ohne Hyper-Threading schon so schnell ausgeführt werden, dass mit Hyper-Threading nicht mehr viel zu gewinnen ist.

Grund zur Hoffnung gibt hier lediglich der vierte Benchmark der bei einer ausgewogenen Mischung von Integer- und Floating-Point-Operationen mit aktiviertem Hyper-Threading immerhin fast 30% schneller ausgeführt wird.

7 Auswirkungen von Hyper-Threading auf den derzeitigen Entwicklungsstand von Nuendo 5

7.1 Performance-Analyse unter Nuendo 5

7.1.1 Demo-Projekte (worst-case)

Die Demo-Projekte wurden mit der Zielsetzung entwickelt zu untersuchen, ob niedrig priorisierte Threads, die für Festplattenzugriffe oder die *GUI* arbeiten, störenden Einfluss auf die hoch priorisierten Threads für das audio-Processing haben.

Zur Abschätzung des Geschwindigkeitsvor- bzw. -nachteils wurden alle Messungen jeweils einmal mit aktiviertem und einmal mit deaktiviertem Hyper-Threading durchgeführt.

Um über einen längeren Zeitraum kontinuierlich für viele Festplattenzugriffe und 'unwichtige' Berechnungen zu sorgen, wurden in die Projekte mehrere 8Gb große wave-Dateien eingefügt. Im Projekt-Fenster wird dafür eine Vorschau der Wellenform angezeigt, dessen Berechnung die gewünschten Zugriffe erzeugt.

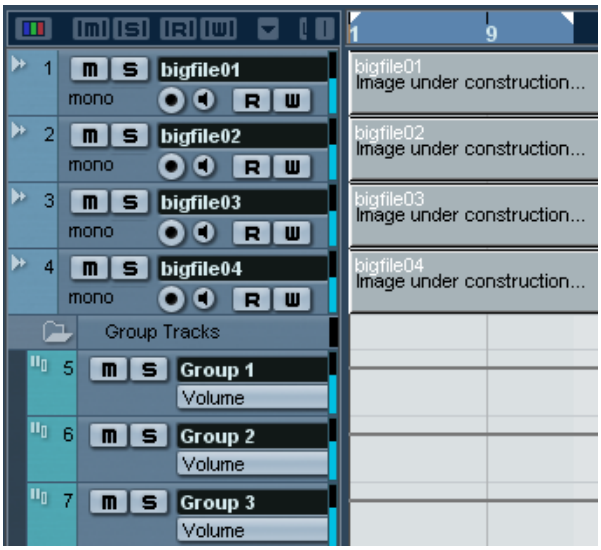

Abb. 31: Aufbau des 'worst-case' Demo-Projektes

Für die Messung wurden insgesamt 8 unterschiedliche Projekte verwendet. Ein Projekt war das Demo-Projekt zu Cubase 4 (blofelds-return.cpr), das darauf ausgelegt ist, dem alltäglichen Gebrauch möglichst nahe zu kommen. Die anderen Projekte haben jeweils 1, 4, 8 oder 32 identische Audio-Spuren. Bei den Projekten mit bis zu 8 Kanälen wurde jeweils einmal mit und ohne Einbindung der wave-Dateien gemessen. Um die Dauer des Processings zu erhöhen und gut

reproduzierbare Werte zu erhalten wurden die einzelnen Kanäle mit rechenintensiven Effekten belegt.

7.1.2 Demo-Projekt (best-case)

Um eine Aussage darüber treffen zu können, welchen Geschwindigkeitsvorteil Hyper-Threading im besten Fall bringen kann, wurde ein weiters Projekt erstellt, das darauf ausgelegt wurde möglichst gute Optimierungsmöglichkeiten für HT zu bieten. Das Projekt wurde zu diesem Zweck mit 8 Audio-Kanälen erstellt und mit so vielen Effekten versehen, dass die CPU-Auslastung bei deaktiviertem HT bei ca. 80-90% lag. Zwischen den einzelnen Kanälen bestanden keine Abhängigkeiten, so dass jeder Kanal seperat berechnet werden konnte. Durch die hohe CPU-Auslastung wurde gewährleistet, dass niedrig priorisierte Threads anderer Prozesse wenig Zeit zum Stören hatten und der Zeitraum in dem das Audio-Processing mit HT optimiert werden konnte, möglichst lang gehalten wurde.

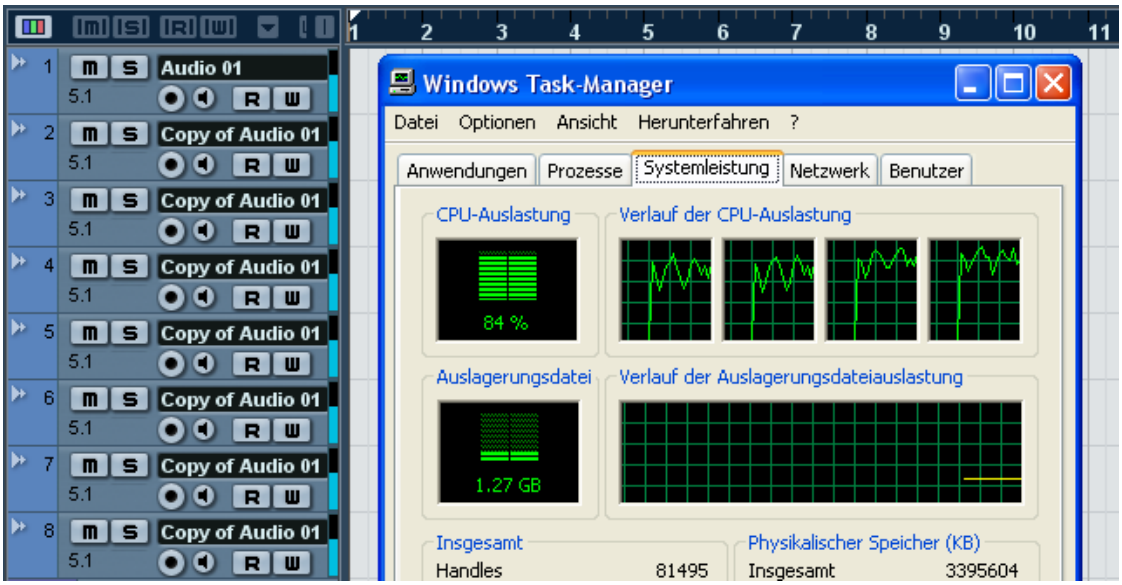

Abb. 32: CPU-Auslastung des 'best-case' Demo-Projektes mit deaktiviertem Hyper-Threading

7.1.3 Processing-Geschwindigkeit

Gemessen wurde die Zeit, die benötigt wird, um einen kompletten *ASIO*-Block zu berechnen und damit maßgeblich für die minimale Latenz des Systems ist. Für die Messung wurde der intern verwendete 'PerformanceExport' um die Auswertung der Messergebnisse hinsichlich der Berechnungszeit für ASIO-Blöcke ergänzt. Als Messergebnisse stehen die durchschnittliche Berechnungszeit, sowie die längste gemessene Zeit über die Dauer der Messung (1000 Blöcke) zur Verfügung. Jede Messung wurde 3 mal durchgeführt, deren Mittelwerte flossen schließlich in die Auswertung ein.

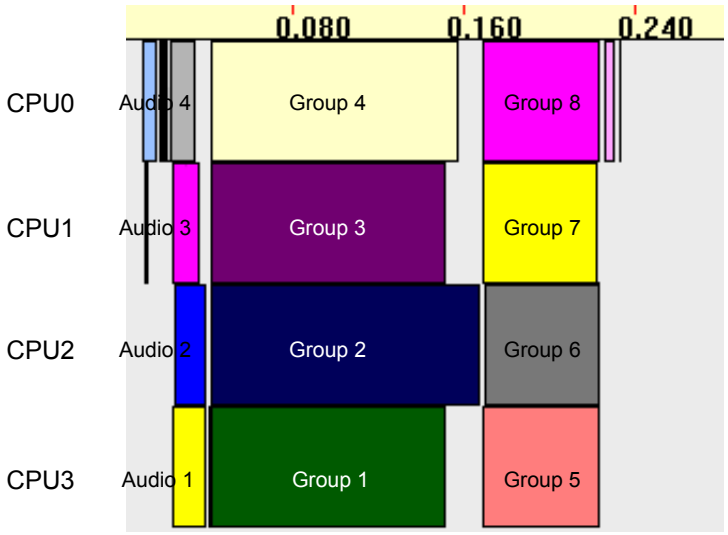

Abb. 33: Processing des Demo-Projektes aus Abbildung 31 bei deaktiviertem Hyper-Threading. Die Länge eines ASIO-Blocks entspricht der Gesamtlänge aller Balken.

Um eventuell noch nicht genau bekannte Einflüsse des scheduling-Verhaltens von Windows Vista auszuschließen, wurden die Messungen unter Windows XP durchgeführt. Die ASIO-Puffergröße wurde auf 128 Samples eingestellt, was bei einer Sampling-Rate von 44,1kHz einer theoretischen Latenz von 2,9ms entspricht.

7.1.4 Messergebnisse Processing-Geschwindigkeit

Für die worst-case Projekte haben die Messungen unter Nuendo ergeben, dass der erhoffte Performance-Gewinn beim Audio-Processing mit ca. 6% noch recht gering ausfällt, im ungünstigsten Fall sind sogar Einbußen bis zu 25% möglich. Die Differenz von fast 35% zwischen dem schlechtesten und dem besten Fall zeigt, dass hier noch ein hohes Optimierungspotenzial vorhanden ist.

Mean Processing Time	**4 core**	**8 core**	HT Speed-up
blofelds_return (worst-case)	0.04 ms	0.05 ms	- 25.0%
4 Channel (worst-case)	0.35 ms	0.42 ms	- 20.0%
8 Channel (worst-case)	0.66 ms	0.62 ms	6.1%
8 Channel (best-case)	4.02 ms	3.67 ms	9.5%

Während sich über die Ursache der Performanceeinbuße bei blofelds_return an dieser Stelle noch keine genaue Aussage treffen lässt, zeigt sich schon recht deutlich, dass die logischen CPU-Kerne, die bei dem Projekt mit nur 4 Spuren frei bleiben, offensichtlich einen störenden Einfluss haben. Diese Messergebnisse waren der Anlass, die Projekte mit 4 und 8 Kanälen sowie blofelds_return einem detaillierten Profiling zu unterziehen.

7.1.5 VTune-Messergebnisse[17]

Die Cycles pro Micro-Operation und Cycles pro Instruktion geben einen Einblick, mit welcher Effizienz der verwendete Code ausgeführt wird. Da die Micro-Operationen aus dekodierten Instruktionen generiert werden und einige Instruktionen zu mehreren Micro-Ops zerlegt werden, muss der Wert für die CPu[18] immer kleiner sein als der CPI Wert.

Cycles per Micro-Op	**4 core**	**8 core**	HT Speed-up
blofelds_return (worst-case)	0.80	1.05	- 23.4%
4 Channel (worst-case)	0.48	0.57	- 15.8%
8 Channel (worst-case)	0.47	0.45	4.4%
8 Channel (best-case)	0.47	0.42	11.9%

Cycles per Instruction	**4 core**	**8 core**	HT Speed-up
blofelds_return (worst-case)	1.26	1.58	- 20.2%
4 Channel (worst-case)	0.55	0.61	- 9.8%
8 Channel (worst-case)	0.57	0.51	11.8%
8 Channel (best-case)	0.60	0.50	20.0%

Auf den ersten Blick mag es verwunderlich erscheinen, dass mehr als eine Instruktion pro Taktzyklus ausgeführt werden kann, hier greift aber schon bei einem einzelnen Thread die prozessorinterne Optimierung, die dafür sorgt, dass alle Instruction-Units möglichst gut ausgelastet werden. Bei einem hohen Anteil von floating-point-Instruktionen würde der CPu-Wert bei optimaler Ausnutzung aller drei FPUs also bei 1/3 liegen. In der Praxis kommen immer auch noch einige Load und Store Instruktionen dazu, so dass dieser Wert praktisch nicht erreicht werden kann.

Die CPu-Werte von 0.47 beziehungsweise 0.42 zeigen also, dass der hier ausgeführte Code schon außerordentlich hoch optimiert ist.

Bei den gemessenen Werten in den folgenden Tabellen ist zu beachten, dass die Werte, die für blofelds_return gemessen wurden, mit Vorsicht zu genießen sind. Im Gegensatz zu den 'synthetischen' Projekten, die darauf ausgelegt sind saubere Messwerte zu produzieren, ist blofelds_return ein Projekt, dass sich an der Nutzung im realen Umfeld orientiert. Die Auswahl der zu untersuchenden Module ist hier längst nicht so eindeutig wie bei den anderen Projekten.[19] Ich habe sie hier der Vollständigkeit halber trotzdem mit aufgeführt.

Bei dem Vergleich der 'synthetischen' Projekte fällt auf, dass die Werte für Execution-Stalls und Instruction-Cache Stalls zurück gehen, während sich Data-Cache Miss Rate erhöht. Ins-

[17] Eine detailliertere Erklärung wie die verwendeten Messwerte ermittelt wurden ist in Kapitel 5.1 zu finden.

[18] Nicht Central Processing Unit, sondern Cycles per Micro-Operation

[19] bei den 4- und 8 Channel Projekten lagen ca. 95% der erzeugten Last bei den Modulen nuend5.exe und vst_pluginset.vst3, bei blofelds_return ist die Last auf etliche Module verteilt, die sich nicht immer sicher zuordnen lassen.

Execution Stalls	4 core	8 core
blofelds_return (worst-case)	35.89%	13.10%
4 Channel (worst-case)	9.96%	9.82%
8 Channel (worst-case)	9.89%	6.10%
8 Channel (best-case)	9.63%	3.38%

L1 Instruction-Cache Stalls	4 core	8 core
blofelds_return (worst-case)	8.97%	4.98%
4 Channel (worst-case)	1.90%	1.29%
8 Channel (worst-case)	1.20%	1.09%
8 Channel (best-case)	0.61%	0.39%

L1 Data-Cache Miss Rate	4 core	8 core
blofelds_return (worst-case)	21.54%	21.30%
4 Channel (worst-case)	1.11%	1.40%
8 Channel (worst-case)	1.40%	4.66%
8 Channel (best-case)	0.85%	4.53%

besondere im Best-Case Fall fällt der relative Unterschied sehr deutlich aus. Eine mögliche Erklärung dafür ist, dass bei einer größeren Erhöhung der Ausführungsgeschwindigkeit auch die Wahrscheinlichkeit für L1 Data-Cache Misses stärker ansteigt.

7.1.6 Thread Profiler-Messergebnisse

Die Anlayse mit dem Thread-Profiler wurde anchließend ausgewertet, um einen genaueres Bild davon gewinnen zu können, aus welchen Grund sich die Ausführungsgeschwindigkeit mit aktiviertem HT in manchen Fällen verschlechtert. Es konnte schließlich eindeutig nachgewiesen werden, dass Threads mit unterschiedlichen Prioritäten sich bei aktiviertem HT nicht genauso verhalten wie bei deaktiviertem HT. In Abbildung 35 ist die Verarbeitung von 2 ASIO-Böcken aus dem worst_case-Projekt bei deaktiviertem Hyper-Threading zu sehen.

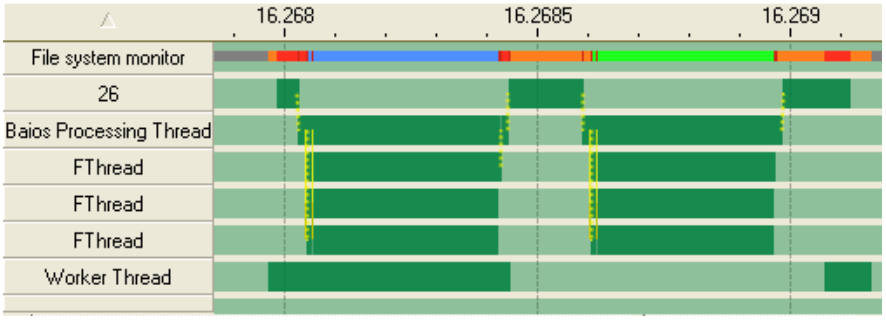

Abb. 34: Processing von 2 ASIO-Blöcken aus dem worst_case-Projekt bei deaktiviertem Hyper-Threading.

Der mit ’26’ bezeichnete Thread stammt vom ASIO-Treiber und signalisiert an den Baios Processing Thread, dass der Sample Buffer bereit zum Beschreiben ist. Der Baios Thread triggert anschließend drei weitere Audio-Threads mit deren Hilfe die zu verarbeitenden Process-Units abgearbeitet werden (2.4). Der Worker Thread arbeitet als niedrig priorisierter Thread an der Berechnung der Vorschaugrafik für eine große Wave-Datei. Im Vergleich mit dem Worker Thread in Abbildung 34, auf dem der gleiche Vorgang bei aktiviertem HT gezeigt wird, ist zu erkennen, dass der Worker Thread bei deaktiviertem HT in der Ausführung wie gewünscht verzögert wird, während er bei aktiviertem HT fast ungebremst weiter arbeitet.

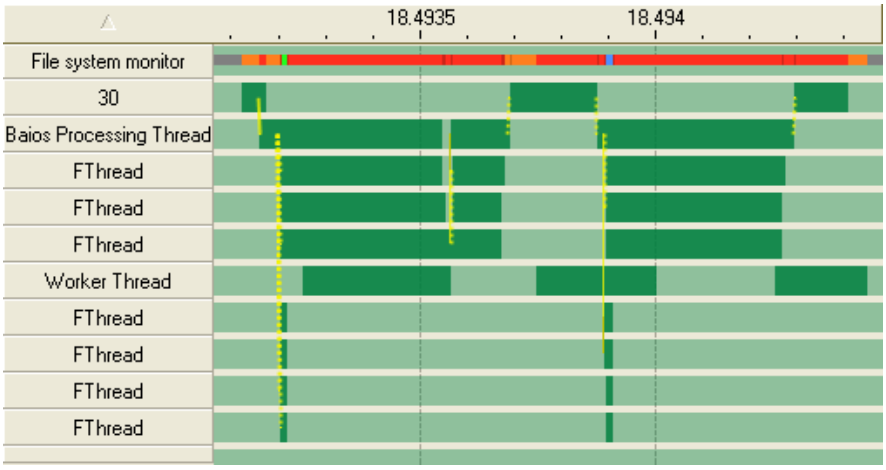

Abb. 35: Processing von 2 ASIO-Blöcken aus dem worst_case-Projekt bei aktiviertem Hyper-Threading.

Über eine größere Anzahl von ASIO-Blöcken betrachtet, ist deutlich zu sehen, dass dieser Effekt nicht nur zufällig bei einzelnen Blöcken auftritt, sondern sich systematisch über das gesamte Audio-Processing fortsetzt. (Abbildung 36 und 37)

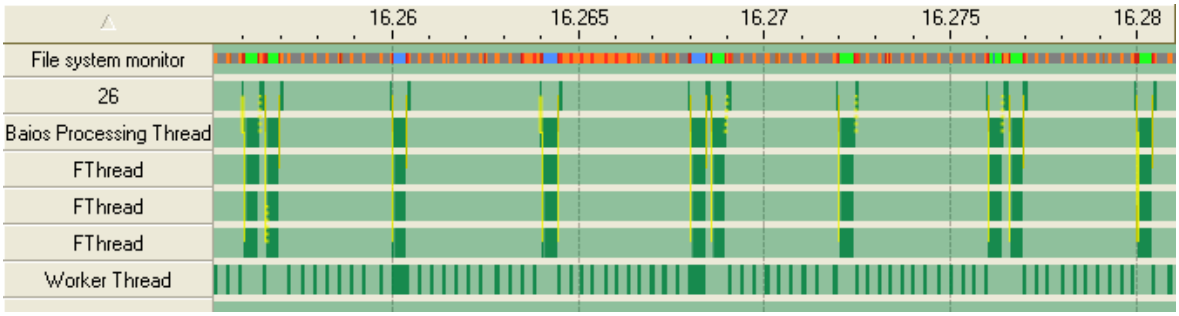

Abb. 36: Processing von 10 ASIO-Blöcken aus dem worst_case-Projekt bei deaktiviertem Hyper-Threading.

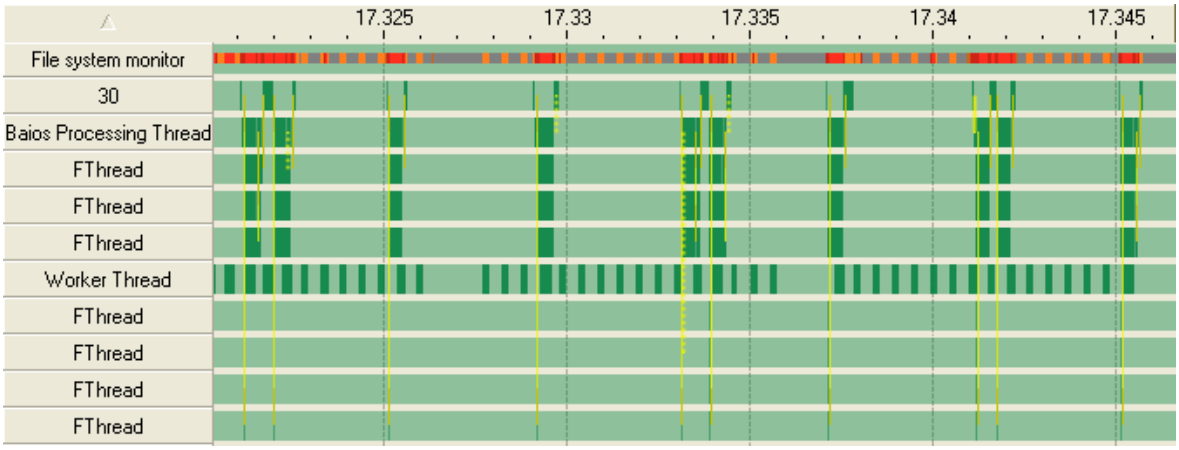

Abb. 37: Processing von 10 ASIO-Blöcken aus dem worst_case-Projekt bei aktiviertem Hyper-Threading.

7.2 Entwicklung von Prototypen zur Untersuchung des Effektes

Um die Auswirkungen auf Nuendo besser nachvollziehen zu können, wurde ein Prototyp entwickelt, der das Audio-Processing nachbildet. Dieser Prototyp diente hauptsächlich dazu, auszuschließen, dass nicht andere Ursachen als die bisher verdächtigten niedrig priorisierten Threads, für die beobachteten Probleme verantwortlich sind. Mit einem weiteren, sehr viel stärker abstrahierten Prototypen, konnte das Problem schließlich genauer eingegrenzt werden. Dieser Prototyp wurde anschließend auch zur Untersuchung der unterschiedlichen Lösungsansätze verwendet.

7.2.1 Nachbildung des Audio-Processings

Um die Schnittstelle zum ASIO-Treiber zu simulieren, wurde ein Thread verwendet, der das nachgebildete Audio-Processing einmal pro Millisekunde angetriggert hat. Vier weitere 'Audio'-Threads simulieren das eigentliche Processing. Jeder der simulierten Audio-Threads verfügt über ein Array von Process-Units, das bei jedem Triggern der Reihe nach abgearbeitet wird. Eine Process-Unit kann dabei entweder eine Reihe von Floating-Point-Berechnungen oder eine *Barriere* enthalten.

Die Threads werden jeweils auf die Barrieren synchronisiert. Bei dem Prototyp wurde ein Pattern verwendet, dass so auch in Nuendo erzeugt werden könnte.

Thread0:	B	P(600)	SB	P(200)	SB	P(400)	SB	P(400)	SB	P(400)	SB	P(400)
Thread1:	B	P(600)	SB	P(150)	SB	P(400)	SB	P(400)	SB	P(400)	SB	P(400)
Thread2:	B	P(600)	SB		SB	P(400)	SB	P(400)	SB	P(400)	SB	P(400)
Thread3:	B	P(600)	SB		SB	P(400)	SB	P(400)	SB	P(400)	SB	P(400)

'B' und 'SB' stehen für Barrier und SpinBarrier, P für eine simulierte Process-Unit. Die Zahlen in der Klammer geben die Anzahl der Schleifendurchläufe und damit die Dauer des Processings an[20].

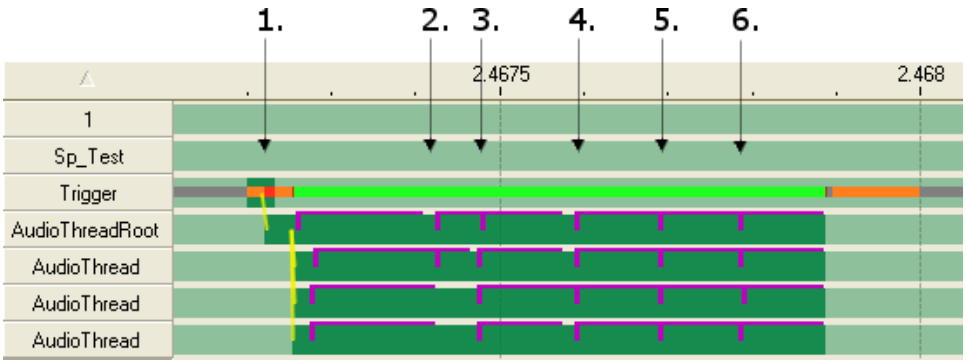

Abb. 38: In 1 wird der Vorgang getriggert während 2 bis 6 die Barrieren zwischen den Processing-Units darstellen. Die lilafarbenen Linien kennzeichnen die Dauer der Berechnung eines Blocks

Thread0 übernimmt eine Sonderfunktion. Er ist als Root-Process implementiert und empfängt das Triggersignal. Erst dann gibt er die erste Barriere frei und löst damit das Processing eines Blocks aus.

Um die störenden Festplattenzugriffe zu simulieren, wurde dieselbe Audio-Datei wie im Worst-Case-Projekt verwendet. Aus der Datei werden jeweils Blöcke von 1600 Byte ausgelesen, zwischendurch werden einige Integer und Floating-Point-Berechnungen durchgeführt. Bei der Ana-

[20] Dieser Unterschied ist für die Darstellung im Threadprofiler wichtig, da nur die Barrieren ohne Spinning in der Ausgabe auch als wait-State zu erkennen sind.

lyse ergibt sich daraus ein ähnliches Muster wie bei dem Thread, der im Worst-Case-Projekt die Berechnung der Wellenform übernimmt.

Die **Messergebnisse** zeigen hier ähnliche Ergebnisse wie die Messung der Processing-Geschwindigkeit in Nuendo.

Mean Processing Time	**4 core**	**8 core**	HT Speed-up
Prototype	0.57	0.59	- 3.4%

Max Processing Time	**4 core**	**8 core**	HT Speed-up
Prototype	0.77	0.86	- 10.5%

7.2.2 Prototyp zur Analyse von Hyper-Threading

Um detailliertere Aussagen über den Einfluss von Hyper-Threading auf die bisherigen Messergebnisse machen zu können, wurde ein Prototyp entwickelt, der es ermöglicht, unterschiedliche Lastsituationen auf unterschiedlichen CPU-Kernen parallel zu simulieren.

Der Overhead, der bei den bisherigen Messungen durch Synchronisiereung von Barrieren verursacht wurde, ist hier außer acht gelassen worden.

Die grundlegende Funktion des Programms besteht darin, ein oder mehrere Threads nahezu zeitgleich zu starten und die Threads dabei nach einem variablen Muster auf den unterschliedlichen logischen CPU-Kernen auszuführen. Für die Ausführung der Threads können den Threads dabei unterschiedliche Arten von Processing-Blöcken mit auf den Weg gegeben zu werden, wobei die Processing-Blöcke sich jeweils in der Auslastung der unterschiedlichen Execution-Units unterscheiden. Für die endgültigen Tests wurde ein Algorithmus verwendet, der für die Berechnung der Korrelation zwischen Audio-Kanälen benutzt wird.

7.2.3 Thread-Profiler Analyse an den Prototypen

Die Messungen an den Prototypen konnten den anfangs geäußerten Verdacht letztendlich bestätigen. Bei der Nachbildung des Audio-Processing tritt bei der Untersuchung mit dem Thread-Profiler ein ähnliches Verhalten auf, wie es auch in den Tests bei der Nuendo-Implementierung zu beobachten ist. Bei deaktiviertem Hyper-Threading wird der niedrig priorisierte Thread während der Ausführung höher priorisierter Threads verzögert. Bei aktiviertem Hyper-Threading läuft er hingegen nahezu ungehindert durch und beeinträchtigt die höher priorisierten Threads dementsprechnd negativ (Abbildung 39 und 40).

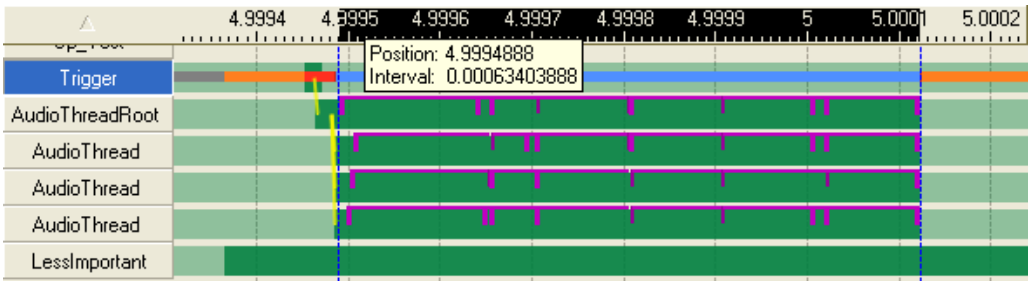

Abb. 39: Simuliertes Audio-Processing bei deaktiviertem Hyper-Threading. An der langen Ausführungszeit des weniger wichtigen Threads ist zu erkennen, dass die Ausführung für die Dauer des simulierten Audio-Processings verzögert wird. Das Processing wird dementsprechend gering beeinträchtigt.

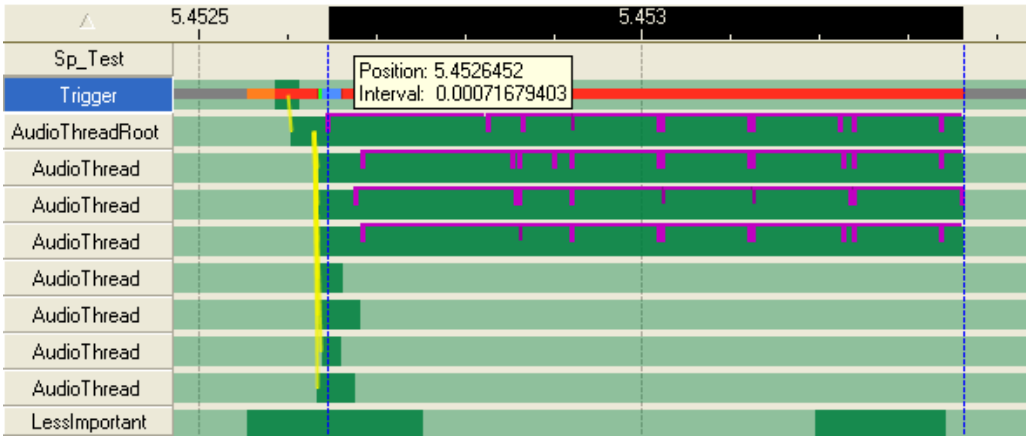

Abb. 40: Simuliertes Audio-Processing bei aktiviertem Hyper-Threading. Der niedrig priorisierte Thread läuft während des simulierten Audio-Processings praktisch uneingeschränkt weiter. Da er sich zwangläufig einen der physikalischen Kerne mit einem der Audio-Threads teilen muss, wird das simulierte Audio-Processing entsprechend ausgebremst.

7.2.4 Test der Thread-Affinitäten

Mit dem Prototyp zur Hyper-Threading-Analyse konnte der Effekt noch wesentlich deutlicher reproduziert werden. Die ersten Messungen dienten ausschließlich der Überprüfung der zugeordneten CPU-Affinitäten. Nach der entsprechenden Dokumentation von Intel bzw. Microsoft[2] sollten die logischen CPUs 0 und 4, sowie 1 und 5 usw. jeweils einer physikalischen CPU zuzuordnen sein. Um das zu überprüfen wurden auf 7 von 8 logischen CPU-Kernen identische Berechnungen ausgeführt, dabei mussten sich 6 Threads jeweils einen physikalischen Kern teilen, der 7. Thread konnte hingegen ungestört arbeiten.

Im Ergebnis (Abbildung 41) ist recht deutlich zu sehen, dass der ungestörte Thread zuverlässig reproduzierbar schneller fertig wird, als die Threads die sich einen physikalischen CPU-Kern teilen müssen. Der Test wurde anschließend für jede weitere Kombination aus logischen CPU-Kernen wiederholt, so dass die Angaben aus [2] vollständig bestätigt werden konnten.

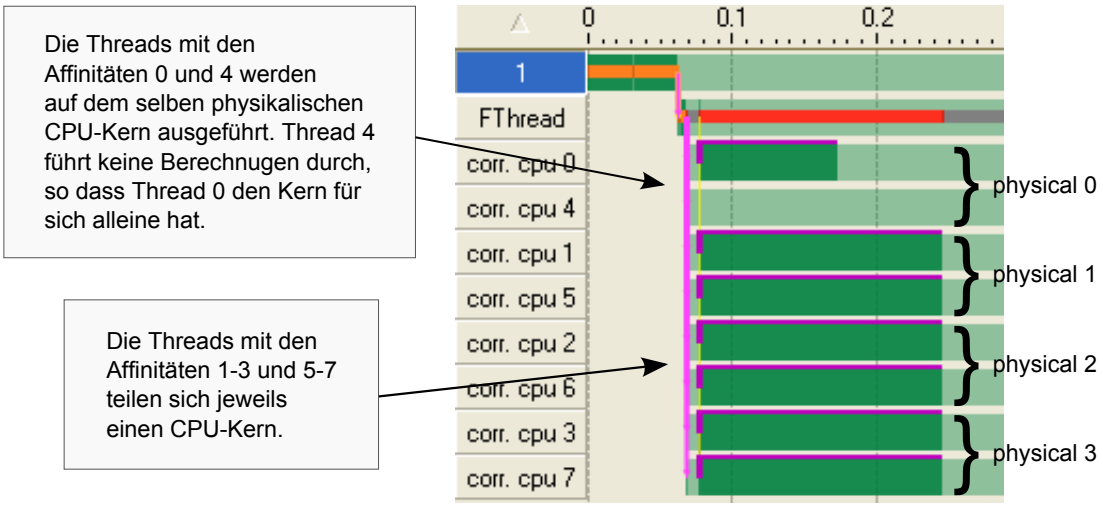

Abb. 41: Test der Zuordnung von Thread-Affinitäten zu physikalischen CPU-Kernen. Durch die Forcierung von Berechnungen auf unterschiedlichen logischen CPU-Kernen und dem anschließenden Vergleich der Berechnungsdauer konnte die von Intel empfohlene Verteilung der CPU-Affinitäten (4.4) auf physikalische CPU-Kerne bestätigt werden.

7.2.5 Analyse der Auswirkung von Thread-Prioritäten

Der nächste Versuch untersucht das Verhalten von unterschiedlich hoch priorisierten Threads auf demselben physikalischen CPU-Kern. Unter Berücksichtigung der CPU-Affinität zur Zuordnung der logischen CPU-Kerne wurden die Messungen dafür mit unterschiedlichen Thread-Prioritäten durchgeführt. Alle Threads hatten dieselbe Menge an Berechnungen durchzuführen. Für die logischen CPUs mit den Affinitäten 0 bis 3 wurden alle Threads mit hoher Priorität gestartet, die Threads die auf den logischen CPUs 5 und 6 ausgeführt wurden, hatten eine niedrige Priorität und die Threads 7 und 8 hatten dieselben Prioritäten wie die Threads 0 bis 3.

Im Ergebnis (Abbildung 42) zeigt sich eindeutig, dass bei der gleichzeitigen Ausführung eines niedrig und eines hoch priorisierten Threads auf dem selben physikalischen CPU-Kern keine messbaren Unterschiede in der Ausführungsgeschwindigkeit feststellbar sind. Für die Nuendo-Implementierung bedeutet das, dass man sich auf einer Hyper-Threading-CPU nicht mehr darauf verlassen kann, dass ein hoch priorisierter Audio-Thread Vorrang vor einem niedriger priorisierten Thread bekommt.

Zum Vergleich ist in Abbildung 43 zu sehen wie drei niedrig priorisierte Threads von einem hoch priorisierten Thread verdrängt werden. In diesem Beispiel wurde allen Threads die selbe CPU-Affinität gegeben. Während Thread 0 eine hohe Priorität zugeteilt wurde, werden die übrigen Threads mit niedriger Priorität ausgeführt.

Thread 0 verdrängt eindeutig alle Threads bis die Ausführung abgeschlossen ist. Die übrigen Threads beeinflussen sich hingegen in der Ausführungsgeschwindigkeit.

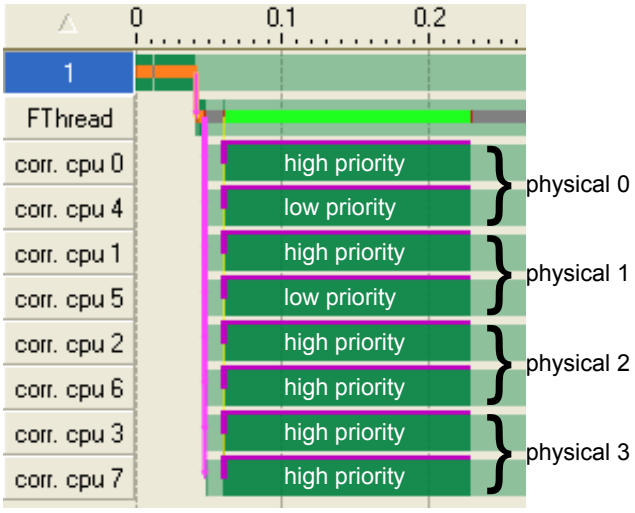

Abb. 42: Test der Auswirkung von Thread-Prioritäten. Auf den ersten beiden physikalischen CPU-Kernen laufen jeweils ein hoch und ein niedrig priorisierter Thread während auf dem dritten und vierten Kern alle Threads mit der selben Priorität ausgeführt werden. An den gleichlangen Ausführungsgeschwindigkeiten ist zu erkennen, dass die unterschiedlichen Thread-Prioritäten keinen messbaren Einfluss auf die Ausführungsgeschwindigkeit haben.

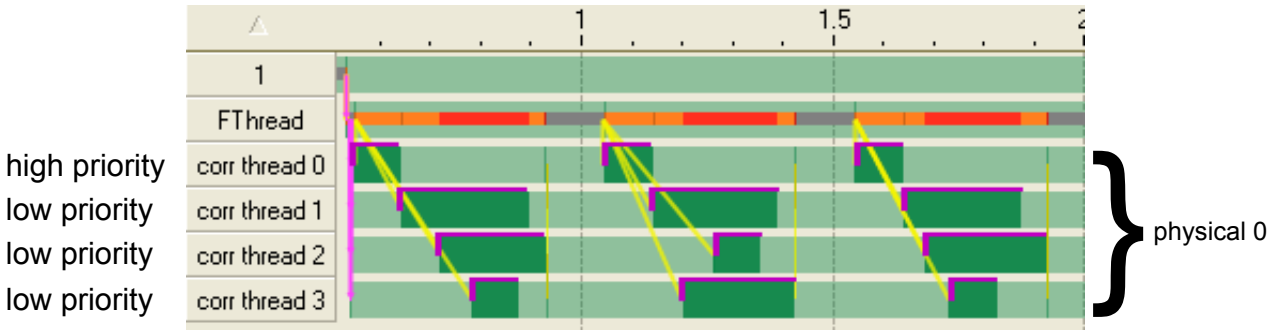

Abb. 43: 4 Threads die bei deaktiviertem Hyper-Threading auf dem selben CPU-Kern identische Berechnungen durchführen. Thread 0 hat eine hohe Priorität, die übrigen Threads werden mit niedriger Priorität ausgeführt.

7.2.6 Abschätzung des maximal möglichen Performancegewinns

Für die Abschätzung des maximal möglichen Performancegewinns mit aktiviertem Hyper-Threading wurden für jeweils 2 physikalische CPU-Kerne 3 Berechnungen parallel ausgeführt. Eine der Berechnungen konnte einen physikalischen Kern exklusiv nutzen, während die anderen beiden Berechnungen gemeinsam auf dem selben physikalischen Kern ausgeführt wurden. Um den gesuchten Wert zu errechnen, wurde der Wert für die exklusiv durchgeführten Berechnungen verdoppelt und mit der Berechnungsdauer der anderen beiden Threads verglichen.

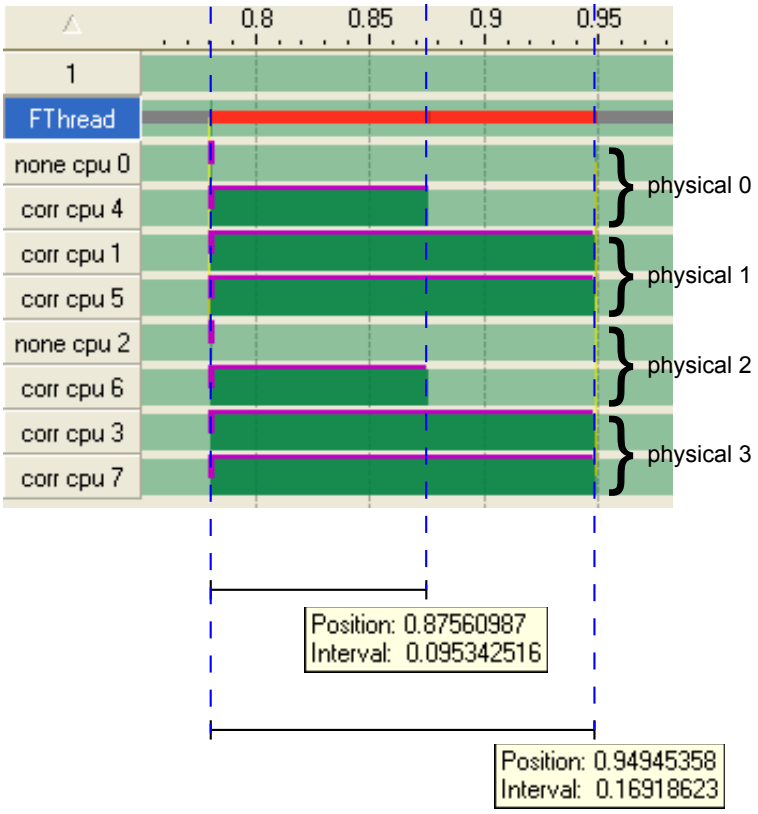

Abb. 44: Abschätzung des maximal möglichen Geschwindigkeitsgewinns durch Hyper-Threading. Die Menge der durchgeführten Berechnungen ist für alle CPUs gleich. Während auf den physikalischen 0 und 2 nur jeweils einer der beiden logischen CPU-Kerne benutzt wird, werden auf den physikalischen Kernen 1 und 3 beide logischen CPU-Kerne mit benutzt woraus sich die doppelte Menge an Berechnungen ergibt.

Wie in Abbildung 44 zu sehen ist brauchen die Threads, die sich einen physikalischen CPU-Kern teilen, nicht ganz doppelt so lange wie die beiden exklusiv ausgeführten Threads. Daraus ergibt sich ein Geschwindigkeitsgewinn von fast 13%.

8 Entwurf und Verifizierung von Lösungsvorschlägen

Nachdem die Anlayse mit Hilfe der Prototypen ergeben hat, dass gegenüber dem derzeitigem Stand der Entwicklung durchaus noch Optimierungspotenzial vorhanden ist, galt es zu überlegen wie die bestehende Implementierung der Audio-Engine dahingehend überarbeitet werden kann, um sich die Hyper-Threading-Technologie optimal zunutze machen zu können.

Die Zielsetzung besteht zunächst darin, zu verhindern, dass unter bestimmten Bedingungen Lastsituationen auftreten können, die zu einer Verschlechterung der Performance gegenüber dem Audio-Processing mit deaktiviertem Hyper-Threading auf der selben Maschine führen. Zur Zeit gibt Steinberg seinen Kunden die Empfehlung Hyper-Threading zu deaktivieren, das Erreichen dieser Zielsetzung wäre daher schon ein großer Fortschritt gegenüber dem bisherigen Stand.

Im besten Fall könnte es sogar gelingen, die gemessene Performanceverbesserung vom Prototypen zu übernehmen, um damit die Audio-Engine auf Hyper-Threading-System nicht unerheblich zu beschleunigen.

8.1 Lösungsvorschläge

8.1.1 Prozess-Affinität

Anstelle der in Kapitel 4.4 beschriebenen Möglichkeit, die Affinität für einen Thread festzulegen, kann auch eine Affinität für einen Prozess vergeben werden. Der Grundgedanke dieser Herangehensweise ist, dass niedrig priorisierte Threads aus dem selben Prozess davon abgehalten werden sollen, auf einen vermeintlich unbeschäftigten CPU-Kern auszuweichen. Das könnte dadurch erreicht werden, dass die Prozessaffinität so vergeben wird, dass nur jeweils einer der beiden logischen Kerne benutzt werden darf.

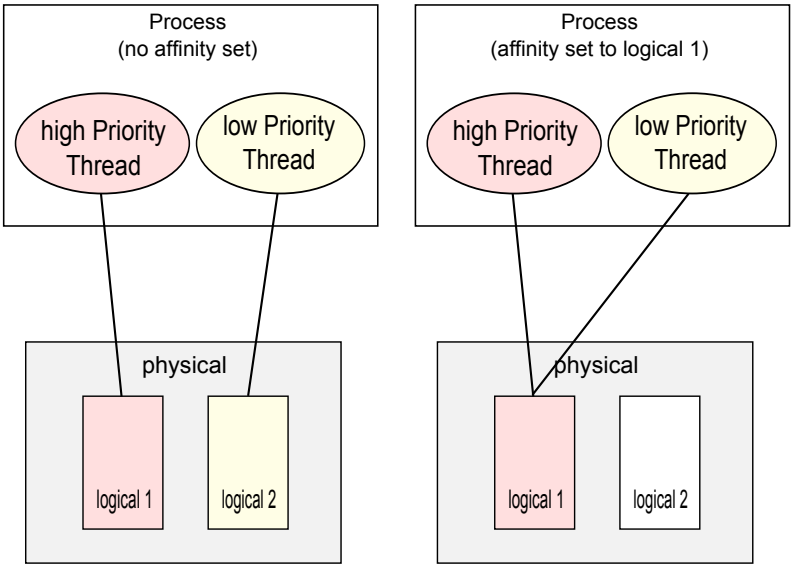

Abb. 45: Gewünschtes Schedulingverhalten mit und ohne Prozess-Affinität. Im linken Bild ist zu erkennen wie sich der Scheduler verhält, wenn ein hoch und ein niedrig priorisierter Thread gleichzeitig ausgeführt werden. Die Threads werden nacheinander auf alle freien (logischen) CPU-Kerne verteilt. Bei zwei freien logischen CPU-Kernen führt das dazu, dass beide Threads gleichberechtigt ausgeführt werden. Um das zu verhindern, soll die Affinität für den Prozess so gesetzt werden, dass der Prozess nur noch auf jeweils einer der beiden logischen CPU-Kerne ausgeführt werden darf.

8.1.2 CPU-Kern reservieren

Diese Lösung sieht die Reservierung eines physikalischen CPU-Kernes für alle Threads vor, die für das Audio-Processing nicht relevant sind. Hier würde die Affinität für alle Threads, die beispielsweise für die GUI oder den *VST-Preloader* zuständig sind, so vergeben werden, dass sie nur einen physikalischen Kern zur Verfügung haben, während alle Audio-Threads auf den verbleibenden CPU-Kernen arbeiten.

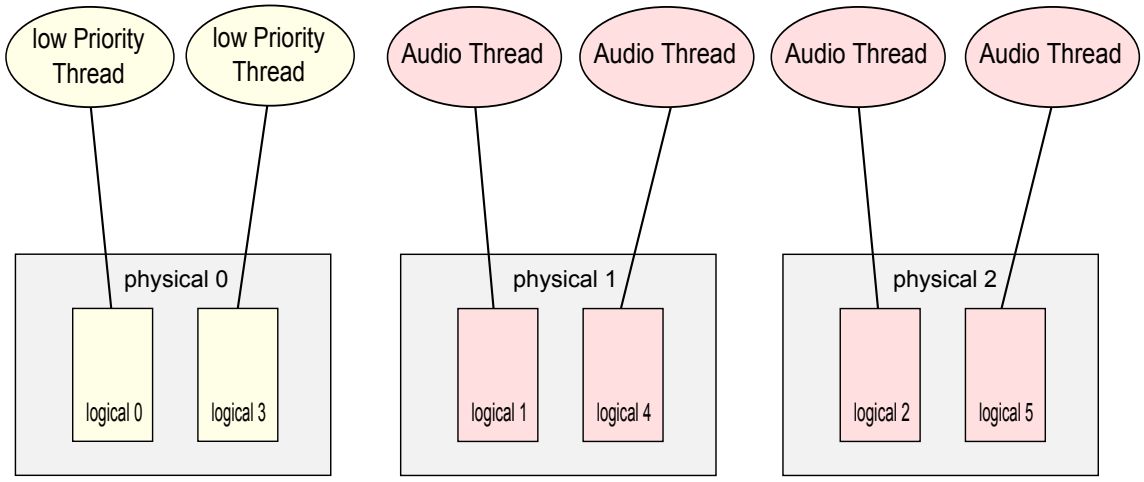

Abb. 46: Trennung der Audio-Threads von niedriger priorisierten Threads durch Thread-Affinität. Durch die Auslagerung niedrig priorisierter Thneads auf einen eigenen physikalischen CPU-Kern wäre das Audio-Processing von unwichtigeren Aufgaben getrennt, so dass das Audio-Processing dadurch nicht gestört werden könnte.

8.1.3 Idle-Thread

Für diese Lösung soll die Möglichkeit untersucht werden, für die Dauer des Audio-Processings, jeweils einen der beiden logischen CPU-Kerne so zu beschäftigen, dass der andere logische CPU-Kern nach Möglichkeit nicht gestört wird. Damit soll verhindert werden, dass logische CPU-Kerne, die nicht für das Audio-Prozessing verwendet werden, von dem Scheduler des Betriebssystems für die Abarbeitung niedrig priorisierter Threads verwendet werden.

Der Vorteil gegenüber der Deaktivierung von Hyper-Threading besteht darin, dass der Idle-Thread nur genau so lange ausgeführt wird, wie auch das Audio-Processing Stattfindet. Alle anderen Bestandteile der Anwendung können in der Zwischenzeit von der Mehrleistung durch Hyper-Threading profitieren.

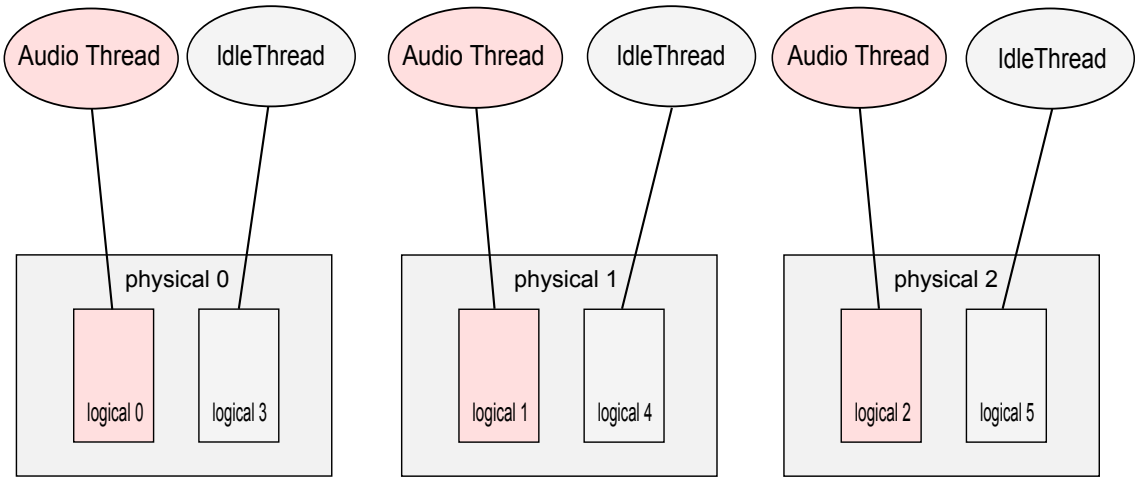

Abb. 47: 'Blockieren' des zweiten logischen CPU-Kernes mittels Idle-Thread. Während die rechenintensiven Audio-Threads auf einem der beiden logischen Kerne ausgeführt werden, soll ein Threads, der auf dem anderen logische CPU-Kern ausgeführt wird, dafür sorgen, dass der Audio-Thread bei der Ausführung nicht gestört werden kann.

8.2 Verifizierung der Vorschläge

8.2.1 Prozess-Affinität

Um die Machbarkeit dieser Idee zu demonstrieren, wurde der Prototyp aus Kapitel 7.2.2 verwendet. Im ersten Testlauf wurden jeweils 4 Threads mit hoher Priorität mit der Affinität 0-3 und 4 Threads mit niedriger Priorität mit der Affinität 4-7 gestartet. Wie bereits erwartet, zeigt die Analyse mit dem Thread-Profiler, dass alle Threads nahezu gleichberechtigt ausgeführt werden (Abbildung 48).

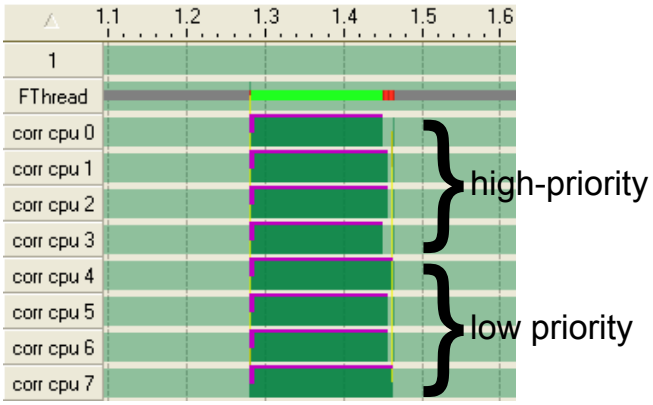

Abb. 48: Testlauf mit 8 Threads von denen jeweils 4 eine hohe und 4 eine niedrige Priorität haben. Wie bereits in Kapitel 7.2.5 näher erläutert, hat die Vergabe von Thread-Prioritäten keinen Einfluss auf das Scheduling-Verhalten.

Für einen weiteren Test wurde der Prototyp so modifiziert, dass die Affinität für den Prozess auf 0 bis 4 gesetzt wurde, um zu gewährleisten, dass nur jeweils einer der beiden logischen CPU-Kerne benutzt wird. Der Testlauf wurde anschließend mit der selben Konfiguration erneut ausgeführt.

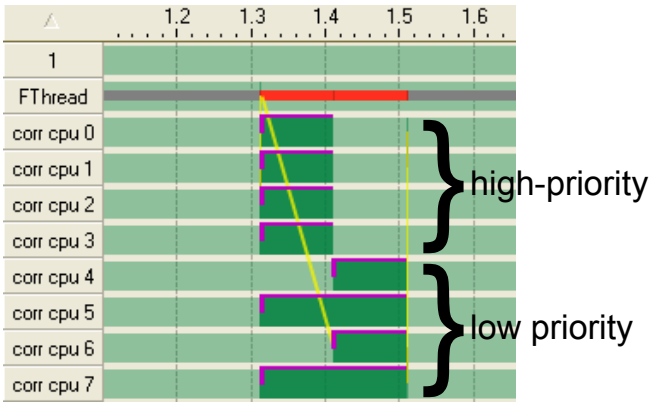

Abb. 49: Testlauf nachdem die Affinität des Prozesses auf 0-4 gesetzt wurde. Die hoch priorisierten Threads werden in fast der Hälfte der Zeit ausgeführt, wobei die niedrig priorisierten Threads entsprechend verzögert werden.

Der Vergleich der Thread-Profiler-Analysen in Abbildung 48 und 49 zeigt, dass die Bearbeitung der niedrig priorisierten Threads entweder verzögert wird oder erst in dem Moment startet, wenn die Ausführung der hoch priorisierten Threads abgeschlossen ist.

Auch wenn des Ergebnis auf den ersten Blick sehr vielversprechend erscheint, lässt sich mit der Vergabe der Prozess-Affinität nicht verhindern, dass der Scheduler des Betriebssystems niedrig priorisierte Threads aus anderen Prozessen auf den verbleibenden logischen CPU-Kernen ausführt. Für Echtzeit-Anwendungen ist dieses Vorgehen daher leider nicht geeignet.

8.2.2 CPU-Kern reservieren

Dieses Vorgehen hat leider den Nachteil, dass auf einer CPU mit 4 physikalischen Kernen 25% der Leistung nicht mehr für das Audio-Processing zur Verfügung steht und zusätzlich nicht sichergestellt werden kann, dass das Audio-Processing nicht von anderen Prozessen gestört wird.

Möglicherweise ergibt sich an dieser Stelle in dem Moment ein interessanter Ansatz, wenn die Anzahl der verfügbaren CPU-Kerne so stark angewachsen ist, dass der 'Verlust' eines CPU-Kernes für das Audio-Processing nicht mehr so stark ins Gewicht fällt.

8.2.3 Idle-Thread

Um herauszufinden, auf welche Art eine logische CPU am besten beschäftigt gehalten werden kann ohne dabei den Thread, der auf dem zweiten logischen CPU-Kern arbeitet, auszubremsen, wurde der Prototyp um einen entsprechenden Prozess erweitert. Nach einigen Experimenten mit unterschiedlichen Herangehensweisen, beispielsweise zur Erzeugung möglichst vieler stall-cycles, stellte sich heraus, dass auf Windows-Systemen die Anweisung _mm_pause () am besten dafür geeignet ist. Der Befehl füllt zwar die Instruction Queue für einen logischen CPU-Kern, der physikalische CPU-Kern bleibt davon aber unberührt, da für diesen Befehl nichts ausgeführt wird.

Da die Anzahl der Pause-Befehle sehr viel größer sein muss als der Overhead durch die umgebende Schleife, wird der Befehl einfach mehrere Male hintereinander ausgeführt. Im Code sieht das etwas merkwürdig aus:

```
while ( ProcessIdle::isProcessing || isEntering )
{
#if WINDOWS
    _mm_pause ();
    _mm_pause ();
    _mm_pause ();
    _mm_pause ();
    _mm_pause ();
    _mm_pause ();
    _mm_pause ();
    _mm_pause ();
    _mm_pause ();
    _mm_pause ();
    _mm_pause ();
    _mm_pause ();
#endif
}
```

In der disassemblierten Ansicht wird aber deutlich, dass dieses Vorgehen notwendig ist, weil der Anteil des Overheads den Einfluss der pause-Anweisungen sonst um ein Vielfaches übersteigen würde.[21]

```
0x440F   process+0xf:     cmp       DWORD PTR [0x415430h], 0x0h
0x4416                    jnz       process+0x23
0x4418   process+0x18:    movzx     eax, BYTE PTR [0x4151b3h]
0x441F                    test      eax, eax
0x4421                    je        process+0x3b
0x4423   process+0x23:    pause
0x4425                    pause
0x4427                    pause
0x4429                    pause
0x442B                    pause
0x442D                    pause
0x442F                    pause
0x4431                    pause
0x4433                    pause
0x4435                    pause
0x4437                    pause
0x4439                    jmp       process+0xf
[...]
0x443B   process+0x3b:
```

Die Auswertung zeigt, dass die Idle-Threads die Berechnung der Korrelation so wenig stören, dass es gegenüber dem möglichen Performancegewinn vertretbar wäre. Bei der gleichzeitigen Ausführung eines Korrelationsthreads mit einem Idle-Thread auf dem selben pyhsikalischen CPU-Kern, ist die Berechnung der Korrelation nur knapp 3% langsamer als bei der Berechnung ohne Idle-Thread. (Abbildung 50)

[21] Die Compiler-Optimierungen dürfen dafür natürlich nicht zu hoch eingestellt werden.

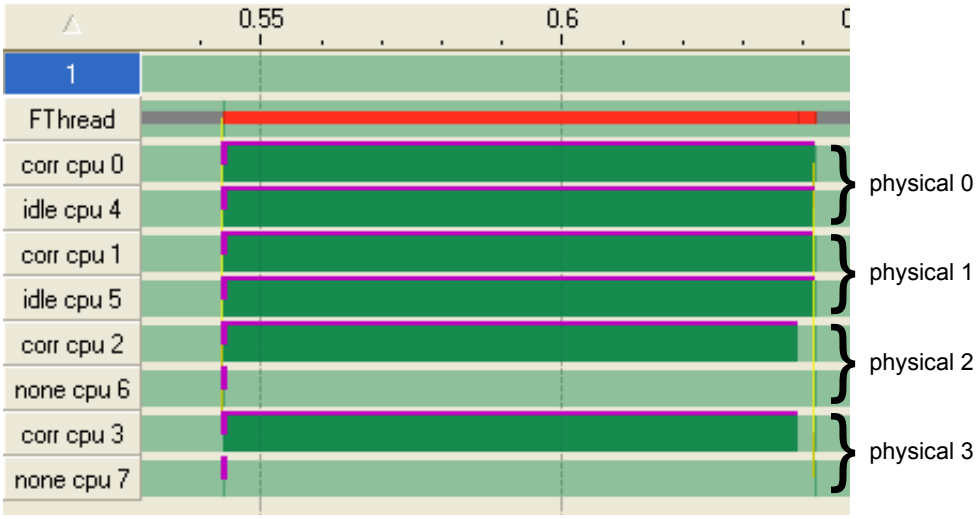

Abb. 50: Bei der gleichzeitigen Ausführung einer Berechnung und einem Idle-Thread auf einem physikalischen CPU-Kern (physical 0 und 1), verliert der Thread der die Berechnung durchführt, nur knapp 3% gegenüber der Berechnung ohne Idle-Thread (physical 2 und 3).

8.3 Tests an der Nuendo-Implementierung

8.3.1 Bisherige Implementierung der Audio-Engine

Um besser auf die Implementierung der Idle-Threads eingehen zu können, möchte ich an dieser Stelle kurz auf Abschnitt 2.4 zurückgreifen und auf die derzeitige Implementierung eingehen. Abbildung 52 zeigt einen etwas vereinfachten Ablauf des Audio-Processings, wie er sich für den Projektaufbau aus Abbildung 51 ergeben würde.

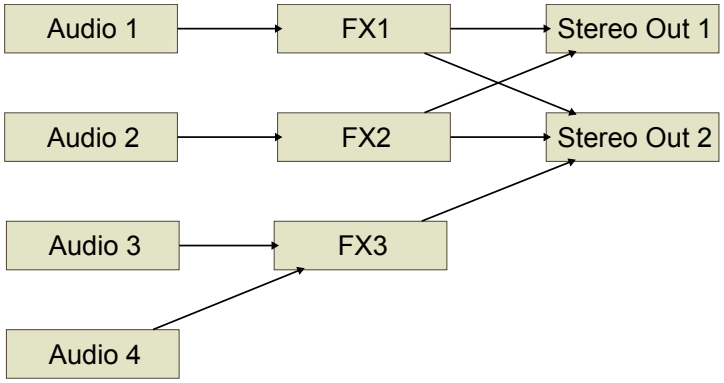

Abb. 51: Projektaufbau für das Audio-Processing in Abbildung 52.

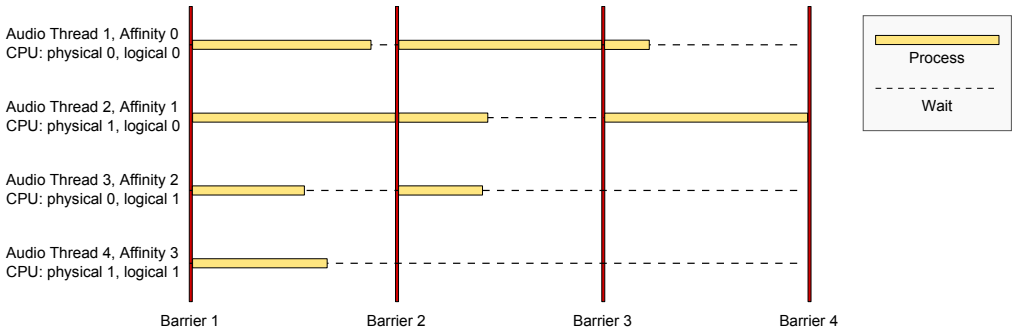

Abb. 52: Vereinfachte Darstellung des Audio-Processings, das aus dem Projektaufbau aus Abbildung 51 resultiert. Nach der ersten Barriere können alle vier Audiospuren parallel berechnet werden, da untereinander keine Abhängigkeiten bestehen. Anschließend werden die drei Effekte und die beiden Stereo-Ausgänge berechnet.

Mit der ersten Barriere wird das Audio-Processing in Gang gesetzt und die vier unabhängigen Kanäle werden parallel berechnet. Anschließend folgt die Berechnung der drei Effekte, sowie die Ausgabe an die Ausganskanäle. Zwischen den einzelnen Schritten muss jeweils auf eine Barriere synchronisiert werden, um zu verhindern, dass die Berechnung eines Effektes begonnen wird bevor alle sendenden Kanäle fertig berechnet sind.

Sobald ein Audio-Thread die Berechnung einer Process-Unit abgeschlossen hat, erreicht er eine Barriere und befindet sich anschließend im wait-state bis alle anderen Audio-Threads die entsprechende Barriere erreicht haben. Sofern der Scheduler keinen Unterschied zwischen normalen Mehrkernsystemen und Systemen mit Hyper-Threading macht, sieht er hier einfach nur

einen unbeschäftigten CPU-Kern, auf dem auch andere, weniger hoch priorisierte Threads ausgeführt werden können. In der Folge können die Audio-Threads, die noch auf dem anderen logischen CPU-Kern beschäftigt sind, ausgebremst werden (Abbildung 53).

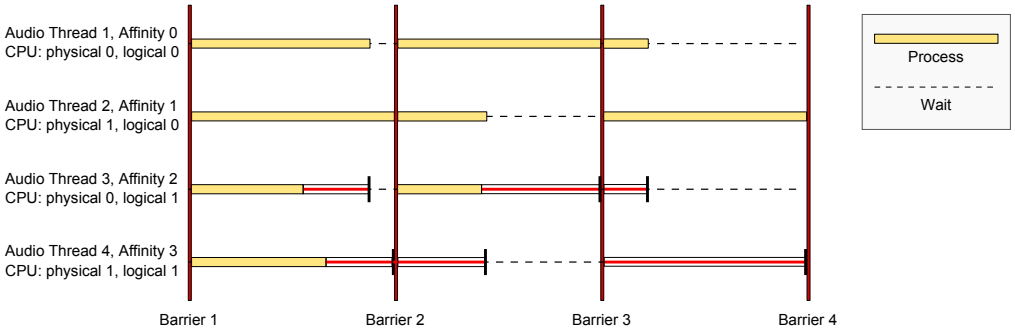

Abb. 53: Diese Illustration zeigt den selben Vorgang wie Abbildung 52 mit dem Unterschied, dass hier die Zeitabschnitte, auf denen ein Audio-Thread möglicherweise durch Threads gestört wird, die nicht für das Audio-Processing arbeiten, rot markiert sind.

8.3.2 Anpassungen an der Audio-Engine

Um das im vorhergehenden Kapitel erläuterte Problem auf Hyper-Threading-Systemen in den Griff zu bekommen, wurde die bisherige Implementierung der Barriern dahingehend angepasst, dass von zwei Threads, die gemeinsam auf einer physikalischen CPU laufen, jeweils der Thread, der schneller fertig wird, als Idle-Thread weiterläuft bis auch der zugehörige Thread die Berechnung abgeschlossen hat.

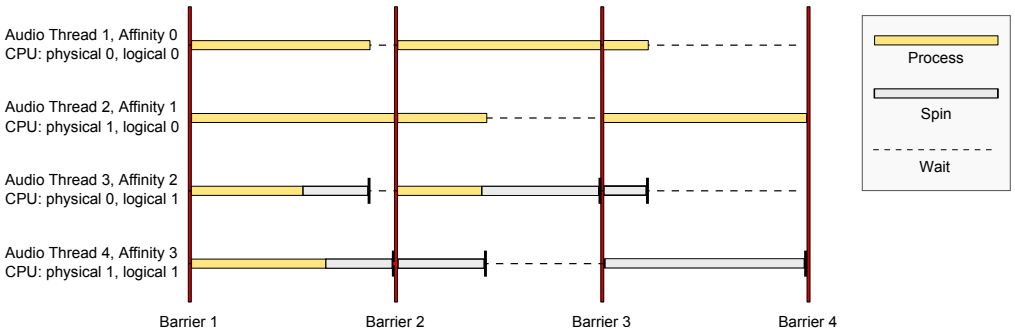

Abb. 54: Die Grafik zeigt das Audio-Processing aus Abbildung 52. Die mit 'spin' gekennzeichneten Abschnitte führen eine Schleife aus, die den selben Code ausführt wie der Idle-Thread aus Abschnitt 8.2.3. Auf diese Weise soll verhindert werden, dass der Scheduler die vermeintlich unbeschäftigte CPU mit niedrig priorisierten Threads auszulasten versucht.

8.3.3 Thread-Profiler Analyse

Für die Untersuchung der Auswirkungen dieser Modifikation wurde zunächst das Worst-Case-Projekt mit 4 Kanälen herangezogen. Um das Verhältnis von reiner Audio-Processing-Zeit zu der für die Synchronisation aufgewendeten Zeit so zu beeinflussen, dass die Synchronisationszeit weniger ins Gewicht fällt, wurde die Gesamtdauer des Audio-Processings auf circa 2 ms gesteigert. Um das zu erreichen wurde die Anzahl der Effekte für jeden Audio-Kanal entsprechend erhöht.

Zunächst wurde mit der unmodifizierten Nuendo-Version und deaktiviertem Hyperthreading eine Referenzmessung vorgenommen, deren Messwerte anschließend mit den Messwerten der modifizierten und der unmodifizierten Nuendo-Version bei aktiviertem Hyper-Threading verglichen wurden.

Mit Hilfe des Thread-Profilers wurde im ersten Schritt untersucht, ob die Modifikation zu dem erwarteten Verhalten führt und der niedrig priorisierte Thread für die Berechnung der Wellenform nicht mehr gleichberechtig mit den Threads für das Audio-Processing ausgeführt wird. In Abbildung 55 bis 57 ist das Ergebnis dieser Analyse zu sehen.

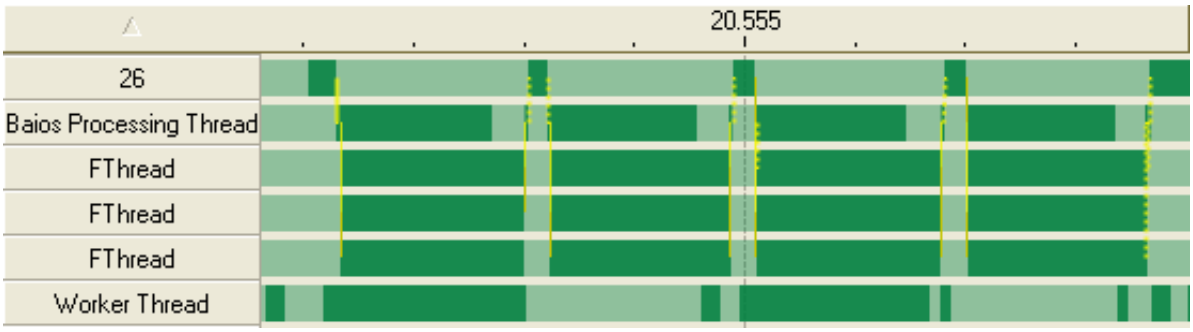

Abb. 55: Worst-Case-Projekt mit deaktiviertem Hyperthreading. Der Worker-Thread wird während des Audio-Processing nicht oder nur verzögert ausgeführt

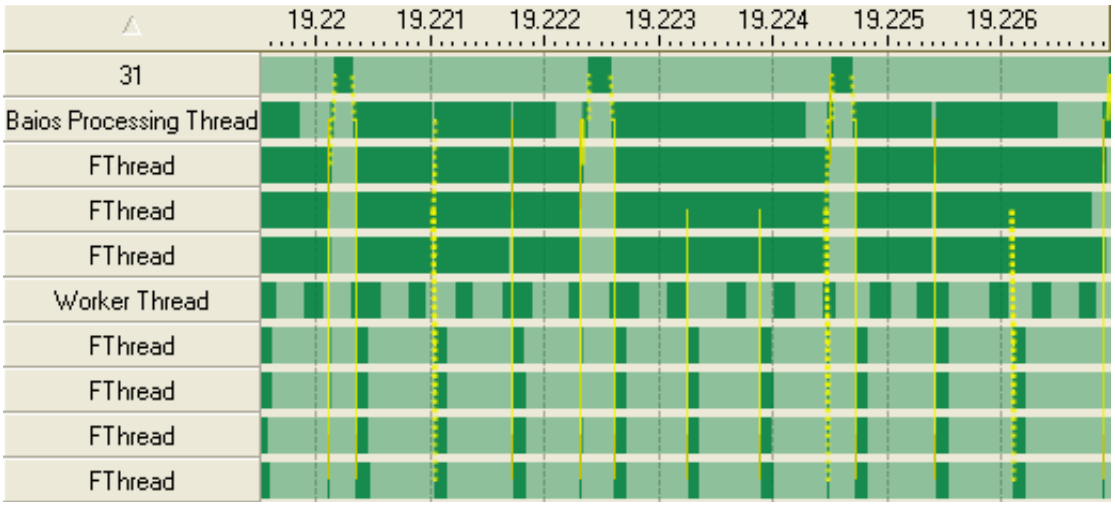

Abb. 56: Worst-Case-Projekt mit aktiviertem Hyperthreading. Bei der unmodifizierten Nuendo-Version wird die Ausführung des Worker Threads vom Audio-Processing nicht unterbrochen.

Abb. 57: Worst-Case-Projekt mit aktiviertem Hyperthreading und modifizierter Nuendo Version. Für den Zeitraum, in dem die Barriern sich in einem Spin-Wait-Zustand befinden, sind die violetten Kennzeichnungen durchgängig. Der Worker-Thread verhält sich genauso wie bei deaktiviertem Hyperthreading und wird für die Dauer des Audio-Processings verzögert.

8.3.4 Messung der Processing-Geschwindigkeit

Um die Auswirkungen der Modifikation auf die Dauer des Audio-Processings zu messen wurden, im Anschluss an die Analyse mit dem Thread-Profiler, die Processing-Zeiten unter Windows XP und Windows Vista gemessen. Für die Messung wurde hier zusätzlich das Best-Case-Projekt herangezogen, das genauso wie das Worst-Case-Projekt so verändert wurde, dass die Processing-Zeit bei etwas 2 ms lag.

Gemessen wurde über 3000 ASIO-Blöcke wobei jeweils der Durchschnittswert (avg.) sowie der langsamste gemessene Wert (max.) für die Processing-Zeit in die Auswertung mit einbezogen wurden. Die Ergebnisse dieser Messungen sind in den nachfolgend dargestellten Tabellen aufgeführt. Die Messung für die modifizierte Nuendo-Version ist mit 'htBarriers' bezeichnet.

Best-Case, Windows XP

Configuration	avg. ms	max. ms
smt disabled	2,069	2,800
smt enabled	1,850	2,013
smt enabled, htBarriers	1,860	1,970

Best-Case, Windows Vista

Configuration	avg. ms	max. ms
smt disabled	2,094	2,346
smt enabled	1,911	2,374
smt enabled, htBarriers	1,933	2,122

Worst-Case, Windows XP

Configuration	avg. ms	max. ms
smt disabled	1,630	1,861
smt enabled	2,039	2,528
smt enabled, htBarriers	2,162	2,358

Worst-Case, Windows Vista

Configuration	avg. ms	max. ms
smt disabled	1,705	1,956
smt enabled	2,243	2,783
smt enabled, htBarriers	2,259	2,456

8.3.5 Auswertung der Messergebnisse

Die Analyse mit dem Thread-Profiler zeigt, dass mit der modifizierten Nuendo-Version ein verbessertes Schedulingverhalten erreicht werden kann. Die Ausführung des niedrig priorisierten Threads wird, im Gegensatz zu der unmodifizierten Version, für die Dauer des Audio-Processings wie gewünscht zurückgestellt.

Vergleicht man die Tests, die mit aktiviertem Hyper-Threading durchgeführt wurden, lässt sich feststellen, dass bei der Messung der maximalen Processing-Geschwindigkeit die modifizierte Version ausnahmslos besser abschneidet als die unmodifizierte Version, während die Messung für die durchschittliche Processing-Geschwindigkeit die unmodifizierte Version ein wenig besser aussehen lässt.

Da die maximale Processing-Geschwindigkeit für das Audio-Processing ausschlaggebend ist, können die Messergebnisse aller Konfigurationen so zusammengefasst werden, dass die modifizierte Version für das Best-Case-Projekt die besten Ergebnisse liefert. Bei der Messung für das Worst-Case-Projekt liegt hingegen die Variante mit deaktiviertem Hyper-Threading eindeutig vorne.

Bei der genaueren Untersuchung einzelner Processing-Blöcke des Worst-Case Projekts im Thread-Profiler lässt sich die Ursache für dieses Verhalten erkennen. Vergleicht man eine größere Anzahl von Processing-Blöcken aus den beiden Tests der unmodifizierten Version, lässt sich feststellen, dass ein Großteil der Processing-Blöcke bei aktiviertem Hyper-Threading genauso lang ist wie bei deaktiviertem Hyper-Threading. Mit ähnlicher Häufigkeit sind aber Processing-Blöcke zu finden die durch den niedrig priorisierten Thread stark verzögert werden, so dass sich auch der Durchschnittswert entsprechend verschlechtert.

Die modifizierte Nuendo-Version zeigt in der Thread-Profiler-Analyse ein sehr viel konstanteres Verhalten. Nur selten sind Processing-Blöcke zu finden, die von der durchschnittlichen Processing-Geschwindigkeit auffällig abweichen. Das Problem liegt vielmehr darin, dass alle Processig-Blöcke konstant langsamer ausgeführt werden als bei der unmodifizierten Version mit deaktiviertem Hyper-Threading.

Anhand dieser Ergebnisse zeigt sich, dass es zwar gelingt, das Schedulingverhalten entscheidend zu verbessern, der erhoffte Gewinn allerdings an anderer Stelle wieder verloren geht. Im Gegensatz zu den Tests am Prototyp macht sich die gleichzeitige Ausführung einer Spinloop und eines Audio-Threads auf einem physikalischen CPU-Kern sehr deutlich als Geschwindigkeitsverlust bemerkbar.

9 Zusammenfassung / Fazit

Bei dem Vergleich zwischen dem ersten, 2002 eingeführten, Intel-Prozessor mit Hyper-Threading-Technologie und dem für diese Arbeit verwendeten Core i7 wird deutlich, dass die Nehalem-Architektur gegenüber dem älteren NetBurst-Prozessor wesentlich effizienter geworden ist.

Wie von Intel selbst bestätigt wird[14], neigten die NetBurst-Prozessoren dazu, rechenintesive CPU-Last ineffizient auszuführen. Beim Vergleich der Messergebnisse in Kapitel 6.3 wird deutlich, dass Intel dieses Problem offensichtlich in den Griff bekommen hat.

Während der Xeon-Prozessor, bei rechenintensiven Aufgaben, durch die Hyper-Threading-Technologie noch Geschwindigkeitsgewinne von bis zu 90% zu erzielen konnte, liegt das beste gemessene Ergebnis für den Core i7 bei knapp 30%.

Angesichts dieser Werte mag der Eindruck enstehen, dass die Implementierung von Hyper-Threading, bei der Effizienz des hier getesteten Prozessormodells, wenig sinnvoll ist. Dazu sollte allerdings beachtet werden, dass der Fokus dieser Arbeit auf sehr rechenintensiven Echtzeitanwendungen liegt.

In einen Umfeld, in dem der Schwerpunkt der Aufgaben des Prozessors darin liegt, möglichst viele Threads gleichzeitig zu bearbeiten, kommt man unter Umständen zu ganz anderen Ergebnissen. Da die Implementierung der Hyper-Threading-Technologie nur etwa 5% der Chipfläche beansprucht[16] kann das Kosten/Nutzen-Verhältnis hier also durchaus gerechtfertigt sein.

Für das Echtzeit-Audio-Prozessing bringt die Hyper-Threading-Technologie leider keinen echten Vorteil. Einerseits sind mit einem Projekt, dass daraufhin optimiert ist, möglichst hohes Optimierungspotential für Hyper-Threading zu bieten, Performancegewinne von fast 10% zu erzielen, andererseits verliert man leicht Performance, sobald in einem Projekt CPU-Kerne ungenutzt bleiben und auf dem System gleichzeitig niedrig priorisierte Threads ausgeführt werden.

Weder bei Windows XP noch bei Windows Vista gibt es eine Möglichkeit, so in das Schedulingverhalten einzugreifen, dass zwischen logischen und physikalischen Kernen ein Unterschied gemacht wird. In der Folge lässt sich nicht sicherstellen, dass Echtzeit-Threads gegenüber niedrig priorisierten Threads vorrangig ausgeführt werden.

Glossar

80386

Erste 32-Bit-CPU von Intel, 1985 herausgebracht.

8086

Erste 1978 herausgebrachte 16-Bit-CPU von Intel. Mit dieser CPU wurde die erste CPU entwickelt, die auf dem x86 Befehlssatz basierte, zu dem bis heute alle entsprechenden Prozessoren abwärtskompatibel sind. Damals entwickelte Programme sind auch heute noch auf aktuellen x86-CPUs lauffähig.

8088

langsamere und billigere Variante des 8086 mit 8Bit-Bus.

ALU

Arithmetic Logic Unit, Bestandteil des CPU-Kernes, der für die Ausführungen von logischen und Integer-Operationen zuständig ist.

ASIO-Block

Samples die benötigt werden um einen Ausgangs-Buffer zu füllen.

ASIO

Audio Stream Input Output; Von Steinberg 1997 entwickelte Softwareschnittstelle, die auf dem ASIO-Treiber aufsetzt und unter anderem darauf hin optimiert ist, Audio-Processing mit niedrigen Latenzen zu realisieren.

Affinität

siehe Kapitel 4.4.

Assemblersprache

Assemblersprachen sind symbolische Darstellungen von Maschinensprachen. Bei einer sehr konsequenten Umsetztung bedeutet das, dass ein Assemblerbefehl genau einen Maschinenbefehl erzeugt, in der Praxis wird die Assemblerprogrammierung beispielsweise durch die Möglichkeit Variablen anzulegen meistens noch etwas vereinfacht. Der Assembler sorgt dann dafür, das in Assemblersprache geschriebene Programm durch Austauschen und Ersetzen von Assemblerbefehlen zu Binärcodes und Adressen in Maschinensprache zu übersetzen.

Assembler

Programm zum Übersetzen von Assemblersprache in ausführbare Programme die Maschinensprache bzw. Binärcode enthalten. Auch: gebräuchliche Abkürzung für *Assemblersprache*.

Barriere

Barrieren können für die Synchronisierung mehrer Threads verwendet werden. Eine Barriere erhält eine feste Anzahl von Threads die miteinander synchronisiert werden sollen. Ein Thread der auf die Barriere wartet, wartet genau so lange, bis insgesamt die vorgegebene Anzahl an Threads erreicht worden ist. Warten beispielsweise 3 Threads auf eine Barriere, die auf 4 Threads synchronisiert, wird die Ausführung der 3 Threads so lange unterbrochen bis sich ein vierter Thread bei der Barriere 'anmeldet'.

Benchmark

Programm zur Geschwindigkeitsermittlung eines Systems in einer definierten Lastsituation.

Binärcode

Normalerweise von einem *Assembler* generierter Code für ein ausführbares Binärprogramm. Der Code besteht nicht mehr aus lesbaren Zeichen, sondern nur noch aus Bytefolgen.

Cache-Block

Bei Zugriffen auf den Hauptspeicher wird, jeweils beginnend mit der Adresse des ersten Zugriffs, ein Block des Speichers in den Cache-Speicher übertragen. Der Wert kann sich je nach CPU-Typ unterscheiden und wird meist als Cache-Line-Size angegeben.

Cache-Line-Size

siehe *Cache-Block*.

Compiler

Programm zur Übersetzung von einer Hochsprache in Assembler oder Maschinensprache (Binärcodes).

DDR

Mit DDR-SDRAM (Double Data Rate) wird eine Weiterentwicklung von *SDRAM* bezeichnet, die sowohl mit steigender als auch mit fallender Flanke des Taktsignals Daten liefert. Daraus ergibt sich eine Verdopplung der Datenrate[16].

Debugger

Programm zur Fehlersuche bei der Ausführung von Programmcode. Ermöglicht die Schrittweise Ausführungs eines Programmes. Bei Hochsprechen meistens in die Entsprechende Entwicklungsumgebung integriert.

Die

Die Halbleiterfläche innerhalb eines Chips.

Disassembliert

Assemblercode der von einem Disassembler aus Maschinencode in Assemblersprache

zurückübersetzt wurde.

Echtzeit

Echtzeitsysteme erfüllen die Vorgabe, in jedem Fall innerhalb eines festgelegten Zeitrahmens, eine bestimmte Aufgabe zu erfüllen. Die Echtzeit-Anforderung an eine Audio-Anwendung besteht beispielsweise darin, den Ausgangs-Buffer so schnell zu füllen, das keine Aussetzer bei der Wiedergabe entstehen.

Execution Unit

Bestandteil des CPU-Kerns, der für die Ausführung von Micro-Operationen, wie zum Beispiel logischen oder Integer-Operationen auf der *ALU* oder Floating-Point-Operationen auf der *FPU*, zuständig ist.

FPU

Floating Point Unit, Bestandteil des CPU-Kernes der Floating-Point-Operationen ausführt.

FSB

Front Side Bus, Bussystem zwischen CPU und Memory Controller.

Floating-Point

Gleitkommazahl oder Fließkommazahl, Teilmenge der rationalen Zahlen mit endlicher Genauigkeit.

GUI

Graphical User Interface, die grafische Benutzeroberfläche eines Programmes.

Gleitkommazahlen

siehe Floating-Point.

Hochsprache

Programmiersprachen, die darauf ausgerichtet sind, für einen Anwendungsentwickler gut verständlich zu sein und sich darin von Sprachen unterer Ebenen, wie zum Beispiel Assembler oder Binärcodes unterscheiden.

Idle-Thread

Der Begriff ist vom Leerlaufprozess (Idle-Process) abgeleitet und bezeichnet in dieser Arbeit einen Thread, der im Idealfall dafür sorgt, dass auf einem logischen CPU-Kern für die Dauer seiner Ausführung keine anderen Threads ausgeführt werden können. Der Unterschied dieses Idle-Threads zum normalen Leerlaufprozess besteht darin, dass er um andere Threads zu verdrängen mit hoher Priorität ausgeführt werden muss. Im Gegensatz zum Leerlaufprozess, der mit der niedrigst möglichen Priorität ausgeführt wird und damit von allen anderen Threads verdrängt wird.

Instruction Pointer

Auch Programm Counter genannt, meist ein Register innerhalb der CPU, das eine Adresse enthält, die auf die Speicherstelle der nächsten auszuführenden Instruktion zeigt.

Instruction Stream

Instruktionsfluss, Folge von Instruktionen.

Integer

Ganze Zahlen.

Itanium

1999 eingeführte Prozessor-Serie, die auf der neuen, gemeinsam mit HP entwickelten IA-64-Architektur basiert. Die Linie wird von Intel noch weiter geführt, spielt auf dem Comsumer-Markt aber überhaupt keine Rolle mehr und konnte sich auch im High-End-Server Umfeld nie durchsetzen.

Maschinenwort

siehe *Wort*.

Micro-Operation

Vom Prozessor dekodierte x86 Instruktionen. In modernen CPUs werden x86 Instruktionen nicht direkt ausgeführt, sondern vorher zu ein oder mehreren Micro-Instruktionen dekodiert, die anschließend vom CPU-Kern ausgeführt werden.

NUMA

Non Uniform Memory Access, nicht einheitlicher Speicherzugriff, bezieht sich auf die Zugriffszeiten auf den Speicher. In Systemen mit mehreren Prozessorkernen ist der L1 Cache und meist auch der L2 Cache einem bestimmten Prozessorkern zugeordnet. Sobald CPU1 auf den Cache von CPU2 zugreifen muss oder ein Scheduler Prozesse oder Threads von einem CPU-Kern auf den anderen verschiebt und infolge dessen auch die zugehörigen Daten verschoben werden müssen, können sich daraus Performanceeinbußen ergeben. Ein 'moderner' Scheduler muss daher bei seinen Entscheidungen der Speicher- bzw. CPU-Topologie Beachtung schenken[13].

On-Die-Cache

Mit auf dem *Die* untergebrachter Cache-Speicher.

Out-of-Order

Die Out-of-Order-Ausführungs-Logik innerhalb eines Prozessorkernes verwaltet mehrere Buffer, mit deren Hilfe durch Umstellen der ursprünglichen Ausführungsfolge der Instruktionen versucht wird, Abhängigkeiten zwischen den Instruktionen zu minimieren und damit ein reibungsloseres Abarbeiten der Instruction-Queue zu erreichen. Stehen beispielsweise zwei voneinander abhängige Befehle, gefolgt von untereinander nicht abhängigen Befehlen, direkt hintereinander, ist es sinnvoll, die Befehle ohne Abhängigkeiten

zwischen den beiden voneinander abhängigen Befehlen auszuführen.

Process-Unit

Process-Unit ist der Oberberiff für eine für das Audio-Processing abzuarbeitende Einheit. Dazu zählen neben den eigentlichen Effekten oder virtuellen Instrumenten auch die Barrieren für die Synchronisierung oder die Verarbeitung von Eingangssignalen.

Prozess

Ein Prozess ist in der Regel ein eigenständig ausführbares Programm, das praktisch aus beliebig vielen Threads bestehen kann. Prozesse können im Gegensatz zu Threads auch räumlich getrennt auf zwei unterschiedlichen Rechnern laufen und beispielsweise über ein Netzwerk miteinander kommunizieren. Threads, die innerhalb eines Prozesses ausgeführt werden 'erben' die Eigentschaften des Prozesses wie Beispielsweise die Prozess-Affinität..

RAM

Random Access Memory, Hauptspeicher oder Arbeitsspeicher; Größter aber auch langsamster Speicher auf den die CPU direkt zugreifen kann.

Ratio

Verhältnis.

Reorder Buffer

Im Reorder Buffer (Umordnungspuffer) werden Mikro-Operationen zwischengespeichert und gegebenenfalls umsortiert.

Reservation Station

Buffer, der zur Ausführung bereitstehende Mikro-Operationen zwischenspeichert bis eine der Execution Units für die Ausführung bereit ist.

Routing

Für das Audio-Processing: Signalweg; der Verlauf, dem die Audio-Signale innerhalb eines Projektes vom Eingang zum Ausgang folgen..

SDRAM

Synchronous *DRAM*. Im Gegensatz zu asynchron gesteuertem *DRAM* reagiert SDRAM taktgesteuert. Mit Hilfe einer Pipeline kann der Speicher in jedem Taktzyklus eine Speicheranforderung verarbeiten, dadurch verkürzt sich die Wartezeit auf die Reaktion des Speichers vor der nächsten Speicheranforderung.

SMP

Symmetric Multiprocessing, Computer-Architektur, bei der alle CPUs als gleichberechtigt betrachtet werden. Im Gegensatz zur NUMA-Architektur hat das System keine Informationen über die Topologie der Prozessoren, was bei größer werdender Anzahl von

CPU-Kernen zu zunehmend inneffektiven Speicherzugriffen führen kann. Beispiel: CPU1 und CPU2 haben gegenseitig direkten Zugriff auf den L1 Cache und CPU3 und CPU4 haben eine schnelle gegenseitige Verbindung. Auf einem SMP-System würde ein Scheduler keinen Unterschied machen, wenn ein Thread von einer CPU auf die andere verschoben wird. Auf einem NUMA-System würden das bevorzugt nur zwischen CPU3 und CPU4 oder CPU1 und CPU2 passieren.

SRAM

Statischer RAM, aus FlipFlop-ähnlichen Schaltungen aufgebaut. Er behält seinen Zustand solange die Betriebsspannung anliegt und benötigt daher keine Refresh-Zyklen. Sehr schnelle, aber auch teure Speichertechnik, die aus diesem Grund meistens nur in Cache-Speichern zu finden ist.

Sample

Im Zusammenhang mit Audio-Anwendungen werden Soundschnipsel für die Verwendung in Projekten oder als Grundlage für virtuelle Instrumente häufig als Samples bezeichnet. Im Zusammenhang mit dem Sample-Buffer sind hier aber die Werte gemeint, die für die Audio-Ausgabe berechnet wurden und in den Ausgangs-Buffer das Audio-Interfaces geschrieben werden.

Scheduler

Scheduler planen die Befehlsausführung und legen sie fest. Ein Scheduler auf Betriebssystemebene legt fest, wann welcher Thread zur Ausführung kommt. Ein Scheduler innerhalb einer CPU steuert die Ausführung von Micro-Operationen.

Stall-Cycles

Warte-Zyklen oder -Takte die die CPU einlegen muss, wenn Sie auf Daten oder Instruktionen aus dem Speicher warten muss.

Thread-Prioritäten

Die Priorität eines Threads ist ein Hinweis an den Scheduler des Betriebssystems mit welcher Priorität ein Thread auszuführen ist. Ist ein CPU-Kern beispielsweise gerade mit der Abarbeitung eines niedrig priorisierten Threads beschäftigt, kann der Scheduler des Betriebssystems dessen Ausführung für einen höher priorisierten Thread zurückstellen.

Thread

Ausführungspfad innerhalb eines *Prozesses*. Innerhalb eines Prozesses können mehrere Threads parallel ausgeführt werden. Beispielsweise kann ein Thread eine Audio-Datei abspielen, während ein anderer Thread auf Benutzereingaben wartet.

VST-Preloader

Zum VST-Preloader gehören einige Hintergrundthreads, die Daten aus Audio-Dateien oder Streams für das Processing cachen.

Wort

eine Gruppe meherer Bits oder Bytes als Informationseinheit wie beispielsweise ein 32-Bit Wort in einem 32-Bit Computer.

Xeon

1998 eingeführte Server-Prozessoren Serie von Intel.

debuggen

siehe *Debugger*.

false sharing

Ineffiziente Speichernutzung zwischen zwei CPUs. Bearbeiten beispielsweise zwei CPUs ständig den selben Speicherbereich, wird es dadurch häufig zu Wartezyklen kommen, weil eine der CPUs auf Daten warten muss, die von der anderen CPU gerade bearbeitet werden. Durch effizientere Programmierung kann der Code in so einem Fall eventuell erheblich beschleunigt werden.

halt

Der Assemblerbefehl Halt (HLT) veranlasst die CPU in den HALT-Status zu wechseln bis ein Interrupt oder ein Reset-Signal empfangen wird. Im HALT-Status werden einige Teile eines CPU-Kernes deaktiviert, was eine Reduzierung des Energiebedarfs und der Hitzeentwicklung zur Folge hat.

x86

x86 bezeichnet den Befehlssatz der von Intel 1978 mit dem 8086 und 8088 Prozessoren eingeführten Mikroprozessor-Architektur. Bis heute sind alle als x86-Prozessoren bezeichneten Microprozessoren zu diesem Befehlssatz abwärtskompatibel.

Abbildungsverzeichnis

Literatur

[1] Nakov, O./Stojchev, S.:Research on a System performance with parallel modes of execution in a multiprocessor computer system with hyper-Threading Technology, International Conference on Computer Systems and Technologies, CompSysTech 2004

[2] Windows Support for Hyper-Threading Technology, Windows Platform Design Notes, 2002
`http://www.microsoft.com/whdc/system/sysinternals/ht-windows.mspx`

[3] Steinberg Audio Streaming Input Output Specification, Development Kit 2.2, Document Release #2, Steinberg Media Technologies GmbH

[4] Oberschelp, W/Vossen, G.: Rechneraufbau und Rechnerstrukturen, Oldenbourg Verlag München Wien, 10. Auflage, 2006

[5] Hennessy J. L./Patterson D. A.:Rechnerarchitektur, Analyse, Entwurf, Implementierung, Bewertung, Vieweg Lehrbuch Informatik, 1990

[6] Hinton, G./Sager, D./Upton, M./Boggs, D./Carmen D./Kyker A./Roussel P.: The Microarchitecture of the Intel Pentium 4 Processor, Intel Technology Journal, 2001

[7] Magro, W./Peterson, P./Shah S.: Hyper-Threading Technology: Impact on Compute-Intensive Workloads, Intel Technology Journal Volume 6 Issue 1, 2002

[8] Intel 64 and IA-32 Architectures Software Developers Manual Volume 1, Chapter 2.2.7, 2009
`http://www.intel.com/products/processor/manuals/`

[9] Gochman, S./Ronen, R./Anati, I./Berkovits, A./Kurts, T./Naveh, A./Saeed, A./Sperber, Z./Valentine, R. C.: The Intel® Pentium® M Processor: Microarchitecture and Performance, Intel® Technology Journal Volume 7 Issue 2, 2003

[10] Using Intel® VTune Performance Analyzer to Optimize Software on Intel® Core i7 Processors, PAT Tools TCE, Software and Services Group, Intel, 2009

[11] Swaminathan, J.: Multi-Core Programming, Intel Software Solutions Group

[12] Hinton, G./Sager, D./Upton, M./Boggs, D./Carmean, D./Kyker, A./Roussel, P.: The Microarchitecture of the Pentium® 4 Processor, Intel® Technology Journal Q1, 2001 `software.intel.com/file/6564`

[13] Siddha, S./Pallipadi, V./Mallick, A.: Process Scheduling Challenges in the Era of Multi-Core Processors, Intel® Technology Journal Volume 11 Issue 04, 2007

[14] Intel® 64 and IA-32 Architectures Optimization Reference Manual, Intel, 2009

[15] Waltman, J.: Intel Pentium 4 and NetBurst Micro-Architecture, MS University of Utah

[16] Tanenbaum, A. S.: Computerarchitektur, Strukturen - Konzepte - Grundlagen, Prentice Hall, 5. Auflage, 2006

[17] Van der Pas, R.:Memory Hierarchy in Cache-Based Systems
http://www.sun.com/blueprints/

[18] Advanced Micro Devices, Inc.: A Guide to Memory Timing
http://www.amd.com/us-en/Processors/ComputingSolutions/0,,30_288_13265_13295%5E13335,00.html

[19] High Performance Reduced Intrusction Set Computers, IBM T.J, Watson Research Center Technical Report RC12434, 1987

[20] Pessach, Y., Make It Snappy - Juice Up Your App with the Power of Hyper-Threading, MSDN-Magazine 2009
http://msdn.microsoft.com/en-us/magazine/cc300701.aspx

[21] The Return Of Hyper-Threading
http://www.tomshardware.com/reviews/Intel-i7-nehalem-cpu,2041-5.html

[22] Datenblatt: Corsair Dominator TWIN2X2048-8500C5D
http://www.corsair.com/_datasheets/TWIN2X2048-8500C5D.pdf

[23] Intel® VTune Performance Analyzer 9.1, Intel Corporation, 2008 http://software.intel.com/en-us/intel-vtune/

[24] Intel® Thread Profiler, Intel Corporation, 2008 http://software.intel.com/en-us/intel-vtune/

[25] CPU-Z, CPUID x86 technical resources
urlhttp://www.cpuid.com/

*Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 17. Juni 2009	
Ort, Datum	Unterschrift