Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Prof. Dr. Hermann Härtig

TECHNISCHE UNIVERSITÄT DRESDEN

Diplomarbeit

zum Thema

Migration Sweetspots

Konstantin Tabere
Geboren am 12.08.1985 in Leningrad
Matrikel-Nr. 3206798

Betreuer:	Dipl.-Inf. Benjamin Engel
	Dipl.-Inf. Michael Roitzsch
betreuender Hochschullehrer:	Prof. Dr. Hermann Härtig

Eingereicht bei der Professur für Betriebssysteme am 30. Oktober 2013.

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 30. Oktober 2013

Inhaltsverzeichnis

I Abbildungsverzeichnis

II Tabellenverzeichnis

1 Einleitung

Jedem Informatiker ist das Moor'sche Gesetz ein Begriff. Die Anzahl der Transistoren auf einem Chip verdoppelt sich alle zwei Jahre. Diese wegweisende Behauptung hatte Gordon Moore im Jahre 1965 aufgestellt und wenn man sich die Entwicklung der Prozessoren der letzten Jahrzehnte anschaut, so gilt das Gesetz noch bis heute [Col13]. Für die Einhaltung der Gesetzmäßigkeit sind vor allem zwei wesentliche Entwicklungen verantwortlich. Zum einen ermöglicht der Fortschritt bei der Reduzierung der Strukturgrößen in integrierten Schaltkreisen immer mehr Transistoren auf einen Chip unterzubringen. Zum anderen gibt es mit der Vermehrung der auf einem Prozessor vorhandenen CPUs eine sinnvolle Möglichkeit mit mehr verbauten Kernen immer leistungsstärkere Prozessorarchitekturen zu schaffen.

Diese Entwicklung stellt nicht nur Softwareentwickler in der ganzen Welt vor neue Herausforderungen. Wie kann man das große Rechenpotential eines modernen Prozessors mit mehreren CPUs intelligent nutzen? Auch aktuelle Betriebssysteme müssen für den Umgang mit Mehrkernprozessoren angepasst und weiterentwickelt werden. Besonders der Scheduler, der für die Verteilung der auf einem System vorhanden Rechenzeit an Anwendungen, die ausgeführt werden, zuständig ist, muss optimal arbeiten. Um den maximalen Durchsatz zu gewährleisten ist es wichtig, dass Anwendungen auf allen zur Verfügung stehenden CPUs verteilt ausgeführt werden. Für diesen Zweck führt der Linux-Scheduler in regelmäßigen Abständen einen Lastausgleich durch. Dabei werden ein Teil der Anwendungen, die auf einer CPU mit hoher Last ausgeführt werden auf andere CPUs mit weniger Last migriert. So wird die Last verteilt und fortan erhalten alle Anwendungen mehr Rechenzeit und werden schneller abgearbeitet.

Eine Migration einer Anwendung zu einer anderen CPU kann sich allerdings auch negativ auswirken. Dies ist der Fall, wenn die Daten mit denen die Anwendung arbeitet vorrangig in CPU-nahen Zwischenspeichern – den Caches – stehen, auf die nach der Migration nicht mehr oder nur deutlich langsamer als vor der Migration zugegriffen werden kann. Alle modernen Prozessoren besitzen mehrere Cache-Ebenen, die über unterschiedliche Kapazitäten und Zugriffszeiten verfügen. Bei vielen aktuellen Prozessoren sind die untersten beiden Cache-Ebenen (L1- und L2-Cache) exklusiv für die CPU eines Kerns nutzbar. Möchte die CPU eines anderen Kerns nach der Migration

einer Anwendung auf Daten zugreifen, die im L1- oder L2-Cache des zuletzt benutzten Kerns stehen, kommt es zu Verzögerungen.

An dieser Stelle soll diese Arbeit eine Methode zur Minimierung dieser ungewollten Verzögerungen vorstellen. Mit Hilfe von Performance-Countern sollen unter dem Betriebssystem Linux zur Laufzeit von Anwendungen Zeitpunkte gefunden werden, an denen eine Migration zu einer anderen CPU so gewählt wird, dass es zu weniger Verzögerungen beim Zugriff auf Daten kommt.

Diese Zeitpunkte werden unter dem Begriff Migration Sweetspots zusammengefasst und sind das zentrale Thema dieser Arbeit. Die Analyse, Implementierung und eine Evaluierung eines Linux-Kerns, der Migration Sweetspots erkennen kann, werden beschrieben.

Die weitere Arbeit gliedert sich in die folgenden fünf Kapitel: Im Kapitel 2 werden die Grundlagen für das allgemeine Verständnis der Arbeit gelegt. Da das Themengebiet ein umfassendes Wissen über Rechenarchitektur und den Aufbau von Betriebssystemen erfordert, werden die wesentlichen Aspekte des Schedulings, Begrifflichkeiten und Funktionsweisen von Betriebssystemfunktionen in Linux beschrieben.

Mit Hilfe von Performance-Countern wird in Kapitel 3 der Entwurf einer Technik beschrieben, wie durch ein spezielles Profiling von Prozessen Migration Sweetspots gefunden werden können. Die Implementierung dieser Technik im Linux-Kern und die Probleme, die während der Programmierung der neu implementierten Funktionen gelöst werden mussten, werden im Kapitel 4 vorgestellt.

Eine Evaluierung und Überprüfung der Funktionalität mit Hilfe ausgewählter Programme eines Mikro Benchmarks wird in dem Kapitel 5 durchgeführt. Kapitel 6 fasst die gesammelten Ergebnisse zusammen und gibt einen Ausblick über weiterführende Arbeiten.

2 Grundlagen

Als Einleitung für die weitere Arbeit werden in diesem Kapitel die Grundlagen, die zum besseren Verständnis der folgenden Kapitel dienen, erläutert.

Zunächst werden im Abschnitt 2.1 die Begriffe Prozess und Thread, im Sinne eines Betriebssystems eingeführt. Abschnitt 2.2 erklärt aufbauend für die nachfolgenden Abschnitte die Funktion von Systemaufrufen. Der Abschnitt 2.3 befasst sich mit dem Scheduler und dazugehörige Systemkomponenten im Linux-Kern. Im Abschnitt 2.4 wird die Architektur moderner x86-Mehrkernprozessoren und die Speicherhierarchie erklärt. Der Aufbau und die Nutzung von Hardware Performance-Countern wird im Abschnitt 2.5 betrachtet.

Abschnitt 2.6 beschreibt anschließend, wie Performance-Counter im Linux-Kern unterstützt werden und welche Schnittstellen Kernelentwicklern zur Verfügung stehen. Im Abschnitt 2.7 werden einige generelle Hinweise zur Linux Kernelentwicklung zusammengefasst. Ein Blick auf verwandte Arbeiten wird im Abschnitt 2.8 geworfen.

2.1 Prozesse und Threads

Zu den Aufgaben eines Betriebssystems gehört es geteilte Ressourcen eines Computersystems auf Anwendungsprogramme (kurz Anwendungen) aufzuteilen. Jede Anwendung wird dabei auf Betriebssystemebene als ein Prozess repräsentiert. Alle Prozesse besitzen einen eigenen Adressraum, mindestens einen Thread und eine Menge an zugewiesenen Ressourcen. Es ist dabei unerheblich, ob der Prozessstart durch eine vom Benutzer getätigte Aktion ausgelöst (Benutzer-Prozesse) wird, oder das Betriebssystem selber (Kernel-Prozesse) den Prozessstart veranlasst hat.

Threads sind die Ausführungseinheiten einer Anwendung. Jeder Prozess besitzt nach dem Start einen Hauptthread, welcher den Start mehrerer weiterer Threads veranlassen kann. Unter dem Ausführungskontext eines Threads versteht man die Attribute und Eigenschaften eines Threads, die einen Thread einem Prozess zuordnen. Die Threads eines Prozesses haben einen gemeinsamen Ausführungskontext, einen eigenen Stack, einen Befehlszähler, eigene Register für lokale Variablen und laufen zunächst unabhängig voneinander auf einer zur Verfügung stehenden CPU (central processing unit). Bei

einem System mit einem Einkernprozessor teilen sich alle Threads eine CPU. Bei einem Mehrkernprozessorsystem gehört es zur Aufgabe des Betriebssystems alle Threads entsprechend einer Scheduling-Policy auf die Anzahl des vorhandenen CPUs zu verteilen.

Allgemein gibt es zunächst nur drei Zustände in denen sich ein Thread befinden kann: bereit, blockiert oder laufend. Ein neu erzeugter Thread beginnt seinen Lebenszyklus immer im bereit-Zustand. Alle Zustandsübergänge eines Threads werden durch den Scheduler geregelt. Dieser implementiert einen Algorithmus, der aus der Menge der bereiten Threads einen auswählt und einer CPU zuweist. Dann befindet sich der Thread im laufend-Zustand und die Befehlsinstruktionen des Threads werden von der CPU abgearbeitet. Der Zustand bleibt erhalten, bis der Thread einen Systemaufruf (siehe Abschnitt 2.2) tätigt, der das Abarbeiten weiterer Befehlsinstruktionen zunächst verhindert. Dies passiert typischerweise beim Lese- oder Schreibzugriff von Daten der externen Festplatte. Dann muss der Thread auf ein bestimmtes Ereignis warten – in dem Fall eines Lesezugriffs auf das Eintreffen der Daten, bis er weiterlaufen kann. Dieser Thread-Zustand heißt blockiert-Zustand oder anders ausgedrückt: Der Thread ist blockiert. Sobald das Ereignis auf das gewartet wird eingetreten ist, wechselt der Scheduler den Zustand des Threads wieder in den bereit-Zustand.

Zusätzlich ist ein Übergang vom laufend-Zustand zum blockiert-Zustand möglich, wenn Interrupts, d.h. Systemunterbrechungen auftreten, die das Ausführen des gerade laufenden Threads verhindern. Ein Interrupt kann auch auftreten, wenn ein Thread schon längere Zeit auf einer CPU gelaufen ist und der Scheduler einen anderen bereiten Thread höher priorisiert einstuft, als den zur Zeit laufenden Thread.

Der Linux-Kern implementiert Threads als leichtgewichtige Prozesse. Diese können sich einige Ressourcen teilen, beispielsweise den Adressraum und offene Dateien. Jedoch besitzt jeder leichtgewichtige Prozess einen eigenen Ausführungskontext und wird als ein unabhängiger Prozess ausgeführt. Der Vorteil liegt auf der Hand. Bei „normalen“ Threads blockiert der gesamte Prozess und damit alle Threads des Prozesses, sobald ein Thread in den blockiert-Zustand wechselt. Dies passiert bei leichtgewichtigen Prozessen nicht. Hier können mehrere leichtgewichtige Prozesse mit gemeinsamen Ressourcen (im weitesten Sinne Threads einer Anwendung) blockieren oder schlafen, ohne dass andere Prozesse davon betroffen werden. Mit dem Systemaufruf `fork()` wird im Linux-Kern ein neuer (leichtgewichtiger) Prozess (Kindsprozess) erstellt, der zunächst eine fast vollständige Kopie des Prozesses darstellt, der `fork()` aufgerufen hat (Elternprozess). Alle Ressourcen werden zunächst gemeinsam genutzt, doch sobald während der Ausführung beispielsweise Änderungen an Daten im Speicher festgeschrieben werden müssen, wird per `Copy-on-write` der Speicher tatsächlich

aufgeteilt und der Kindsprozess bekommt einen eigenen neuen Speicherbereich. Führt ein Kindsprozess mit `exec()` ein neues Programm aus, wird die Aufteilung aller Ressourcen herbeigeführt. Der Kindsprozess ist dann komplett vom Elternprozess getrennt. Der Systemaufruf `clone()`erzeugt einen neuen Prozess unter Angabe der von Kinds- und Elternprozess fortan gemeinsam genutzten Ressourcen. Die Begriffe Thread, Prozess oder gelegentlich auch Task werden bei Linux als Synonyme verwendet und beschreiben alle leichtgewichtige Prozesse. Weitere Details zu leichtgewichtigen Prozessen und deren Implementierung im Linux-Kern folgen im Kapitel 4.

2.2 Systemaufrufe

Grundsätzlich unterscheiden moderne Prozessoren und Betriebssysteme verschiedene Betriebsmodi der CPU unter denen Prozesse ausgeführt werden. Es wird zwischen dem Kernel-Modus (Ring 0) und den Benutzer-Modi (Ring 1–3) unterschieden. Nur wenn Prozesse im Kernel-Modus ausgeführt werden, haben sie Zugriff auf den kompletten Befehlssatz der CPU. Laufen sie stattdessen im weniger privilegierten Benutzer-Modus, so steht nur ein reduzierter Befehlssatz aus Sicherheitsgründen zur Verfügung.

Diese Einteilung ist notwendig, damit die Sicherheit eines Systems gewährleistet werden kann. Wenn Benutzer-Prozesse einige für die Systemstabilität kritische Aufgaben versuchen selbst zu erledigen, ohne Rücksicht auf andere Prozesse zu nehmen, so droht Datenverlust oder Fehlfunktionen sind die Folge.

Ein Beispiel wäre hier ebenfalls der Zugriff auf Daten der Festplatte. Sobald ein Benutzer-Prozess Daten von der Festplatte lesen möchte, so teilt er dies durch einen Systemaufruf mit und es erfolgt ein Wechsel in den Kernel-Modus (Kerneintritt). Dann wird die Ausführung der Befehlsinstruktionen des Prozesses auf der CPU unterbrochen und der Befehlszähler wird gespeichert. Anschließend liest eine Kernel-Routine die Daten von der Festplatte aus und kopiert diese in den Adressbereich des Prozesses. Danach erfolgt wieder ein Wechsel des Betriebsmodus der CPU und diese schaltet in den Benutzer-Modus und der Prozess wird dort weiter ausgeführt, wo er unterbrochen wurde (Kernaustritt). Bei jedem Systemaufruf erfolgen demzufolge ein Kernein- und ein Kernaustritt.

2.3 Linux-Scheduler

Die Aufgabe des Scheduler ist es die auf einem System vorhandene Rechenzeit an Prozesse zu verteilen. Dabei sind mehrere Anforderungen zu berücksichtigen:

- Fairness – jeder Prozess bekommt einen fairen Anteil der Rechenzeit
- Policy Enforcement – vorgegebene Scheduling-Strategien werden durchgesetzt
- Balance – alle CPUs des Systems sind ausgelastet

Neben diesen drei Hauptanforderungen werden ja nach Einsatz des Betriebssystems weitere Ziele wichtig. Bei Stapelverarbeitungssystemen möchte man den Durchsatz (Anzahl der erledigten Jobs pro Zeiteinheit) maximieren und die Durchlaufzeit (Zeit vom Start bis zur Beendigung) minimieren. Bei interaktiven Systemen rückt die Antwortzeit (schnelle Antwort auf Anfragen) und die Proportionalität, also die Erfüllung der Erwartungen des Benutzers, in den Vordergrund. Eine weitere Umgebung sind Echtzeitsysteme, das heißt Systeme deren Prozesse bestimmte Deadlines einhalten müssen, um Datenverlust oder weitere kritische Systemfehler zu verhindern. Gleichzeitig muss der Scheduler in der Lage sein, Aussagen zur Vorhersagbarkeit zu treffen, um Qualitätseinbußen im System zu vermeiden. Aufgrund des breiten Einsatzes von Linux in fast allen dieser Umgebungen berücksichtigt der Linux-Scheduler viele der möglichen Szenarien. Es werden drei Klassen von Prozessen unterschieden.

Echtzeit-FIFO

Die höchste Priorität haben Echtzeit-FIFO-Prozesse (fifo = first in first out). Diese können von keinem anderen rechenbereiten Prozess unterbrochen werden, außer von einem anderen Echtzeit-FIFO-Prozess mit einer höheren Priorität. Linux-intern werden für Echtzeit-Prozesse die Prioritätsebenen von 0 bis 99 vergeben. Dabei hat 0 die höchste und 99 die niedrigste Priorität.

Echtzeit-Round-Robin

Ähnlich, wie bei Echtzeit-FIFO, verhält es sich mit den Echtzeit-Round-Robin-Prozessen. Der Unterschied ist der, dass diese nur für ein zugewiesenes Quantum an Rechenzeit ausgeführt werden und dann vom Scheduler unterbrochen werden. Anschließend werden sie an das Ende einer Liste aller Echtzeit-Round-Robin-Prozesse

gesetzt, aus der der nächste rechenbereite -Prozess ausgewählt wird. Hier werden auch die Prioritätsebenen von 0 bis 99 vergeben.

Timesharing

Alle „normalen“ Nicht-Echtzeit-Prozesse besitzen die Prioritätsebenen 100 bis 139. Insgesamt gibt es in Linux damit 140 verschiedene Prioritäten für Prozesse. Die Standardpriorität von neu gestarteten Benutzer-Prozessen liegt bei 120. Diese kann jedoch mit dem Systemaufruf `nice()` durch den Prozess selber, beispielsweise ausgelöst durch eine Benutzeraktion verändert werden. Erlaubte Werte für den `nice`-Wert reichen von -20 bis +19, womit die Priorität entsprechend der Grenzen für Nicht-Echtzeit-Prozesse verändert werden kann. Timesharing-Prozesse laufen ebenso, wie die Echtzeit-Round-Robin-Prozesses für ein vorgegebenes Quantum an Zeitintervallen. Diese Zeitintervalle sind festgelegt auf die Dauer von einer Millisekunde und werden im Linux-Kern `Jiffies` genannt. Für die Einhaltung ist im Scheduler ein Timer mit 1.000 Hz implementiert.

Für jede logische CPU wird im Linux-Scheduler eine Runqueue angelegt. Dies ist eine Datenstruktur, die unter anderem die Arrays `active` und `expired` der Länge 140 enthält, deren Elemente auf eine Liste mit Prozessen (Tasks) mit entsprechender Priorität zeigen. Alle bereiten oder laufenden Prozesse befinden sich zunächst im `active`-Array. Läuft die Zeitscheibe (Quantum) eines Tasks ab, so wird er in das `expired`-Array verschoben. Blockiert ein Task, so wird er aus dem `active`-Array entfernt und erst wieder hinein gesetzt, wenn er deblockiert wurde. In der Zwischenzeit befindet sich der Task in der `wait`-Queue. Sobald das `active`-Array leer ist werden die Zeiger auf das `active`- und `expired`-Array einfach getauscht und ein neuer Zyklus beginnt.

Jede Prioritätsebene erhält eine eigene Zeitscheibenlänge (die Länge der Zeitscheibe ist immer ist ein Vielfaches eines `Jiffy`) und Tasks mit höheren Prioritäten dürfen länger laufen als solche mit niedrigeren Prioritäten. Beispielsweise besitzt die Prioritätsebene 100 eine Zeitscheibe mit einer Gesamtlänge von 800 ms und ein Task mit Prioritätsebene 139 eine Zeitscheibenlänge von nur fünf ms. Je nach Verhalten der Prozesse und deren Zustand werden deren Prioritäten vom Scheduler dynamisch angepasst. So bekommen beispielsweise blockierte Prozesse, die auf das Lesen von Daten von der Festplatte warten, dynamisch eine höhere Priorität zugewiesen, um möglichst schnell weiterlaufen zu können, wenn die Daten angekommen sind.

Bei Mehrkernprozessoren findet neben der dynamischen Prioritätsanpassung in regelmäßigen Abständen noch ein Lastausgleich zwischen den CPUs statt. Dabei wird versucht die Menge an Tasks pro Runqueue möglichst gleichmäßig aufzuteilen, um sicherzustellen, dass die Systemlast ausgeglichen ist. Ein neuer Task wird bei Einplanung durch den Scheduler in die Runqueue der CPU mit der kleinsten momentanen Last platziert.

Lokale Affinitätsanforderungen der Tasks müssen vom Scheduler ebenfalls berücksichtigt werden. Einige Prozesse wollen selber festlegen, auf welchen CPUs sie ausgeführt werden möchten. Um die positiven Wirkungen eines „heißen“ Caches zu maximieren bleiben Tasks in der Regel auf einer CPU und wechseln nur in dem Fall, wenn eine andere CPU keine weiteren bereiten Tasks in der eigenen Runqueue besitzt. Solche Wechsel sind sehr teuer. Neben den üblichen Kosten für einen Kontextwechsel müssen noch alle Registerinhalte, Speichertabellen usw. auf die neue CPU kopiert werden. Zusätzlich müssen einige Daten für die neue CPU aus den CPU-nahen Caches (L1- und L2-Cache) der alten CPU heraus kopiert werden, was sich in einer verschlechterter Task-Laufzeit, auswirkt.

2.3.1 Kontextwechsel

Kontextwechsel treten auf, wenn der Scheduler die Ausführung eines Prozesses auf einer CPU beendet und anschließend einen anderen Prozess startet. Dabei wird der komplette Zustand des gerade noch laufenden Prozesses (Daten in Registern, Befehlszähler, Prozesscounter, Seitentabelle …) im Hauptspeicher zwischengespeichert.

Anschließend wird der Zustand des neuen Prozesses geladen und die CPU beginnt mit dessen Ausführung. Bei einem Kontextwechsel können Daten im TLB (translation lookaside buffer) ungültig werden und dieser muss unter Umständen geleert werden. In Abschnitt 2.4.1 wird näher auf den TLB eingegangen. Die Dauer eines Kontextwechsels schwankt je nach verwendeter Prozessor-Architektur und Betriebssystem zwischen 1300 ns und 1900 ns [Sig10].

2.4 Architektur moderner x86-64-Mehrkernprozessoren

Früher hatten Prozessoren nur einen Kern mit einer CPU verbaut. Seit 2005 gibt es bereits x86-Architekturen mit zwei Kernen in einem Prozessor auf dem Markt [amd04]. Heutige Prozessoren besitzen häufig mehr als nur zwei Kerne und jedes Jahr erscheinen neue Prozessoren mit immer mehr Kernen. Aktuelle Serverprozessoren der Ivy Bridge-EP Prozessorfamilie besitzen bis zu zwölf physische Kerne und mit aktiviertem Hyper-Threading entspricht das 24 logischen CPUs [int13b].

Hyper-Threading ist die Intel-Bezeichnung für Simultaneous Multithreading (SMT) – einer Vorstufe zu echten Mehrkernprozessoren. Dabei besitzt ein Kern zwar nur eine Recheneinheit (ALU und FPU), legt aber einige CPU-Ressourcen, wie Registersätze für beispielsweise Stackpointer und Programmcounter mehrfach aus. Dadurch können pro Kern mehrere Prozesse echt parallel ausgeführt werden, wenn sie auf Ressourcen zugreifen, die im gleichen Kern gerade nicht durch einen anderen Prozess auf einer anderen CPU belegt sind. Dazu werden die Instruktionsfolgen der Prozesse in kleinere Teile (Micro-Ops) zerlegt, die eine bessere Aufteilung auf mehrere logische CPUs ermöglichen. Im Multitasking-Betrieb kann diese Technik bis zu 20 Prozent mehr Performance bringen.

Heutige Prozessoren sind in der Regel alle 64 Bit-Architekturen. Zu den bekanntesten Vertretern gehören beispielsweise die Firmen Intel (IA-64 Architektur und Intel 64 Architektur), AMD (x86-64/AMD64 Architektur) und IBM (POWER). AMD64 ist vollständig kompatibel zu der 32-Bit-Architektur x86 und enthält einige zusätzliche Modifikationen, um mehr als vier GB Speicher adressieren zu können:

- 64-Bit physischer Adressraum
- Erweiterung der vorhandenen General-Purpose Register (GPR) auf 64 Bit
- weitere acht GPRs (R8 - R15)
- Erweiterung um acht SSE-Register (XMM8 - XMM15)
- 64 Bit-Befehlszeiger (RIP) mit relativer Adressierung

Mit Hilfe eines Kontroll-Bits LMA (long mode active) lässt sich die Betriebsart des Prozessors von 32 Bit auf 64 Bit zur Laufzeit ändern, was dem Prozessor ermöglicht auch unter einem 64-Bit-Betriebssystem 32-Bit-Anwendungen zu starten.

Neben den durch die Anwendungen selbst verwalteten Registern, die sehr klein sind und eine typische Kapazität von <1 KB besitzen, gibt es in heutigen Prozessorarchitek-

turen noch größere Speicher – die Caches. Diese werden hauptsächlich von der Hardware selber gesteuert und angesprochen. Die Größe der Caches kann auf aktuellen Chips mehr als zehn MB überschreiten (beispielsweise 30 MB L3-Cache beim Intel E5-2697 v2 Prozessor [int13b]). Diese Speicher werden als Zwischenspeicher für häufiger verwendete Daten aus dem Hauptspeicher genutzt, da ein Zugriff auf den Cache deutlich schneller ist, als ein Zugriff auf den Hauptspeicher selbst. In der Regel reichen für den Zugriff auf den L1-Cache 2-3 CPU-Takte aus.

Bei Prozessor-Architekturen mit Caches wird der Hauptspeicher in Blöcke von typischerweise 64 Byte Größe eingeteilt. Diese Größe entspricht der Breite einer Cache-Line. Die am häufigsten gebrauchten Speicherblöcke, auf die ein Prozess zugreift, werden in den L1-Cache kopiert. Reicht der Platz im L1-Cache nicht aus, wird eine Cache-Line in den L2-Cache verdrängt. Diese beiden Caches liegen sehr nahe an der CPU und sind meistens pro physischen Kern mehrfach auf einem Prozessor vorhanden. Eine Hierarchieebene höher liegt der L3-Cache und häufig wird dieser bei den heutigen Architekturen (Intel Core i7 und AMD Phenom II) von allen Prozessorkernen gemeinsam benutzt. Dies muss aber nicht immer der Fall sein. Der L3-Cache ist als Puffer für die aus dem L2-Cache verdrängten Cache-Lines aktiv und bildet die letzte Stufe der Cache-Hierarchie. Den letzten Cache vor dem Hauptspeicher nennt man auch Last Level Cache (LLC).

Wird ein Speicherwort von einem Programm gelesen, so überprüft die Puffer-Hardware, ob der entsprechende Block bereits in einem der Caches vorhanden ist. Wenn das der Fall ist, so spricht man von einem Cache-Hit. Befindet sich der Block nicht in dem Cache (Cache-Miss), so muss dieser aus dem Hauptspeicher geladen werden. Dabei erfolgen die Zugriffe auf den Cache und den Hauptspeicher gleichzeitig, um bei einen Cache-Miss die Verzögerungen des Speicherzugriffs minimal zu halten. Ein Zugriff auf einen Cache wird mit Cache-Reference bezeichnet. Cache-References können nur auftreten, wenn ein Cache-Miss in der darunter liegenden Hierarchie-Ebene stattgefunden hat. Ein Cache-Reference führt immer zu einem Cache-Hit oder zu einen Cache-Miss in der entsprechenden Hierarchie-Ebene. Das Verhältnis aus Cache-Hits zu Cache-References heißt Hit-Verhältnis und das Verhältnis aus Cache-Misses zu Cache-References analog dazu Miss-Verhältnis.

Man spricht von einem „kalten“ Cache, wenn Daten eines Prozess noch nicht im Cache stehen. Läuft ein Prozess längere Zeit auf einer CPU, so füllt sich der Cache nach und nach mit den Daten auf die der Prozess häufiger zugreift und der Cache wird „heiß“.

2.4.1 Paging und MMU

Um dem großen Speicherbedarf von einigen Programmen zu begegnen, unterstützen heutige x86-Prozessoren alle Paging. Das ist eine Technik um den virtuellen Adressraum für Prozesse zu erhöhen. Wenn in einem System für die Ausführung eines Prozesses nicht genügend physischer Hauptspeicher zur Verfügung steht, so kann mit Paging ein Teil der gerade nicht benutzten Daten im Hauptspeicher zeitweise auf die Festplatte ausgelagert werden. Virtuell stehen dem Programm somit mehr Speicheradressen und damit ein größerer Adressraum zur Verfügung.

Beim Paging wird der vorhandene physische Speicher in Einheiten fester Größe (typischerweise 512 Byte bis 64 KB) eingeteilt. Diese werden Seitenrahmen genannt. Seitenrahmen werden mit Seiten gefüllt. Seiten sind genauso groß, wie Seitenrahmen und bilden den virtuellen Adressraum ab. Bei knappem physischem Speicher werden Seiten aus ihrem Seitenrahmen verdrängt und auf die Festplatte ausgelagert.

Der Teil eines Computersystems, welcher virtuelle Adressen auf physische Adressen abbildet, nennt sich MMU (memory management unit). Dieser befindet sich in der Regel innerhalb der CPU. Um die Aufgabe der Adressübersetzung durchführen zu können, verwaltet die MMU eine Seitentabelle. Diese speichert zu jeder virtuellen Seite ab in welchem Seitenrahmen sie sich befindet oder ob die Seite ausgelagert wurde. Bei einem Speicherzugriff auf eine ausgelagerte Seite spricht man von einem Seitenfehler (page fault). Bei einem Seitenfehler ist es die Aufgabe des Betriebssystems einen wenig benutzten Seitenrahmen auszuwählen und dessen Seite auf der Festplatte zu sichern. Danach muss die angeforderte Seite in den frei gewordenen Seitenrahmen geladen werden und der Zugriff auf die Seite wird wiederholt. Um die Suche in der Seitentabelle zu beschleunigen wird je nach CPU-Architektur ein TLB verwendet. Der TLB ist ein kleiner mehrstufiger Assoziativspeicher, der häufig benutzte virtuelle Speicheradressen auf physische Speicheradressen abbildet. Damit lassen sich wiederholende Zugriffe auf die Seitentabelle einsparen.

Die meisten Cache-Architekturen nutzen virtuell adressierten Cache, da diese einfach schneller Cache-Hits liefern. Für physisch adressierten Cache muss die MMU zwingend zwischen CPU und Cache liegen. Da diese aber für die Adressübersetzung mehrere Speicherzugriffe benötigt, verlangsamt das die Suche im Cache erheblich.

2.5 Performance-Counter

Eine wiederkehrende Frage in der Softwareentwicklung ist: Welche Leistung erreicht ein Programm oder Algorithmus auf einer vorgegebenen Hardware? Besonders bei sehr teuren Supercomputern, wo jeder unnötige CPU-Takt eingespart werden muss und Forscherteams sehr viel Zeit in die Optimierung ihres Codes stecken, um das Maximum an Performance (performance = Leistung) aus einem System herauszuholen, ist dieses zentrale Thema von großer Bedeutung.

Moderne x86-Prozessen besitzen schon seit mehreren Generationen Performance-Counter. Das sind zusätzliche Logikeinheiten, die direkt als Schaltkreise in der Hardware integriert sind, um hardwarenahe Operationen oder Ereignisse (events) verfolgen zu können. Solche CPU-Einheiten werden performance monitoring units (PMU) genannt und sind bei Intel seit Einführung der ersten Pentium-Prozessoren auf allen x86-Prozessoren vorhanden. Die Register zur Steuerung und zum Auslesen der Performance-Counter sind bei x86-Prozessoren im MSR (model-specific register) untergebracht. Nicht alle Prozessoren besitzen die gleiche Menge an Performance-Countern. Deshalb spricht man im Allgemeinen von modellspezifischen Performance-Countern.

Es gibt zwei Arten von messbaren Größen für Performance-Counter. Entweder man zählt die Anzahl des Auftretens von Ereignissen oder es wird die Dauer eines fest definierten Zustands in Prozessortakten gezählt.

Performance-Counter können genutzt werden, um zum Beispiel die Gesamtanzahl der ausgeführten CPU-Operationen, die Anzahl der durchgeführten Fließkommaoperation oder die Anzahl an Operationen aus dem erweiterten Befehlssatz SSE oder MMX zu zählen. Auf Prozessoren mit integrierten Caches gibt es weitere Anwendungsfälle. So lassen sich beispielsweise auch die absolute Anzahl der Cache-Misses bzw. Cache-References zur Laufzeit eines Programms verfolgen. Je kleiner das Miss-Verhältnis ausfällt, desto „besser“ läuft ein Programm und man kann Rückschlüsse auf die Performance ziehen. Es gibt jedoch auch Anwendungsfälle, wo genau das Gegenteil gewünscht ist. Dann werden Daten gezielt am Cache vorbeigeschrieben, um dessen Inhalt nicht zu gefährden und dadurch die Performance zu steigern.

Der Zugriff auf die MSRs (Register im MSR) ist nur im Kernel-Modus erlaubt, was zur Folge hat, dass normale Benutzer-Prozesse auf Systemaufrufe oder andere Schnittstellen des Betriebssystems angewiesen sind, um auf die entsprechenden Register zugreifen zu können.

2.5.1 MSRs in x86-Intel-Prozessoren

Intel-Prozesseren verfügen über eine Reihe von Performance-Countern, die in unterschiedliche Versionen eingeteilt sind. Ob und wie viele Versionen ein bestimmter Prozessor besitzt, lässt sich über die Instruktion `cpuid` auslesen [int11]. In der Version der ersten Generation „architectural performance monitoring version 1“ stehen je nach Modell mindestens zwei programmierbare Performance-Counter zur Verfügung. Um einen Performance-Counter zu starten, muss vorher über ein spezielles Auswahlregister (performance event select register `IA32_PERFEVTSELx` MSR) die Auswahl der zu messenden Kenngröße (performance event) festgelegt werden. Zu jedem Performance-Counter gehören genau ein Auswahlregister und ein Zählregister (`IA32_PMCx` MSR). Die Bitbreite der Zählregister ist Architektur- und Modell-pezifisch. Die Register-Adressen sind jedoch über alle Prozessor-Modellreihen konstant. Bei der Programmierung der MSRs kann man angeben, ob der Counter nur im Kernel- und/oder auch im Benutzer-Modus mitzählen soll. Je nach ausgewählten performance event zählen die Counter Ereignisse, die nur einen Kern betreffen, oder aber es werden Ereignisse außerhalb eines Kerns gezählt (offcore).

Die zweite Version der Performance-Counter „architectural performance monitoring version 2“ besitzt ferner ein globales Kontrollregister (`IA32_PERF_GLOBAL_CTR`), das Steuersignale für alle Performance-Counter bereithält, um ein einfacheres Ansteuern der Performance-Counter zu ermöglichen. Zusätzlich stehen drei nicht programmierbare Performance-Counter (`IA32_FIXED_CTR0` – `IA32_FIXED_CTR2`) mit einer fixen Funktion zur Verfügung, die ebenfalls über das globale Kontrollregister gesteuert werden können. Für das Verfolgen von Überläufen in den Zählregistern steht ein globales Statusregister (`IA32_PERF_GLOBAL_STATUS`) zur Verfügung.

Ab der dritten Version der Performance-Counter „architectural performance monitoring version 3“ werden logische CPUs und Hyper-Threading in Prozessor-Kernen unterstützt. Prozessoren der aktuellen Core-i7-Generation verfügen über acht programmierbare MSRs pro Kern, die gleichmäßig auf die vorhandenen CPU-Threads aufgeteilt werden. Das ergibt bei aktiviertem Hyper-Threading vier programmierbare Performance-Counter pro logische CPU. Wird Hyper-Threading im Prozessor deaktiviert oder generell nicht unterstützt (beispielsweise bei der Intel Core 2 Modell-Reihe), so stehen bis zu acht programmierbare Performance-Counter pro logische CPU zur Verfügung. Die Bitbreite der Zählregister der Performance-Counter beträgt bei Prozessoren der Intel Core-i-Baureihe 48 Bit. Daraus ergibt sich für einen Zähler, der bei null

anfängt zu zählen das Erreichen des Überlaufs nach 2^{48} Takten, wenn jeder Takt den Zähler inkrementiert. Bei einer Taktfrequenz von beispielsweise drei GHz wäre die Überlaufsituation damit nach $2^{48}/(3\times10^{12}) \sim 93$ Sekunden erreicht. Ein Überlauf wird im Status-Register des Performance-Counters durch ein Bit angezeigt.

2.6 Performance-Counter in Linux

Die Unterstützung für Performance-Counter im Linux-Kern für die x86-Architektur wurde erstmals mit dem `perfctr`-Patch durch Mikael Pettersson von der Uppsala Universität im Jahre 1999 eingeführt. Damals musste man den Linux-Kern noch selber patchen und eine allgemeine Verfügbarkeit gab es nicht [Wea13].

Erst im Jahr 2009, nach der Einführung von perfmon2 – einem generischen Interface für die IA64-Itanium-Architektur, wurden Anstrengungen unternommen Performance-Counter Unterstützung direkt in den x86-Kernel einzupflegen. Ab dem Linux-Kern v2.6.31 wurde die durch Ingo Molnar und Thomas Gleixner entwickelte `perf_event`-Schnittstelle in den Linux-Kern aufgenommen. Die Schnittstelle bietet einige wichtige Funktionen und Vorteile:

- `sys_perf_event_open()`-Systemaufruf für das Konfigurieren von Performance-Countern durch Benutzer-Prozesse
- über die `perf_event_attr`-Struktur wird angegeben, welche Ereignisse von dem Performance-Counter erfasst und gezählt werden
- mit Hilfe von `ioctl()`-Aufrufen können Performance-Counter gestartet und gestoppt
- das Auslesen der Zähler der Performance-Counter erfolgt mit `read()`

Die meisten Linux-Distributionen bieten mit dem Programm `perf` ein vollständiges Performance-Analyse-Tool an, das mit der `perf_event`-Schnittstelle arbeitet.

2.6.1 Die perf_event-Schnittstelle im Linux-Kern

Der große Vorteil von `perf_event` ist die einheitliche Form beim Umgang mit Performance-Countern. Es wird von der konkreten Prozessor-Architektur abstrahiert

und damit eine generalisierte und standardisierte Form für die Konfiguration und das Auslesen der Zählerstände der Performance-Counter präsentiert. Mit `perf_event` übernimmt der Linux-Kern die Kontrolle über das Zuordnen von Ereignissen zu passenden Zählern. Damit ergeben sich auch Möglichkeiten für Zeit-Multiplexing. Das bedeutet, wenn gleichzeitig mehr Ereignisse verfolgt werden sollen, als Performance-Counter zur Verfügung stehen, so kann mit Hilfe von verkürzten Zeitfenstern Ereignis-Zähler nur teilweise aufgenommen werden und dann auf die Werte der Gesamtzeit extrapoliert werden. Je nach Ereignis und Prozess ist dies ein gewünschter Ansatz, kann natürlich aber auch hinderlich sein.

Da Performance-Counter nur Ereignisse pro Kern zählen gibt es bei einem System mit mehreren parallel laufenden Prozessen keine Garantie, dass die Zählerstände nur von einem Prozess verursacht wurden. Deshalb speichert `perf_event` die Zählerstände eines Prozesses bei jedem Kontextwechsel, um die tatsächliche Werte nur eines Prozesses zu ermitteln. Daraus ergibt sich ein gewisser Overhead in Form von bis zu 20% längeren Kontextwechselzeiten [Wea13]. Bei Prozessen mit vielen Kontextwechseln sollte das beim Auswerten der Messwerte immer berücksichtigt werden.

Neben den tatsächlich durch die Hardware verursachten Ereignissen bietet `perf_event` ebenso die Möglichkeit softwareseitig durch den Linux-Kern verursachte Ereignisse mitzuzählen. Dazu zählen zum Beispiel das Verfolgen der aufgetretenen Kontextwechsel oder die Anzahl der Seitenfehler beim Referenzieren des virtuellen Speichers. Diese Kenngrößen sind ebenfalls interessant, wenn man beispielsweise die Performance des Betriebssystems messen möchte.

2.7 Linux Kernel-Entwicklung und Kernel-Module

Linux ist eine freie Software und aufgrund der GPL (GNU Public License) ist der Quellcode des Kernels frei zugänglich [Fre07]. Es ist gänzlich unmöglich für einen einzelnen Menschen den kompletten Code eines solch komplexen Betriebssystems alleine zu schreiben und zu pflegen. Und obwohl der Quellcode der ersten Version von Linux 0.0.1 im Jahr 1991 noch komplett durch Linus Torvalds allen geschrieben wurde, arbeitet heute eine ganze Community aus mehreren hunderten Programmierern am Linux-Kern [Tor01].

Jeder Interessierte kann den vollständigen Quellcode aus dem Git-Repository herunterladen und Änderungen in Form von Patches einreichen. Jedoch entscheidet eine Handvoll von langjährig an der Entwicklung beteiligten Programmierern um Torvalds

herum, welche Änderungen letztendlich in den offiziellen stabilen Linux-Kern aufgenommen werden [lif13]. Das hindert aber Niemanden daran eigene Funktionen und Erweiterungen in den Linux-Kern für Testzwecke oder den eigenen Gebrauch einzubauen, denn der Kernel ist sehr gut dokumentiert und modular aufgebaut.

Der Quellcode des Kernels ist aufgeteilt auf mehrere tausend Dateien und kann mit dem gcc-Compiler kompiliert werden. Zu allen Bereichen gibt es Textdateien mit Anleitungen und Erklärungen zum Verständnis.

Obwohl der Linux-Kern ein monolithischer Kernel ist, kann dieser dynamisch Kernel-Binaries in Form von Kernel-Modulen laden und ausführen. Viele Teile des Kernels lassen sich in Kernel-Module auslagern, wenn diese nicht dauerhaft für Funktionen des Betriebssystems genutzt werden. Durch eine Konfigurationsdatei lässt sich festlegen, welche Teile des Kerns als Kernel-Modul zur Verfügung stehen sollen.

Es gibt einige Einschränkungen, die man als C-Entwickler beim Arbeiten mit dem Linux-Kern beachten muss und die für die Implementierung von neuen Kernel-Funktionen nicht außer Acht gelassen werden dürfen [Lov05]:

- Der Kernel bindet keine Standard-C-Bibliothek ein, womit viele übliche C-Funktionen nicht zur Verfügung stehen.
- Es gibt keinen Speicherzugriffsschutz, wie bei Benutzer-Prozessen. Bei einem illegalen Speicherzugriff im Kernel kann ein kritischer Kernelfehler Oops auftreten.
- Es werden keine Fließkommazahlen im Kernel unterstützt, da die Umschaltung von Integer- zu Floating-Point-Befehlssatz auf unterschiedlicher Hardware individuell geregelt ist.
- Der Kernel-Stack ist relativ klein und statisch und deshalb sollten größere Menge an statischer Speicherallokation vermieden werden.
- Im Kernel gibt es viele Ressourcen, die aufgrund von Präemptierung, asynchronen Unterbrechungen und mehrfachen Zugriff besonders geschützt sein müssen. Wettlaufsituationen (race condition) treten sehr häufig auf und müssen durch Maßnahmen, wie Semaphore, Locks oder Mutex strikt vermieden werden.

2.8 Verwandte Arbeiten

Um die Fähigkeiten zur Parallelität von Mehrkernprozessorsystemen besser auszunutzen, werden mehrere Ansätze verfolgt. Die meisten Arbeiten konzentrieren sich auf

Techniken und Methoden, um die Abarbeitung von Single-Thread Anwendungen zu beschleunigen (Speedup) [VIA05]. Da solche Anwendungen nicht darauf ausgelegt sind, mehrere CPUs eines Prozessors parallel zu benutzen, gibt es zwei Möglichkeiten. Ein trivialer Ansatz ist es die Anwendungen direkt anzupassen. Dazu muss der Quellcode allerdings vorliegen und eine Neukompilierung ist notwendig. Eine weitere besonders gut geeignete Technik, um speicherintensive Anwendungen zu beschleunigen, ist das Prefetching durch Hilfs-Threads [CSR99] [ZS01] [KLW04].

Für Prefetching mittels Hilfs-Threads werden spezielle Threads gestartet, die zukünftige Speicherzugriffe vorausberechnen und die demnächst benötigten Daten näher an die CPU - d.h. in einen CPU-nahen Cache - bringen, um potentielle Cache-Misses bei der Ausführung des Haupt-Threads der (Single-Thread) Anwendung zu verhindern. Für Prozessorarchitekturen mit SMT ist es sinnvoll Hilfs-Threads auf einer weiteren logischen CPU im gleichen Kern zu starten, auf dem in einer anderen logischen CPU der Haupt-Thread ausgeführt wird. Der Grund ist, dass alle logischen CPUs eines Kerns den Cache aller Cache-Ebenen teilen und damit die Daten bis in den L1-Cache vorgeladen werden können. Diese Methode setzt SMT voraus und ist damit nur bedingt anwendbar. Bei Mehrkernprozessorsystemen ohne SMT müssen Hilfs-Threads daher auf einem anderen Kern des Prozessors, als der Kern auf dem der Haupt-Thread ausgeführt wird, gestartet werden. Steht in einem solchen Fall ein geteilter inklusiver L3-Cache zur Verfügung, so sind die Daten immerhin in dem L3-Cache vorhanden und müssen nicht teuer aus dem Speicher geholt werden.

Gibt es keinen geteilten Cache, so kann man, wie in der Arbeit [KST11] vorgeschlagen wird, mittels Inter-core Prefetching mit einer Migration des Haupt-Threads zu dem Kern, auf dem der Hilfs-Threads ausgeführt wurde, trotzdem einen Speedup erzielen. Ausschlaggebend für die Performancesteigerung ist die Größe der durch den Hilfs-Thread vorgeladen Daten und die Zeit für einen Kontextwechsel bei einer Migration des Haupt-Threads. Es sollten nur so viele Daten vorgeladen werden, wie einem Kern an exklusiven Cache zur Verfügung steht. Ansonsten werden die vorgeladenen Cache-Lines noch vor der Migration des Haupt-Threads schon wieder verdrängt und stehen dem Haupt-Thread anschließend nicht zur Verfügung. Die direkten Migrationskosten sollten möglichst gering sein, damit das Migrieren des Haupt-Threads nicht zur kritischen Komponente bei der Ausführung wird.

Idealerweise sollte der Hilfs-Threads nur solche Daten vorladen, die nicht gleichzeitig vom Haupt-Thread benötigt werden, um Cache-Synchronisations-Problemen aus dem Weg zu gehen und den Datenverkehr zwischen den Caches so gering wie möglich zu halten. Lässt sich eine Anwendung in Ausführungsabschnitte, bei der für einen festen Zeitraum nur eine begrenzte Arbeitsmenge zu erwarten ist, die der exklusiven Cache-

Größe eines Kerns entspricht, so kann mittels Inter-Core Prefetching signifikante Performance-Gewinne erzielt werden [KST11]. Der Main-Thread kann auf den Daten der ersten Arbeitsmenge arbeiten, während der Hilfs-Thread die Daten der nächsten Arbeitsmenge in den Cache des anderen Kerns vorlädt. Sobald beide mit ihrer Arbeit fertig sind, migrieren beide Threads zum jeweils anderen Kern und setzen dort ihre Arbeit fort. In diesem Fall findet der Haupt-Thread einen komplett „heißen“ Cache vor und kann ohne Verzögerungen arbeiten. Der Hilfs-Thread lädt wieder die nächste Arbeitsmenge vor.

Die Herausforderung liegt in der Generierung der Hilfs-Threads. Diese müssen entweder von Hand erstellt, oder mit speziellen Compilern-Techniken dynamisch oder mit Unterstützung von Profilingwerkzeugen generiert werden. Eine andere Möglichkeit ist die Generierung von Hilfs-Threads komplett in Hardware. Auf heutigen x86-Prozessoren ist dies bisher nicht möglich.

2.8.1 Vergleich Inter-Core Prefetching mit Migration Sweetspots

Inter-core Prefetching ermöglicht einen Speedup von Single-Thread Anwendungen, durch Ausnutzung von freien Rechenkapazitäten in Kernen, die sonst ungenutzt bleiben würden. In Überlastsituationen, wenn mehrere Anwendungen parallel laufen, wird allerdings durch die zusätzliche Rechenlast für die Ausführung der Hilfs-Threads die Gesamtperformance des Systems beeinträchtigt. Auch kann die Ausführung von weiteren speicherintensiven Anwendungen, die den Großteil des Caches eines Kerns ebenfalls für sich beanspruchen dazu führen, dass der Haupt-Thread auf einem Kern keinen Nutzen durch die Arbeit des Hilfs-Threads erfährt. Der „heiße“ Cache wird sofort durch die speicherintensive Anwendung wieder zerstört.

Migration Sweetspots bieten den Vorteil, dass sie besonders in Überlastsituationen eine bessere Verteilung der Rechenkapazität ermöglichen. Anwendungen werden so lange auf einer CPU ausgeführt, bis eine Überlastsituation auf eben dieser entsteht. Bei voller Auslastung der CPU, kann ein speicherintensiver Prozess unter Minimierung der indirekten Migrationskosten an einem Migration Sweetspot auf eine weniger belastete CPU migrieren. Dadurch wird die Anwendung durch die Migration nicht sonderlich ausgebremst und kann auf der anderen CPU ungehindert schneller ausgeführt werden, als dies auf der alten CPU der Fall wäre.

Beide Verfahren können in jeweils unterschiedlichen Situationen angewendet werden. So lange keine Überlastsituation auf einer der für das Inter-core Prefetching genutzten CPUs durch das Scheduling von weiteren Prozessen eintritt, ist eine zusätzli-

che Suche nach Migration Sweetspots nicht nötig. Ist dies nicht mehr der Fall, sollte Inter-core Prefetching deaktiviert werden und der Scheduler könnte mittels Migration Sweetspots Anwendungen auf unterschiedliche CPUs verteilen und so die Überlast lösen und gleichzeitig für optimale Ausführungsbedingungen für alle beteiligten Prozesse sorgen.

3 Entwurf

In diesem Kapitel werden die Einzelheiten und Designentscheidungen zum Vorgehen zur Suche von Migration Sweetspots in Linux vorgestellt. Dazu erfolgt im ersten Abschnitt 3.1 die Definition und theoretische Betrachtung der Migrationskosten, die bei einer Migration eines Prozesses entstehen. Am Beispiel einer Matrixmultiplikation wird im Abschnitt 3.2 die positive Wirkung von Migration Sweetspots für die Miss-Rate nach der Migration eines Prozesses gezeigt.

Im Abschnitt 3.3 werden dann die Anforderungen für eine erfolgreiche Suche nach Migration Sweetspots zusammengetragen. Eine Strategie zur Implementierung der Suche nach Migration Sweetspots wird in Abschnitt 3.4 gewählt. Auf die Unterschiede zwischen einer Suche im Kernel- und Benutzer-Modus wird in Abschnitt 3.5 eingegangen. Der letzte Abschnitt 3.6 befasst sich mit den zeitlichen Schranken und Folgen für die Durchlaufzeit eines Prozesses bei einer Verzögerung der Migration, wegen einem Migration Sweetspot.

3.1 Theoretische Betrachtung

Zur Veranschaulichung des Sachverhalts der Migrationen von Prozessen, erfolgt zunächst eine Analyse in einem abstrahierten Computer-System mit nur zwei physischen Kernen mit jeweils einer CPU – CPU1 und CPU2.

Der in Linux implementierte Scheduler ab der Kernel-Version 2.6.23 ist ein Completely Fair Scheduler mit dynamischem Lastausgleich (load-balancing) [Aas05]. Das heißt, dass während der Laufzeit des Systems in regelmäßigen Abständen von n Zeiteinheiten ein Lastausgleich vorgenommen wird. Die Last einer CPU wird vom Scheduler in der Variable `cpu_load` als Attribut einer jeden Runqueue gespeichert. Bei jedem `scheduler_tick()`, das heißt einmal pro Jiffy-Intervall, erfolgt eine Neuberechnung der CPU-Last durch die Methode `rebalance_tick()`. Liegt der letzte Lastausgleich länger als n Zeiteinheiten zurück, so wird ein neuer Lastausgleich durchgeführt und ein neues Intervall beginnt. Um den Lastausgleich durchzuführen, laufen auf jeder CPU spezielle Kernel-Threads mit einer hohen Priorität. Befinden sich die CPUs bei

einem Lastausgleich bezüglich ihrer aktuellen CPU-Last in einem Ungleichgewicht, so werden Prozesse von einer Runqueue einer CPU mit hoher Last zu einer Runqueue einer CPU mit wenig Last verschoben. Dabei werden Prozesse aus dem `expired`-Array der Runqueue beginnend mit der niedrigsten Priorität bevorzugt verschoben.

Zur weiteren Analyse wird ein hypothetischer Fall angenommen, um die nachfolgenden Definitionen besser erklären zu können. Es wird eine Situation betrachtet bei der neben dem Kernel-Thread für den Lastausgleich und dem Leerlaufprozess nur ein weiterer Prozess auf dem System, dessen Laufzeit länger ist, als das Intervall für den Lastausgleich, läuft. Bei einem Lastausgleich wird angenommen, dass der Prozess von der CPU1 zu CPU2 migrieren wird. Die Zeit für die Migration ist Zeit die bei der Ausführung des Prozesses verloren gegangen ist. Diese Zeit soll direkte Migrationskosten genannt werden. Der Sachverhalt ist in Abbildung … dargestellt.

Direkte Migrationskosten

Unter direkten Migrationskosten versteht man die Zeit, die für die reine Migration eines Prozesses von einer CPU auf eine andere CPU vergeht. Während dieser Verzögerungszeit kann der Prozess nicht ausgeführt werden.

Direkte Migrationskosten entstehen beispielsweise nach einem Lastausgleich, wenn Prozesse einer Runqueue einer anderen CPU zugewiesen werden. Für aktiv von einer CPU ausgeführte Prozesse beinhalten die direkten Migrationskosten die Zeitdauer für den Eintritt in den Kern, die verstrichene Zeit für das Einplanen des Prozesses in den Scheduling-Plan auf der anderen CPU, einem anschließenden Kontextwechsel und schließlich die Zeitdauer für einen Kernelaustritt.

Die direkten Migrationskosten sind abhängig von dem verwendeten Betriebssystem und der zu Grunde liegenden Prozessorarchitektur eines Systems. Durch die direkten Migrationskosten verlängert sich die Durchlaufzeit eines Prozesses. Abhängig vom Zustand der Runqueue auf der CPU nach der Migration, kommt zu den direkten Migrationskosten noch eine weitere Verzögerung hinzu, bis ein Prozess auf der CPU nach der Migration wieder ausgeführt wird. Diese Verzögerung lässt sich durch den Scheduler zumindest annähernd vor der Migration abschätzen, indem die Runqueue in die der Prozess verschoben wird analysiert wird. Alle Prozesse im `active`-Array und alle Prozesse im `expired`-Array mit einer höheren Priorität werden vor dem Prozess ausgeführt werden, da Prozesse bei der Migration immer in das `expired`-Array verschoben werden. Wenn all diese Prozesse ihr Quantum komplett ausnutzen, ergibt das die maximale Verzögerung.

Zusätzlich zu den direkten Migrationskosten müssen noch die indirekten Migrationskosten berücksichtigt werden. Nach der Migration sind der L1- und der L2-Cache auf der anderen CPU für den Prozess kalt, da beide Caches üblicherweise über das MOESI-Protokoll synchronisiert sind und die Daten des Prozesses noch in den Caches der alten CPU liegen [AMD11]. Bei einem gemeinsam genutzten L3-Cache können Lesezugriffe auf nicht modifizierte Cache-Lines immerhin ohne Zeitverzögerung aus dem L3-Cache geladen werden.

Aufgrund der Folge eines Kontextwechsels nach einer Migration ist auch der TLB für den Prozess anschließend kalt. Dies hat die größte Auswirkung auf die Durchlaufzeit des Prozesses, da alle Seitenzugriffe auf der anderen CPU zunächst mehrere Speicherzugriffe auf die Seitentabelle nach sich ziehen.

Indirekte Migrationskosten

Bei indirekten Migrationskosten handelt es sich um die negativen zeitlichen Auswirkungen für die Durchlaufzeit eines Prozesses, die nach einer Migration des Prozesses auf eine andere CPU durch einen dort „kalten“ Cache verursacht werden.

Indirekte Migrationskosten können eine große Varianz aufweisen, abhängig von dem zum Migrationspunkt aktiven Prozesszustand. Die minimalen indirekten Migrationskosten ergeben sich für den Fall, dass ein Prozess nach der Migration auf keine Daten in den Cache-Lines vor der Migration zugreift. Nach einer Migration muss in jedem Fall auch der L1-Instruction-Cache auf der anderen CPU für den Prozess neu gefüllt werden und somit können die indirekten Migrationskosten nie gleich null sein.

Bei sehr hohen indirekten Migrationskosten muss der Prozess den Cache erst wieder „warm“ bekommen, damit die vor der Migration durch „heiße“ Caches begünstigte hohe Abarbeitungsgeschwindigkeit wieder erreicht werden kann. Der Scheduler hat ohne Modifikationen selbst keine Möglichkeit indirekte Migrationskosten im Voraus einer Migration abzuschätzen, da er dafür in den Algorithmus der Prozesse und die „Temperatur“ der Caches hineinsehen müsste. Dies ist keine Aufgabe, die der Scheduler in O(1)-Zeit bewältigen kann.

Migration Sweetspot

Ist die Entscheidung gefallen, dass ein Prozess von einer CPU zu einer anderen migriert wird, so ist ein Migration Sweetspot ein Zeitpunkt an dem die direkten und indirekten Migrationskosten für den Prozess kleiner sind, als bei einer sofortigen Migration.

3.2 Vorteile von Migration Sweetspots

Generell sollten Migration von Prozessen zunächst vermieden werden. Ganz vermeiden lassen sie sich jedoch nicht. In einem Mehrkernprozessorsystem kann es zu einer ungleichen Lastverteilung kommen. In solchen Fällen ist es für die Abarbeitungsgeschwindigkeit aller Tasks von Vorteil Prozesse zu migrieren. Für den zu migrierenden Prozess ist eine Migration zu einem Migration Sweetspot am besten.

Aus dem vorherigen Abschnitt ist bekannt, dass die direkten Migrationskosten sich gut vorhersagen lassen. Die Verschiebung eines Prozesses in eine andere Runqueue ist konstant in einer endlichen Zahl von Takten machbar. Des Weiteren migriert ein Prozess fast immer zu einer weniger ausgelasteten CPU, weshalb die potentielle Verzögerung bis zur nächsten Ausführung des Prozesses ohnehin minimiert wird. Für die direkten Migrationskosten spielt es keine Rolle zu welchem Zeitpunkt ein Prozess auf eine andere CPU migriert.

Ein besserer Anhaltspunkt zur Senkung der Migrationskosten ist die Minimierung der indirekten Migrationskosten. Wie kann dies mit dem Wissen um die Probleme bei der Bestimmung der indirekten Migrationskosten erfolgen? Besonders bei speicherintensiven Prozessen, fallen die indirekten Migrationskosten im Vergleich zu den direkten Migrationskosten weitaus höher aus.

Es ist bekannt, dass der Scheduler kein Wissen über den Algorithmus des Programms, den ein Prozesses ausführt, besitzt. Aber wenn dem Scheduler ein einfaches Maß für die „Temperatur“ eines Caches bezüglich eines Prozesses zur Verfügung stehen würde, dann wäre eine grobe Abschätzung der indirekten Migrationskosten möglich. Ein einfaches Beispiel anhand eines Prozesses, der eine Matrixmultiplikation durchführt sollen die Vorteile von Migration Sweetspots gezeigt werden.

3.2.1 Beispiel mit einer Matrixmultiplikation

Zur einfacheren Berechnung wird für dieses Beispiel die Multiplikation von zwei quadratischen Integer-Matrizen der Größe [3; 3] gewählt. Die Größe sei so gewählt, dass die Elemente von einer Zeile der Matrix in eine Cache-Line passen. Es wird ferner angenommen, dass alle drei Matrizen $A \bullet B = C$ komplett in den Cache-Speicher passen und am Anfang des Beispiels keine Daten der Matrizen bereits im Cache stehen. Die Elemente der Matrizen sind zeilenweise abgespeichert und alle drei Matrizen stehen

direkt hintereinander im Speicher. Die Bezeichnung der Elemente der Matrizen sei folgendermaßen.

$$A = \begin{bmatrix} A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 \\ A_7 & A_8 & A_9 \end{bmatrix} \quad B = \begin{bmatrix} B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 \\ B_7 & B_8 & B_9 \end{bmatrix} \quad C = \begin{bmatrix} C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 \\ C_7 & C_8 & C_9 \end{bmatrix}$$

Das Zwischenspeichern von temporären Variablen soll im Folgenden vernachlässigt werden. Der Grund ist zum einen die bessere Übersichtlichkeit und zum anderen gibt es auf modernen x86-Prozessoren einen FMA-Befehlssatz (FMU = fused multiply-add) der Instruktionen der Form r1 = r1 + r2×r3 zulässt [int13a]. Das Zwischenspeichern der Multiplikation in einem separaten Speicherblock kann bei der Matrixmultiplikation damit entfallen. Es ergibt sich der Ablaufplan aus Tabelle 1 bei der Berechnung der Matrixmultiplikation. Es soll phasenweise die Belegung des Caches während der Berechnung der Elemente von C protokolliert werden. Dabei werden auch die Anzahl der Cache-Misses und der Cache-Hits pro Phase gezählt. Ein Strich (beispielsweise $\overline{A_1}$) über einen Element soll einen Cache-Miss anzeigen. Die Berechnung wird von links nach rechts ausgeführt und bei einem Cache-Miss wird die Cache-Line mit den entsprechenden Speicherblock gefüllt.

Tabelle 1: Ablaufplan einer Matrixmultiplikation ohne Migration

Phase	Berechnung	Cache-Misses	Cache-Hits	Cache-Belegung
1	$\overline{A_1}\overline{B_1} + A_2\overline{B_4} + A_3\overline{B_7} = \overline{C_1}$	5	2	$\overline{A_1} \rightarrow A_1\ A_2\ A_3$ $\overline{B_1} \rightarrow B_1\ B_2\ B_3$ $\overline{B_4} \rightarrow B_4\ B_5\ B_6$ $\overline{B_7} \rightarrow B_7\ B_8\ B_9$ $\overline{C_1} \rightarrow C_1\ C_2\ C_3$
2	$A_1B_2 + A_2B_5 + A_3B_8 = C_2$	0	7	…
3	$A_1B_3 + A_2B_6 + A_3B_9 = C_3$	0	7	…

4	$\overline{A_4}B_1 + A_5B_4 + A_6B_7 = \overline{C_4}$	2	5	$A_1\ A_2\ A_3$ $B_1\ B_2\ B_3$ $B_4\ B_5\ B_6$ $B_7\ B_8\ B_9$ $C_1\ C_2\ C_3$ $\overline{A_4} \rightarrow A_4\ A_5\ A_6$ $\overline{C_4} \rightarrow C_4\ C_5\ C_6$
5	$A_4B_2 + A_5B_5 + A_6B_8 = C_5$	0	7	…
6	$A_4B_3 + A_5B_6 + A_6B_9 = C_6$	0	7	…
7	$\overline{A_7}B_1 + A_8B_4 + A_9B_7 = \overline{C_7}$	2	5	$A_1\ A_2\ A_3$ $B_1\ B_2\ B_3$ $B_4\ B_5\ B_6$ $B_7\ B_8\ B_9$ $C_1\ C_2\ C_3$ $A_4\ A_5\ A_6$ $C_4\ C_5\ C_6$ $\overline{A_7} \rightarrow A_7\ A_8\ A_9$ $\overline{C_7} \rightarrow C_7\ C_8\ C_9$
8	$A_7B_2 + A_8B_5 + A_9B_8 = C_8$	0	7	…
9	$A_7B_3 + A_8B_6 + A_9B_9 = C_9$	0	7	…

Während der Phase 1 gibt es aufgrund des „kalten“ Caches mehr Cache-Misses als Cache-Hits. Doch schon ab Phase 2 profitiert der Prozess von den Daten im Cache und es gibt weniger Cache-Misses. Nur noch vereinzelt in der Phase 4 und Phase 7 entstehen Verzögerungen durch je zwei Cache-Misses.

Bei einem Lastausgleich zum Anfang der Phase 2 erfolgt anschließend eine sofortige Migration des Prozesses zu einer anderen CPU. Der „warme“ Cache aus Phase 1 geht verloren und es ergibt sich ein neuer Ablaufplan, der in Tabelle 2 zu sehen ist.

Tabelle 2: Ablaufplan einer Matrixmultiplikation mit Migration ohne Berücksichtigung von Migration Sweetspots

Phase	Berechnung	Cache-Misses	Cache-Hits	Cache-Belegung
1	$\overline{A_1 B_1} + A_2\overline{B_4} + A_3\overline{B_7} = \overline{C_1}$	5	2	$\overline{A_1} \rightarrow A_1\ A_2\ A_3$ $\overline{B_1} \rightarrow B_1\ B_2\ B_3$ $\overline{B_4} \rightarrow B_4\ B_5\ B_6$ $\overline{B_7} \rightarrow B_7\ B_8\ B_9$ $\overline{C_1} \rightarrow C_1\ C_2\ C_3$
Migration				
2	$\overline{A_1 B_2} + A_2\overline{B_5} + A_3\overline{B_8} = \overline{C_2}$	5	2	$\overline{A_1} \rightarrow A_1\ A_2\ A_3$ $\overline{B_2} \rightarrow B_1\ B_2\ B_3$ $\overline{B_5} \rightarrow B_4\ B_5\ B_6$ $\overline{B_8} \rightarrow B_7\ B_8\ B_9$ $\overline{C_2} \rightarrow C_1\ C_2\ C_3$
3-9	analog zu Ablaufplan 1			

Durch die Migration kommen in Phase 2 fünf zusätzliche Cache-Misses hinzu, die es ohne Migration nicht gegeben hat. Jeder Cache-Miss bedeutet eine Verzögerung in der Ausführung eines Prozesses. Die Dauer der Verzögerung ist von den Zugriffszeiten des Caches im Vergleich zu der Zugriffszeit des Speichers abhängig. Die indirekten Migrationskosten steigen.

Aufgrund der zeilenweisen Abarbeitung der Matrixmultiplikation fällt auf, dass ab Phase 4 keine Zugriffe mehr auf die Cache-Lines $A_1 - A_3$ und $C_1 - C_3$ erfolgen. Demzufolge wäre der Zeitpunkt vor Beginn der Phase 4 ein Migration Sweetspot, da man für den Gesamtablaufplan dann die Daten der beiden Cache-Lines nicht mehr erneut in den Cache kopieren muss. Man erspart sich dadurch zwei Cache-Misses und damit sinken auch die indirekten Migrationskosten.

Ein weiterer Migration Sweetspot wäre vor Beginn der Phase 7, denn ab da wird auf die Cache-Lines $A_4 - A_6$ und $C_4 - C_6$ nicht mehr zugegriffen und somit können bei einer Migration zu diesem Zeitpunkt ebenso zwei Cache-Misses eingespart werden.

3.3 Anforderungen an die Suche nach Migration Sweetspots mit dem Linux-Kern

Für die Suche nach Migration Sweetspots soll ein aktueller Linux-Kern verwendet werden. Ferner sollen eine Reihe weiterer Anforderungen A1-A7 aus Tabelle 3 bei der Suche mit berücksichtigt werden. Mögliche Limitierungen oder Abweichungen von den Anforderungen werden in dieser Arbeit an den entsprechenden Stellen erwähnt.

Tabelle 3: Anforderungen an die Suche nach Migration Sweetspots

A1	*Vorhersagbarkeit von Migrationskosten*. Der Scheduler soll eine Möglichkeit erhalten indirekte Migrationskosten zur Laufzeit eines Prozesses abschätzen zu können, um Migration Sweetspots zu ermitteln.
A2	*Allgemeiner Ansatz*. Unabhängig von dem zu migrierenden Prozess und dessen Zustand sollen sich Migration Sweetspots finden lassen, sofern diese existieren. Es sollen neben dem Vorhandensein von Performance-Countern keine weiteren Ansprüche an den Prozessor oder andere Hardware gestellt werden. Unterschiedliche Cache-Architekturen und -Größen verschiedener x86-Prozessoren sollen unterstützt werden.
A3	*Minimaler Ressourcenverbrauch*. Die Ermittlung der Migration Sweetspots darf keine hohe CPU-Last oder gestiegenen Speicherverbrauch im Kernel verursachen.
A4	*Geringer Eingriff in den Linux-Kern*. Alle im Kern enthaltenen Strukturen und Methoden sollen erhalten bleiben und sollen nur um Elemente und Funktionen zur Suche nach Migration Sweetspots erweitert werden.
A5	*Berücksichtigung der zeitlichen Beschränkungen*. Die Suche muss sehr schnell Ergebnisse liefern. Es dürfen keine langen zeitlichen Verzögerungen entstehen. Die Abarbeitungsgeschwindigkeit der Prozesse darf nicht verlangsamt werden.
A6	*Beliebige Prozessmenge und Parallelität*. Die Suche nach Migration Sweetspots soll für beliebig viele rechenbereite Prozesse in allen CPUs des Systems durchführbar sein.
A7	*Protokollierung der Funktion*. Es soll während der Entwicklungs- und Testphase möglich sein, wichtige Kenngrößen der Suche nach Migration Sweetspots zu protokollieren, um eine spätere Auswertung durchführen zu können.

3.4 Auswahl der Suchstrategie und geeigneter Performance-Counter

Durch die Betrachtungen aus Abschnitt 3.2.1 lässt sich schon erahnen, dass zwischen der Verteilung von Cache-Misses und dem Auftretens von Migration Sweetspots eine Korrelation besteht. Eine Erklärung liefert das Lokalitäts-Prinzip, dass auch für die Effektivität von Caches von Bedeutung ist.

Programme beschränken in einer bestimmten Phase der Ausführung sehr häufig ihren Zugriff nur auf einen kleinen Teil ihrer Daten [Her11]. Als Folge daraus kann mit großer Wahrscheinlichkeit vorausgesagt werden, auf welche Daten ein Programm als nächstes zugreifen wird. Es wird zwischen räumlicher und zeitlicher Lokalität unterschieden. Räumliche Lokalität bedeutet, dass bei einem Zugriff auf eine bestimmte Speicheradresse der folgende Zugriff vermutlich auf eine Adresse in der Nachbarschaft erfolgen wird. Zeitliche Lokalität heißt, dass nach einem Speicherzugriff auf eine Adresse in naher Zukunft ein erneuter Zugriff auf die gleiche Adresse erfolgen wird.

Das Lokalitäts-Prinzip hat zur Folge, dass sich zur Laufzeit eins Prozesses Phasen herausbilden in denen nur auf einen kleinen Teil der Seiten des Prozesses zugegriffen wird. Je kürzer man die Phasen wählt, desto höher die räumliche Lokalität und desto kleiner die Arbeitsmenge (Untermenge an Seiten aus dem virtuellen Speicher auf die mit hoher Wahrscheinlichkeit zugegriffen wird) eines Prozesses. Hält man die gesamte Arbeitsmenge eines Prozesses im Cache, so kann man die Ausführung des Prozesses beschleunigen.

Die Breite von Cache-Lines ist typischerweise größer ist als die Operanden einer Instruktion aus dem Befehlssatz der CPU [HMN09]. Dies ist ebenfalls die Ausnutzung des Lokalitäts-Prinzips. Um die Latenz bei den Speicherzugriffen zu minimieren, werden gleich größere Bereiche aus dem Speicher in den Cache geholt. Mit hoher Wahrscheinlichkeit hat man einen Operanden der nächsten Instruktion schon im Cache. Bei einer Häufung von Cache-Misses kündigt sich eine neue Phase in der Abarbeitung eines Prozesses an, in der dieser mit neuen Daten arbeiten wird. Dies kann für die Suche von Migration Sweetspots benutzt werden. Folgt auf einen Zeitraum, bei dem es keine oder wenig Cache-Misses gab, ein Zeitraum mit vielen Cache-Misses, so wechselt mit hoher Wahrscheinlichkeit gerade die Arbeitsmenge des Prozesses und folglich wird dieser mit einer kleineren Wahrscheinlichkeit auf die zur Zeit im Cache vorhanden Daten zugreifen. Das heißt zu diesem Zeitpunkt ist der Cache für den Prozess „kalt“ und bei einer Migration zu einer anderen CPU gibt es weniger indirekte Migrationskosten und damit einen Migration Sweetspot.

Für die schnelle Erfassung von Cache-Misses können Performance-Counter genutzt werden. Zählt man die absolute Anzahl der Cache-Misses für ein kurzes Intervall T und berechnet nach Ablauf des Intervalls das Delta des Zählerstands des Performance-Counters zum letzten Stand, so erhält man eine zeitliche Kurve mit der aktuellen Miss-Rate. Erfasst man mit einem zweiten Counter auf die gleiche Art und Weise auch die absolute Anzahl an Cache-References im Intervall T, so kann man durch Division beider Werte das Miss-Verhältnis ermitteln. Damit hat man einen absoluten Wert, der unabhängig von der absoluten Anzahl der Cache-Misses zur Bestimmung der Migration Sweetspots benutzt werden kann. Es soll zwischen zwei Arten von Migration Sweetspots unterschieden werden.

Migration Sweetspot Typ 1

Detektiert man Zeitpunkte zur direkten Ausführungszeit eines Prozesses auf einer CPU bei denen das Miss-Verhältnis sprunghaft von einem niedrigen Niveau auf einen hohes ansteigt und dabei über einen Schwellwert S1 ($0 < S1 < 1$) steigt, so soll von einem Migration Sweetspot Typ 1 gesprochen werden. Das sind Zeitpunkte an denen die Arbeitsmenge eines Prozesses wechselt und damit ist der Cache für den Prozess „kalt“ und folglich sinken die indirekten Migrationskosten.

Migration Sweetspot Typ 2

Treten während der Präemptierung (Unterbrechung) eines Prozesses Zeitpunkte auf, an denen das Miss-Verhältnis sprunghaft von einem niedrigen Niveau auf einen hohes ansteigt und dabei über einen Schwellwert S2 ($0 < S2 < 1$) steigt, so soll von einem Migration Sweetspot Typ 2 gesprochen werden. Das sind Zeitpunkte an denen die Wahrscheinlichkeit, dass Cache-Lines des präemptierten Prozesseses verdrängt werden, sehr hoch ist. Dadurch „erkaltet“ der Cache des Prozesses und die indirekten Migrationskosten sinken.

Ein Migration Sweetspot Typ 2 kann auch als Blick in die Zukunft gedeutet werden, denn sobald der Prozess nach der Präemptierung weiter ausgeführt werden würde, kommt es mit hoher Wahrscheinlichkeit zu einem sprunghaftem Anstieg des Miss-Verhältnisses und damit kündigt ein Migration Sweetspot Typ 2 häufig einen Migration Sweetspot Typ 1 an.

Es ergeben sich zwei Dinge, die bei der Bestimmung von Migration Sweetspots mit Performance-Countern beachtet werden müssen. Zum einen muss sichergestellt sein, dass durch die Performance-Counter gezählten Cache-Misses und Cache-References

dem tatsächlich ausgeführten Prozessen zugeordnet werden. Zum anderen gibt es auf aktuellen x86-Prozessoren nur sehr wenige programmierbare Performance-Counter, die man zur Aufzeichnung der Cache-Misses und Cache-References benutzen kann. Bei einer gleichzeitigen Suche von Migration Sweetspots in mehr als zwei Prozessen pro logische CPU bei nur vier Performance-Counter im MSR, würden die Counter nicht mehr ausreichen, wenn man für jeden Prozess zwei dedizierte Performance-Counter verwendet.

Aus diesen beiden Gründen und aufgrund der Anforderungen A2 und A6 muss man versuchen, mit nur zwei Performance-Countern auszukommen. Seit den ersten Intel Pentium-Prozessoren sind mindestens zwei Performance-Counter in allen x86-Prozessoren in der PMU vorhanden. Damit erhält man auch die größtmögliche Kompatibilität.

Die Anforderungen lassen sich erfüllen, wenn logische Performance-Counter (Prozess-Counter) eingeführt werden. Jeder Prozess bekommt einen Prozess-Counter zugeordnet. Bei Prozessstart wird der Prozess-Counter auf null gesetzt und der aktuelle Wert des Performance-Counters wird gespeichert. Bei jedem Kontextwechsel bei dem der Prozess die Ausführung auf der CPU entzogen bekommt, wird der aktuelle Wert des Zählerstands des benutzten Performance-Counters erneut gelesen und der Differenzbetrag seit dem Start bzw. seit dem letzten Kontextwechsel bei dem der Prozess wieder auf der CPU ausgeführt wird, auf den Prozess-Counter addiert. Dies erfordert das Abspeichern der aktuellen Zählerstände der Performance-Counter bei jedem Kontextwechsel bei dem der Prozess als nächstes von der CPU ausgeführt wird.

Wird der logische Performance-Counter gelesen, so wird ebenfalls nur der Differenzbetrag des Zählers des benutzten Performance-Counters auf den logischen Performance-Counter addiert und man erhält einen korrekten Wert. Mit diesem Vorgehen lässt sich für jeden auf einer CPU laufenden Prozess mit Nutzung von nur zwei Performance-Countern ein prozesseigener Counter für Prozess-Cache-Misses und Prozess-Cache-References erstellen.

Da die beiden Performance-Counter für die Erfassung der Cache-Misses und Cache-References auch während der Präemptierung weiterzählen, kann man auch das Miss-Verhältnis während der Präemptierung erfassen. Somit ist auch die Erkennung von Migration Sweetspots Typ 2 möglich.

3.5 Suche nach Migration Sweetspots durch Benutzer-Prozesse

Im aktuellen Linux-Kern ist der Zugriff auf die Register im MSR nicht ausschließlich dem Kern überlassen. Durch das Laden des MSR-Moduls ist es möglich auf Performance-Counter direkt aus einem Benutzer-Prozess zuzugreifen. Dafür muss der Benutzer-Prozess allerdings über Root-Rechte verfügen. Auch das Nachladen des MSR-Moduls benötigt Root-Rechte.

Für jede logische CPU erzeugt das MSR-Modul eine Geräte-Datei im Pfad `/dev/cpu/`. Anschließend kann mit den Systemaufrufen `wrmsr()` und `rdmsr()` auf die Register der Performance-Counter zugegriffen werden [lin09]. Ein Benutzer-Prozess, der über Root-Rechte verfügt, kann mit `wrmsr()` dann von sich aus zwei Performance-Counter zur Protokollierung der Cache-Misses und Cache-References konfigurieren und starten. Auch das Auslesen von Prozess-Informationen und CPU-Ausführungszeiten ist mit Hilfe des proc-Dateisystem für Benutzer-Prozesse mit Root-Rechten durchführbar [lka99]. Durch das Parsen der Werte `utime` und `stime` aller Prozesse, die in `/proc/<PID>/stat` aufgeführt werden, kann ein Benutzer-Prozess so die CPU-Last aller im System vorhandener logischer CPUs feststellen. Bemerkt der Benutzer-Prozess eine hohe Last auf der durch den Prozess benutzten CPU, kann dieser unter Verwendung des selbst errechneten Miss-Verhältnisses Migration Sweetspots finden.

Es wäre ein Szenario vorstellbar, bei dem so ohne Änderungen am Linux-Kern Benutzer-Prozesse mit Root-Rechten von sich aus dem Kern mitteilen würden, dass Sie gerne auf eine andere CPU migrieren möchten. Dies wäre eine bezüglich der Anforderung A4 optimale Variante. Denkbar wäre beispielsweise ein vom Prozess zur Protokollierung gestarteter Thread, der einmal im Intervall T mit `rdmsr()` die Performance-Counter für Cache-Misses und Cache-Reference ausliest. Des Weiteren kann der Prozess den Systemaufruf `sched_setaffinity()` zur Mitteilung von Migration Sweetspots nutzen [lin09]. So lange keine Migration erwünscht ist, könnte sich der Prozess an eine bestimmte CPU pinnen. Ermittelt der Prozess anhand der Performance-Counter einen Migration Sweetspot, hebt es das Pinning auf und überlässt damit dem Scheduler die Möglichkeit den Prozess zu migrieren.

Dieses Vorgehen ist allerdings nicht sinnvoll. Es gibt ein gravierendes Problem. Ein Benutzer-Prozess mit Root-Rechten erhält mit dem MSR-Modul zwar die Möglichkeit Performance-Counter zu steuern und zu lesen, allerdings sind das nur globale Counter, die für alle Prozesse, die auf der CPU ausgeführt werden, mitzählen. Es gibt keine Garantie, dass nur der eine gepinnte Prozess auf der CPU ausgeführt wird. Somit können die durch den Protokoll-Thread ermittelten Cache-Misses und Cache-

References ebenso durch andere Prozesse verursacht worden sein. Logische Performance-Counter können nicht durch einen Benutzer-Prozess selbst eingeführt werden, da dieser nicht weiß, wann Kontextwechsel auf der CPU stattfinden. Der Linux-Scheduler teilt Prozessen nicht mit, ob und wann diese präemptiert werden.

Ein weiteres Problem ist, dass ein Benutzer-Prozess nicht wissen kann, wer sonst noch auf die MSR-Register zugreift. Ein anderer Prozess mit Root-Rechten oder der Kern könnte die Performance-Counter beliebig umprogrammieren. Auch die Anforderung A6 ist mit Benutzer-Prozessen, aufgrund der limitiert vorhanden Performance-Counter in x86-Prozessoren, nicht haltbar. Pro Benutzer-Prozess wären schon zwei Counter belegt und bei zwei gleichzeitig ausgeführten Benutzer-Prozessen wäre bei einem aktuellen Core-i7-Prozessor bereits das Limit erreicht.

3.6 Folgen der Migrationsverzögerung

Wenn Prozesse generell nicht mehr zum Zeitpunkt eines Lastausgleichs migrieren, sondern erst verzögert zu einem Migration Sweetspot migrieren hat das Folgen. Die positiven Aspekte der Migration wurden schon in Abschnitt 3.2 besprochen.

Gleichzeitig entstehen durch die verzögerte Migration natürlich auch Nachteile. Ein Prozess der von einer viel ausgelasteten CPU zu einer wenig ausgelasteten CPU migriert, wird anschließend mehr Ausführungszeit bekommen und dadurch eher beendet sein. Zentral ist die Frage ob die Verringerung der indirekten Migrationskosten bei einem Migration Sweetspot insgesamt größer ist als die Zeitdauer der Migrationsverzögerung d. Zur Abschätzung wird die folgende Symbolik verwendet:

Dauer der Migrationsverzögerung (Takte)	d
Cache-Miss-Kosten (Takte)	t_S
Anzahl der Cache-Hits während der Migrationsverzögerung d	$n(d)$
Wahrscheinlichkeit, dass auf ein Datenwort nach einem Zugriff des Prozesses in naher Zukunft wieder zugegriffen wird	p

Die Cache-Miss Kosten geben an, wie hoch die zeitliche Verzögerung bei einem Cache-Miss beim Lesen eines Datenwortes im Vergleich zu einem Cache-Hit ausfallen. Die Wahrscheinlichkeit p ist natürlich bei jedem Prozess unterschiedlich und hängt auch von den tatsächlichen Daten ab, mit denen ein Prozess arbeitet. Es ist zu erwarten, dass die Wahrscheinlichkeit p indirekt proportional zur Größe der Arbeitsmenge eines Prozesses angenommen werden kann. Bei insgesamt mehr potentiellen Daten steigt die Wahrscheinlichkeit, dass man auf Teile davon nicht mehr zugreifen wird. Genau bestimmen lässt sich p allerdings nicht und ist vom konkreten Algorithmus eines Programms abhängig. Aufgrund des Lokalitätsprinzips darf p für die meisten Prozesse als relativ hoch angenommen werden.

Die Anzahl der Cache-Hits während der Migrationsverzögerung $n(d)$ ergibt sich aus der Differenz von Cache-References und Cache-Misses. Dies ist praktisch, da man den Performance-Counter für die Cache-Misses zur Suche nach Migration Sweetspots ebenfalls benutzt.

Es folgt die Überlegung, dass mit einer Wahrscheinlichkeit von $(1-p)$ für einen während der Migrationsverzögerung d aufgetreten Cache-Hit, dies der vorerst letzte Speicherzugriff auf ein Datenwort gewesen ist. Die zusätzlichen Takte, die die CPU bei einer sofortigen Migration auf die Daten länger warten müsste, ergeben die Cache-Miss Kosten. Das ist genau die Zeit, die man bei einer verzögerten Migration zu einem Migration Sweetspot einsparen könnte, da das Datenwort bereits im Cache steht.

Die Wahrscheinlichkeit, dass eine Migration zu einem Migration Sweetspot mehr die Durchlaufzeit eines Prozesses mehr verkürzt, als durch die Dauer der Migrationsverzögerung d an Zeit verloren geht, wird begünstigt durch hohe Cache-Miss-Kosten und eine kleine Wahrscheinlichkeit p. Der Zeitpunkt an dem sich die Verkürzung der Durchlaufzeit und die Migrationsverzögerung egalisieren soll d_{max} heißen. Je mehr Cache-Hits während d auftreten, desto größer kann d_{max} ausfallen. Zur Abschätzung lässt sich folgende Gleichung nutzen:

$$d_{max} = n(d)(1-p)t_S$$

Ein Beispiel mit $d = 300\,Takte,\ n(d) = 100,\ p = 0.8,\ t_S = 200\,Takte$ ergibt ein d_{max}von 4000 Takten. Findet sich bis d_{max} kein Migration Sweetspot und treten keine weiteren Cache-Hits auf sollte rein statistisch nicht länger gewartet werden. Ein d_{min} lässt sich nur für den Fall ermitteln, wenn der Scheduler weiß wie viele Cache-Hits $\tilde{n}(k)$ ein Prozesses in naher Zukunft für eine kurze konstante Zeit k erhalten wird. Dann gilt als Abschätzung für d_{min} die folgende Gleichung:

$$d_{min} = \tilde{n}(k)(1-p)t_S$$

Nur für den Fall $d_{min} > k$ sollte eine verzögerte Migration zu einem Migration Sweetspot überhaupt in Erwägung gezogen werden. Der Wert für k sollte hinreichend klein gewählt werden.

Tabelle 4 zeigt die Takte, die benötigt werden um ein Datenwort zu lesen, in Abhängigkeit von Cache-Hits der unterschiedlichen Ebenen der Speicherhierarchie [int08]. Für einen Intel Core-i7-Prozessor mit einer Taktfrequenz von drei GHz ergeben sich damit bei Speicher-Zugriffszeiten von etwa 100 ns [BCC08] und einem L1-Cache-Hit, Cache-Miss-Kosten von über 300 Takten. Die Cache-Miss-Kosten verlängern sich nochmal deutlich um die Dauer von etwa zwei weiteren Speicherzugriffszeiten, wenn im TLB der Eintrag zur virtuellen Adresse nicht vorhanden ist (TLB-Miss) und die Seitentabelle nicht im Cache liegt.

Tabelle 4: Zugriffszeiten bei Cache-Hit bei Intel Core-i7-Prozessoren, aufgeschlüsselt nach Cache-Ebene

L1 Cache-Hit	~4 Takte
L2 Cache-Hit	~10 Takte
L3 Cache-Hit, Cache-Line nicht geteilt	~40 Takte
L3 Cache-Hit, Cache-Line geteilt mit einem anderen Kern	~65 Takte
L3 Cache-Hit, Cache-Line modifiziert in einem anderen Kern	~75 Takte

4 Implementierung

In diesem Kapitel wird die Implementierung der Suche nach Migration Sweetspots in einem Linux-Kern beschrieben. Zunächst wird im ersten Abschnitt 4.1 erklärt, wie für ausgewählte Programme eines Mikro-Benchmarks ein Profiling durchgeführt werden kann.

Im Abschnitt 4.2 wird ausführlich vorgestellt, wie Performance-Counter innerhalb des Linux-Kerns mit der `perf_event`-Schnittstelle konfiguriert werden können. Abschnitt 4.3 befasst sich mit den Änderungen, die am Scheduler durchgeführt wurden und beschreibt neu implementierte Funktionen. Die Überlegungen zur Wahl der Periodendauer des Intervalls T werden im Abschnitt 4.4 behandelt.

Schließlich wird im Abschnitt 4.5 noch das für die Suche implementierte Kernel-Modul vorgestellt und eine allgemeine Zusammenfassung der Probleme, die während der Implementierung aufgetreten sind, befindet sich im Abschnitt 4.6.

Zur Implementierung wurde der zu dem Zeitpunkt der Implementierung letzte stabile Linux-Kern Version 3.9.9 gewählt. Der Quellcode des Kerns wurde aus dem `linux-stable`-GIT-Repository heruntergeladen. Für die Entwicklungszeit und das Debugging wurde eine virtuelle Maschine mit QEMU eingerichtet, um den neuen Kern schnell testen und debuggen zu können. Mit dem Parameter `-kernel` konnte beim Start der virtuellen Maschine in QEMU eine fertig kompilierte Kernel-Image-Datei ausgewählt. Damit entfällt der Schritt der Installation eines neuen Kernels auf einem Testsystem. Dies hat sehr viel Zeit während der Implementierung eingespart.

Die zur Kompilierung des Kerns gewählte Konfigurationsdatei basiert auf der Standartkonfiguration `make defconfig` für die x86-64 Architektur. Für die Virtualisierung benötigte Treiber wurden entsprechend Listing 1 aktiviert.

```
…
CONFIG_VIRTIO_PCI=y;
CONFIG_VIRTIO_BALLOON=y;
CONFIG_VIRTIO_BLK=y;
CONFIG_VIRTIO_NET=y;
CONFIG_VIRTIO=y;
CONFIG_VIRTIO_RING=y;
…
```

Listing 1: Angepasste Einstellungen der Konfigurationsdatei für die Kompilierung des Linux-Kernels zur Ausführung in QEMU

Die Implementierung erfolgte auf einem Desktop-PC mit einem Intel Core-i5-2400 Prozessor mit vier Kernen und einer Taktfrequenz von 3,1 GHz. Hyper-Threading ist auf diesem Prozessor nicht vorhanden. Das Energie-Management des Systems wurde deaktiviert. Dafür wurden im BIOS alle Übertaktungs- und Untertaktungs-Optionen (Turbo-Boost, SpeedStep) ausgeschalten. Zur Einhaltung der konstanten Taktfrequenz wurde mit dem Tool `indicator-cpufreq` der `ondemand`-Governor ausgewählt und 3,1 GHz als Minimal- und Maximalwert eingestellt.

Um eine Migration der zu untersuchenden Prozesse bei einem Lastausgleich zu verhindern, wurden die Prozesse mit einem `sched_setaffinity`-Systemaufruf auf eine festgelegte CPU gepinned. Mit diesen Maßnahmen wurde erreicht, dass die Abarbeitungsgeschwindigkeit der Prozesse durchgängig konstant gehalten wird und keine durch Anpassungen der Taktfrequenz des Prozessors bedingten Verschiebungen beim Wechsel in Ausführungsphasen mit einer neuen Arbeitsmenge eines Prozesses auftreten.

4.1 Profiling und Mikro-Benchmarks

Für die Entwicklung- und Evaluierungsphase wurden zunächst einige einfache Programme mit vorhersagbarem Speicherzugriff geschrieben. Damit wurde ein Mikro-Benchmark geschaffen, mit dessen Hilfe sich das korrekte Auslesen von Cache-Misses und Cache-References durch Prozess-Counter überprüfen lässt. Zu dem Mikro-Benchmark gehören die Programme `cpuburner`, `memeater` und `change`. Alle drei lassen sich per Startparameter auf eine CPU pinnen. `Cpuburner` ist ein Programm, das nur eine Endlosschleife durchführt. Es werden keine Daten im Speicher angefasst. Für

den ausgeführten Prozess soll ein Miss-Verhältnis von null nachgewiesen werden. `Memeater` allokiert einen Speicherbereich fest definierter Größe und überschreibt diesen in einer Endlosschleife immer wieder mit neuen Werten. Ein hohes Miss-Verhältnis wird erwartet. Die Größe des Datenbereiches lässt sich ebenfalls per Startparameter verändern. Eine Zusammenführung beider Programme stellt `change` dar. Dieses führt immer im Wechsel die Endlosschleifen von `cpuburner` und `memeater` aus. Das Umschalten zwischen beiden Schleifen erfolgt durch ein periodisch auftretendes `SIGNAL`. Damit sollte ein periodischer Wechsel zwischen hohem und niedrigem Miss-Verhältnis zur Ausführungszeit des Programms auftreten.

Das in C++ programmierte Programm `matrixmult` gehört ebenfalls zum Mikro-Benchmark. Es führt eine Matrixmultiplikation von zwei quadratischen Matrizen durch. Mit einem Startparameter lässt sich die Größe der verwendeten Matrizen festlegen. Bei steigender Größe der Matrizen wird ein Zeitpunkt erreicht, ab dem die Matrizen nicht mehr vollständig in den Cache passen. Damit müssen Migration-Sweetspots Typ 1 während der Ausführung zu finden sein.

Das Aufzeichnen des Miss-Verhältnisses während der Ausführung der Programme (Profiling), erfolgt durch einen modifizierten Linux-Kern. Dafür muss der Kern so angepasst werden, dass der Start und das Ende der Ausführung von Prozessen, die zu einem Programm aus dem Mikro-Benchmark gehören, erkannt wird.

4.1.1 Erkennen von Prozess-Start und Prozess-Ende ausgewählter Programme

Der Start eines neuen Prozesses verläuft zweistufig in zwei Phasen. Die erste Phase ist das Forking, und anschließend folgt die Ausführungsphase [Tan09].

Die Forking-Phase wird durch einen `fork()`-Systemaufruf eingeleitet. Dabei wird eine identische Kopie (nur zu unterscheiden durch eine neue Prozess-Id `PID`) eines Elternprozesses erstellt, der nachfolgend Kindsprozess genannt werden soll. Nach dem Abarbeiten der Forking-Routine im Kern wird der Kindsprozess bereits vom Scheduler auf einer CPU eingeplant und führt zunächst den gleichen Code, wie der Elternprozess aus.

Erst in der Ausführungsphase erfolgt die eigentliche Trennung des Kindsprozesses vom Elternprozess. Die Ausführungsphase wird mit dem `execve()`-Systemaufruf oder einem ähnlichen Systemaufruf durch den Kindsprozess gestartet. In dieser Phase wird das Speicherabbild des Prozesses gewechselt und anschließend führt der Prozess einen neuen Code aus. Nach der Ausführungsphase haben Kinds- und Elternprozess jeweils eigene Adressräume.

Zur eindeutigen Identifizierung, welches Programm ein Prozess ausführt, kann im Linux-Kern das `comm`-Attribut in der `tast_struct`-Struktur eines Prozess genutzt werden [lin09]. Darin wird der Dateiname der Binärdatei, die den Code enthält, der vom Prozess ausgeführt wird, eingetragen. Sobald durch die Funktion `setup_new_exec()` in der Datei `fs/exec.c` der Ausführungskontext eines Prozesses überschrieben wird und die `task_struct`-Struktur mit den Attributen des auszuführenden Programms aktualisiert wurde, kann anschließend durch einen Abgleich mit dem `progs`-Array festgestellt werden, ob ein Programm aus dem Mikro-Benchmark gestartet wurde. Die entsprechende Änderung am Quellcode des Linux-Kern wird in Listing 2 gezeigt.

```
void setup_new_exec(struct linux_binprm * bprm)
{
   int n;
   const char *progs[] = {"matrixmult","cpuburner","memeater","change"};
   …
   //current->comm speichert den Dateinamen der ausführbaren Datei
   set_task_comm(current, bprm->tcomm);
   …
   for (n = 0; n < 4; n++) {
      if (!strcmp(progs[n], current->comm)){
         kk_perf_setup_task(current);
         break;
      }
   }
   …
}
```

Listing 2: Erkennen des Prozess-Starts von für das Profiling von Programmen aus dem Mikro-Benchmark (setup_new_exec-Funktion)

Der Vorteil dieses „Linux-Hacks“ ist, dass das Profiling der Prozesse noch vor der eigentlichen Ausführung des ersten Codes der Programme starten kann. Außerdem müssen Programme, für die ein Profiling durchgeführt werden soll, nicht extra angepasst werden. Somit lassen sich auch andere Programme, die nicht zum Mikro-Benchmark gehören, unproblematisch auf Migration Sweetspots untersuchen. Die Funktion `setup_new_exec()` wird unabhängig vom Binärformat einer ausführbaren Datei immer aufgerufen.

Eine andere Möglichkeit wäre der Weg über die `PID`. Über das proc-Dateisystem könnte ein Programm aus dem Mikro-Benchmark beim Start der Ausführung an ein spezielles Kernel-Modul eine Nachricht mit der `PID` des ausgeführten Prozesses

schicken. Beim Empfang der Nachricht kann das Kernel-Modul das Profiling starten. Nachteil dieser Methode ist allerdings, dass die Programme dafür angepasst werden müssen und ein Kernel-Modul für die Kommunikation nötig wird.

Zur Initialisierung der logischen Performance-Counter für das Profiling der Prozesse wird die Funktion `kk_perf_setup_task()` aufgerufen. Mit `current` gibt es im Linux-Kern ein Makro, um auf die `task_struct`-Sturktur des zur Zeit auf der CPU ausgeführten Prozesses zuzugreifen [lin09].

Um das Profiling der Prozesse zu beenden und nicht mehr benötigte Ressourcen, die durch die Funktion `kk_perf_setup_task()` belegt werden, wieder freizugeben, muss das Ende der Ausführung eines Prozesses festgestellt werden. Dies kann in der Ausführungsroutine des `exit()`-Systemaufrufs erfolgen. Die entsprechende Funktion `do_exit()` befindet sich in der Datei `kernel/exit.c` und wird ebenfalls bei unerwartetem Beenden von Prozesses aufgerufen. Analog zu Listing 2wird für Prozesse, die einen Eintrag im `progs`-Array besitzen, die Funktion `kk_perf_release_task()` ausgeführt. Diese kümmert sich um das Deallokieren von nicht mehr benötigtem Speicher und das Beenden des Profilings.

4.2 Konfiguration der Performance-Counter mit der perf_event-Schnittstelle

Im Linux-Kern gibt es eine Sammlung von Funktionen, die die Nutzung der `perf_event`-Schnittstelle im Kern vereinfacht. Dazu gehören im Wesentlichen die drei Funktionen `perf_event_create_kernel_counter()`, `perf_event_read_value()` und `perf_event_release_kernel()`. Die beiden Letztgenannten werden für das Lesen und das wieder freigeben von durch `perf_event_create_kernel()` gestarten Perfomrance-Counter genutzt.

Die perf_event_create_kernel_counter()-Funktion

Die Funktion `perf_event_create_kernel_counter()` kann genutzt werden, um Performance-Counter zu starten. Die Funktion erwartet die Parameter: `struct perf_event_attr *attr`, `int cpu, struct task_struct *task`.

Der `attr`-Parameter ist einen Zeiger auf eine `perf_event_attr`-Struktur, die einen Hardware-Event beschreibt. Ein Beispiel für eine Konfiguration der `perf_event_attr`-Struktur zur Erfassung von Cache-Misses ist in Listing 3 zu sehen.

```
struct perf_event_attr my_attr = {
   .type = PERF_TYPE_HARDWARE;
   .config = PERF_COUNT_HW_CACHE_MISSES;
   .exclude_user = 0;
   .exclude_kernel = 0;
   .exclude_idle = 1;
   .pinned = 1;
}
```

Listing 3: Beispiel der Konfiguration der perf_event_attr-Struktur zur Erfassung von Cache-Misses

Mit dem `type`-Attribut lässt sich der generelle Typ des zu erfassenden Ereignisses auswählen. Unter dem Makro `PERF_TYPE_HARDWARE` sind einige allgemeine Ereignisse, die die PMU-Schaltkreise des Prozessors nutzen, zusammengefasst. Mit dem `config`-Attribut lässt sich steuern, welches konkrete Ereignis der Counter zählen soll. Ausgewählte für diese Arbeit relevante Konfigurationsmöglichkeiten für das `config`-Attribut vom Typ `PERF_TYPE_HARDWARE` sind in Tabelle 5 zusammengefasst.

Tabelle 5: Beispiel für Werte des config-Attributs bei ausgewähltem PERF_TYPE_HARDWARE type-Attribut

`PERF_COUNT_HW_CACHE_REFERENCES`	Zählt Cache-Zugriffe auf den Last Level Cache (LLC).
`PERF_COUNT_HW_CACHE_MISSES`	Zählt Cache-Misses beim Zugriff auf den LLC.
`PERF_COUNT_HW_CPU_CYCLES`	Zählt CPU-Takte. Änderungen der Taktfrequenz wirken sich direkt aus.
`PERF_COUNT_HW_REF_CPU_CYCLES`	Zählt Referenz-CPU-Takte des Prozessors. Änderungen der Taktfrequenz wirken sich nicht aus.

Um die Cache-Misses ausgewählter Caches eines Prozesses zu zählen, muss das `type`-Attribut auf das Makro `PERF_TYPE_HW_CACHE` gesetzt werden. Das `config`-Attribut setzt sich anschließend, mit dem folgendem Aufbau zusammen:

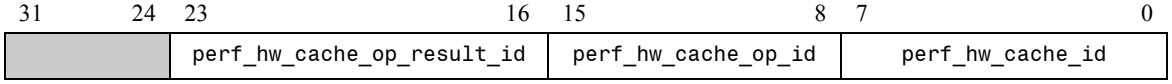

Tabelle 6 gibt eine Übersicht über ausgewählte Werte für die `config`-Attribut Ereignisauswahl. Mit `perf_hw_cache_id` wird ein bestimmter Cache ausgewählt. Mit `perf_hw_cache_op_id` wählt man die Art der Zugriffe und mit `perf_hw_cache_op_result_id` kann zwischen Misses und References gewählt werden.

Tabelle 6: Ausgewählte Werte für die Konfiguration des config-Attributs bei ausgewähltem PERF_TYPE_HW_CACHE type-Attribut

`perf_hw_cache_id`	`PERF_COUNT_HW_CACHE_L1D`	Wählt L1-Datacache.
	`PERF_COUNT_HW_CACHE_LL`	Wählt LLC-Cache.
	`PERF_COUNT_HW_CACHE_DTLB`	Wählt Data-TLB-Cache.
	`PERF_COUNT_HW_CACHE_NODE`	Wählt den Hauptspeicher aus.
`perf_hw_cache_op_id`	`PERF_COUNT_HW_CACHE_OP_READ`	Wählt Lesezugriffe.
	`PERF_COUNT_HW_CACHE_OP_WRITE`	Wählt Schreibzugriffe.
`perf_hw_cache_op_result_id`	`PERF_COUNT_HW_CACHE_RESULT_ACCESS`	Zählt die Anzahl der Zugriffe.
	`PERF_COUNT_HW_CACHE_LL`	Zählt die Anzahl der Misses.

Mit Hilfe der `exclude`-Attribute in der `perf_event_attr`-Struktur lässt sich die Inkrementierung der Performance-Counter für bestimmte Betriebs-Modi der CPU deaktivieren. Mit dem `pinned`-Attribut kann man verlangen, dass der Performance-Counter zu allen Zeiten einer eindeutigen PMU auf der CPU zugeordnet wird und die PMU zu keiner Zeit durch einen anderen Counter benutzt werden kann.

Mit dem `cpu`-Parameter der `perf_event_create_kernel_counter()`-Funktion kann ausgewählt werden, ob der Counter nur die Ereignisse einer bestimmten logischen CPU erfassen soll, oder wenn der dritte Parameter `task` gesetzt ist, der Zähler bei einer Migration des bestimmten Prozesses auf eine andere CPU dort die Ereignisse für den Prozess weiterzählen soll.

Es ergeben sich die Optionen zur Zählung aller Ereignisse auf einer logischen CPU für alle darauf ausgeführten Prozesse (globaler Performance-Counter), wenn `cpu` größer gleich `null` ist und `task` nicht definiert wird (`task = null`). Wird `cpu` auf `-1` gesetzt, dann muss der Parameter `task` mit einem festgelegten Prozess angegeben sein (`task != null`). Ansonsten tritt ein Fehlerfall auf [lin13].

4.3 Änderungen am Scheduler

Der Eingriff in den Quellcode des Linux-Schedulers fällt relativ gering aus. Innerhalb der `__schedule()`-Funktion in der Datei `kernel/sched/core.c` erfolgt nach der Ausführung eines Kontextwechsels noch der Aufruf der neu implementierten Funktionen `kk_preempt_start()`, `kk_preempt_end()` und `kk_perf_log_task()`. Eine Übersicht über alle neu implementierten Funktionen gibt Tabelle 7.

Tabelle 7: Übersicht aller neu implementierten Funktionen im Linux-Kern

Funktion	Nutzung
`kk_perf_init()`	Statische Allokierung des Speichers und Initialisierung der Objekte zur Konfiguration der Performance-Counter.
`kk_perf_setup_task()`	Start des Profilings für einen ausgewählten Prozess.
`kk_perf_release_task()`	Ende des Profilings für einen ausgewählten Prozess.
`kk_preempt_start()`	Zwischenspeichern der Zählerstände beim Kontextwechsel während des Wechsel des Prozess-Zustandes vom laufend- in den bereit-Zustand (Start der Präemptierung).
`kk_preempt_end()`	Zwischenspeichern der Zählerstände beim Kontextwechsel während des Wechsel des Prozess-Zustandes vom bereit- in den laufend-Zustand (Ende der Präemptierung).
`kk_perf_log_task()`	Aufruf am Ende jedes Intervalls T zur Entscheidung, ob ein Migration Sweetspot aufgetreten ist. Protokollierung der Zählerstände der Performance-Counter.

4.3.1 Änderungen an der task_struct-Struktur

Aus Kapitel 3 ist bekannt, dass zwei durchgängig aktive Performance-Counter – je einer für Cache-Misses und Cache-References – ausreichend sind, um während des Profilings von Prozessen nach Migration Sweetspots zu suchen. Während der Prozess aktiv auf der einer CPU ausgeführt wird, also im laufend-Zustand, können die Performance-Counter für die Suche nach Migration Sweetspots Typ 1 genutzt werden. Werden andere Prozesse ausgeführt und der Prozess befindet sich im bereit- oder blockiert-Zustand können die Performance-Counter für die Suche nach Migration Sweeetspots Typ 2 genutzt werden. Es müssen jederzeit zwei Counter pro logische CPU aktiv sein. Die Notwendigkeit begründet sich darin, dass während der Präemptierungsphase (Suche nach Migration Sweetspots Typ 2) bei einem von allen CPUs gemeinsam genutzten LLC alle Cache-Misses aller CPUs zu erfassen sind.

Während einer Zählperiode eines Intervalls T kann ein Prozess mit aktiviertem Profiling mehrmals den Zustand wechseln. Deshalb ist es nötig die Zählerstände aller aktiven Performance-Counter für die Suche nach Migration Sweetspots beider Typen,

beim Anfang und beim Ende einer Präemptierungsphase zwischenzuspeichern. Beginnt eine Präemptierungsphase wird die Differenz der aktuellen Zählerstände der Performance-Counter auf der vom Prozess aktiv ausgeführten CPU (`kk_oncpu-Variable`) mit den zuletzt zwischengespeicherten Werten gebildet und auf die logischen Performance-Counter `kk_values` addiert. Endet eine Präemptierungsphase wird die Differenz der aktuellen Zählerstände aller Performance-Counter mit den zuletzt zwischengespeicherten Werten gebildet und auf die logischen Performance-Counter `kk_preemt_values` addiert. Sowohl `kk_values`, als `auch kk_preempt_values` müssen groß genug sein, um genau zwei Zählregister eines Performance-Counters aufzunehmen. Das Array `kk_stand_counter` wird für das Zwischenspeichern der Zählerstände genutzt und muss daher bei vier logischen CPUs acht Performance-Counter speichern. Die logischen Performance-Counter in `kk_values` werden für die Suche nach Migration Sweetspots Typ 1 benutzt. Für Migration Sweetspots Typ 2 können die logischen Performance-Counter in `kk_preempt_values` genutzt werden.

Für die Protokollierung der Zählstände der logischen Performance-Counter während der Entwicklungs- und Evaluierungsphase muss die genaue Startzeit des Prozesses gespeichert werden. Diese wird in der Variable `kk_process_start` gespeichert. Die Variable `kk_lastread` speichert die Zeit in der das letzte Intervall T begonnen hat. Um die Werte der logischen Performance-Counter `kk_preempt_values` des letzten Intervalls T als Referenz für das nächstfolgende Intervall zu sichern, wird das Array `kk_last_cm` genutzt.

Der entsprechend modifizierte Teil der `task_struct`-Struktur für die Suche nach Migration Sweetspots auf einem Prozessor mit vier logischen CPU ist in Listing 4 zu sehen. Insgesamt wird die `task_struct`-Struktur um 132 Byte vergrößert. Bei einer vorherigen Größe von 3272 Bytes ergibt sich ein ca. vier Prozent erhöhter Speicherbedarf. Da die Variable `kk_process_start` nur für die Protokollierung genutzt wird und `kk_oncpu` durch den Aufruf der Kernel-Funktion `smp_processor_id()` ersetzt werden könnte, lässt sich bei einer finalen Implementierung eine Reduzierung des Speicherverbrauchs um mindestens zwölf Byte erreichen.

```
struct task_struct {
   …
   u64 kk_process_start;
   u64 kk_lastread;
   int kk_oncpu;
   u64 kk_stand_counter[8];
   u64 kk_values[2];
   u64 kk_preempt_values[2];
   u64 kk_last_cm[2];
   …
}
```

Listing 4: Änderungen an der task_struct-Struktur im Linux-Kern

4.3.2 Neu implementierte Funktionen

Die Funktion `kk_perf_setup_task()` wird in der Ausführungsphase beim Start eines Prozesses aufgerufen, bevor nach einem Kernaustritt der ersten Code des Benutzer-Prozesses auf der CPU ausgeführt wird. Zunächst überprüft die Funktion ob schon andere Prozesse mit aktiviertem Profiling auf dem System ausgeführt werden. Im negativen Fall werden auf allen CPUs die Performance-Counter für das Zählen von Cache-Misses und Cache-References mit der Funktion `perf_event_create_kernel_counter()` gestartet. Ferner setzt die Funktion alle logischen Performance-Counter des Prozesses auf null und speichert die Startzeit des Profilings für die Protokollierung. Es beginnt das erste Intervall T.

Solange der Prozess nicht präemptiert wird und durchgängig auf der CPU ausgeführt wird, sind die Performance-Counter semantisch die logischen Performance-Counter des Prozesses und werden für die Suche nach Migration Sweetspots Typ 1 benutzt. Wird dem Prozess durch den Scheduler die CPU entnommen und eine Präemptierungsphase beginnt, wird die Funktion `kk_preempt_start()` aufgerufen. Sie übernimmt das Zwischenspeichern der Werte, wie im vorherigen Abschnitt beschrieben. Ab jetzt sind die Performance-Counter semantisch logische Performance-Counter für die Suche nach Migration Sweetspots Typ 2.

Am Ende der Präemptierungsphase, wenn der Scheduler den Prozess wieder zur Ausführung auf einer CPU reaktiviert, wechselt wieder die Semantik der Performance-Counter für den Prozess und die Funktion `kk_preempt_end()` wird aufgerufen. Bei einem Kontextwechsel bei dem einem Prozess die Ausführung auf einer CPU entzogen bzw. wieder zugewiesen wird, sorgen `kk_preempt_start()` und `kk_preempt_end()`

damit dafür, dass in den Arrays der logischen Performance-Counter `kk_values` und `kk_preempt_values` die aktuellen Zählerstände eines Intervalls T gespeichert sind. Das sind damit auch die Zeitpunkte an denen nach Ablauf des Intervalls T die Entscheidung, ob ein Prozess sich gerade an einem Migration Sweetspot befindet, gefällt werden kann. Diese Entscheidungslogik ist in der Funktion `kk_perf_log_task()` implementiert.

Um die Division bei der Berechnung des Miss-Verhältnisses in der `kk_perf_log_task()`-Funktion zu vermeiden (Anforderungen A3 und A5), werden die Überschreitungen von den Schwellenwerten S1 und S2 mit der nachfolgenden Bedingung verknüpft. Dabei wird für Faktor das Reziproke des zu überschreitenden Schwellenwerts eingesetzt. Wenn beispielsweise der Schwellwert S1 = 0,1 gewählt wurde, dann wird für Faktor der Wert 10 eingesetzt.

$$\text{Faktor} \times \text{Cache-Misses} > \text{Cache-References}$$

Für die Entwicklungsphase und zur Protokollierung der Cache-Misses und Cache-References wird innerhalb der `kk_perf_log_task()`-Funktion `trace_printk()` benutzt. Diese Funktion wurde in den Linux-Kern implementiert, um zeitlich sensible Bereiche, wie beispielsweise den Scheduler zu debuggen [lka13b]. Im Gegensatz zu `printk()` schreibt `trace_printk()` nicht in den Kernel-Log-Puffer und hat einen sehr geringen zeitlichen Overhead von weniger als einer Mikrosekunde [Ros09]. Der Einsatz innerhalb des Schedulers ist unproblematisch, da die Funktion erst aufgerufen wird, nachdem die Runqueue nach dem Kontextwechsel bereits nicht mehr gesperrt ist. Da die Protokollierung nur einmal am Ende des Intervalls T ausgeführt wird, sind die zeitlichen Eingriffe praktisch vernachlässigbar. Um für das Debugging den Ftrace-Ausgabe-Puffer zu benutzen, wurden in der Konfigurationsdatei der Kompilierung des Linux-Kerns die Einstellungen `CONFIG_FUNCTION_TRACER`, `CONFIG_FUNCTION_GRAPH_TRACER`, `CONFIG_STACK_TRACER` und `CONFIG_DYNAMIC_FTRACE` aktiviert.

4.4 Wahl des Intervalls T

Das Intervall T sollte so gewählt sein, dass es lang genug ist, um eine Häufung von vielen Cache-Misses bei einem Wechsel der Arbeitsmenge eines Prozesses feststellen zu können. Zeitgleich sollte die Zeit d_{max} aus Abschnitt 3.6 nicht überschritten werden. Je schneller ein Prozessor getaktet ist, desto weniger Zeit ist nötig, um eine Häufung

von Cache-Misses feststellen zu können, da pro Zeiteinheit mehr Befehle auf der CPU ausgeführt werden. Damit kann die Länge des Intervalls T für schneller getaktete Prozessoren verringert werden, muss aber für langsam getaktete Prozessoren entsprechend vergrößert werden. Eine Verknüpfung mit der Taktfrequenz erscheint sinnvoll.

Da der Wert für d_{max} sehr stark von der Kombination aus verwendeten Speicher und konkreter CPU-Architektur abhängig ist, fällt es schwer für eine allgemeine Implementierung einen guten Wert für T zu ermitteln. Es muss beachtet werden, dass d_{max} für jede Anwendung in Abhängigkeit von der Wahrscheinlichkeit p anders ausfallen kann. Eine allgemeine Wahrscheinlichkeit p für alle zu erwartenden Prozesse existiert nicht. Der Erwartungswert und die Varianz müssen empirisch für eine große Menge an Prozessen ermittelt werden. Dies würde den Rahmen dieser Diplomarbeit sprengen.

Während der Implementierung wurden mehrere Periodenlängen für das Zeitintervall T experimentell implementiert. Eine Periodenlänge von fünf Millisekunden scheint ausreichend zu sein (siehe Abschnitt 5.3ff), um bei allen untersuchten Prozessen Migration Sweetspots finden zu können. In wieweit das Intervall sich noch verkürzen lässt, muss weiter untersucht werden.

4.5 Kernel-Modul zur Erzeugung des Intervalls T

Die aus dem Abschnitt 3.4 vorgestellte Methode der Implementierung erfordert die Erzeugung einer Unterbrechung der Ausführung der zu migrierenden Prozesse, um nach jedem Intervall T feststellen zu können, ob ein Migration Sweetspot aufgetreten ist. Auf der CPU laufende Prozesse können jedoch nur unterbrochen werden, wenn ein anderer höher priorisierter Prozess bereit wird. Für diesen Zweck wurde eigens ein Kernel-Modul implementiert, welches auf jeder logischen CPU einen Kernel-Thread startet.

Die Kernel-Threads führen nur eine Endlosschleife, deren einzige Aktion der Aufruf der Funktion `usleep_range()` ist. Diese Funktion versetzt den Kernel-Thread in einen blockiert-Zustand und gibt die CPU für das Intervall T an andere Prozesse ab. Die Kernel-Threads schlafen somit die meiste Zeit und beanspruchen die CPU nicht. Die Funktion `usleep_range()` wurde gewählt, da dies die empfohlene Schlaf-Mechanismus von Prozessen für Intervalle von 10 µs – 20 ms Länge ist [lka13a].

Bei der Erstellung der Kernel-Threads wird ihnen die niedrigste Echtzeit-FIFO Priorität `MAX_RT_PRIO`-1 zugewiesen. Damit ist sichergestellt, dass ein beliebiger Benutzer-Prozess nach Ablauf eines Intervalls T sicher durch den Kernel-Thread unterbrochen

wird. Gleichzeitig werden andere Kernel-Threads, die typischerweise Treiberfunktionen übernehmen und für die Systemperformance kritische Aufgaben erledigen und mit einer höheren Priorität laufen, nicht behindert. Durch die kurze Präemptierung der zu migrierenden Prozesse wird entweder beim Kontextwechsel vom migrierenden Prozess zum Kernel-Thread oder beim Ende der Präemptierung – beim Kontextwechsel vom Kernel-Thread zum migrierenden Prozess – die Zeit der Periodendauer seit dem Start der letzten Intervalls T überschritten und die Funktion `kk_perf_log_task()` wird genau einmal ausgeführt.

Für den Fall, dass kurz vor Ende eines Intervalls T ein anderer höher priorisierter Prozess ausgeführt wird, der dann von dem Kernel-Thread unterbrochen wird und anschließend andere Prozesse mit ebenfalls höheren Prioritäten zunächst vor einem zu migrierenden Prozessen auf der CPU laufen, bis schließlich dieser selbst an der Reihe ist (lange Präemptierung), kann die Ausführung der Funktion `kk_perf_log_task()` nach dem eigentlichen Ende des Intervalls T nicht eingehalten werden. Durch die Nichtausführung des Prozess während dieser Phase ist das für die Suche nach Migration Sweetspots Typ 1 nicht weiter negativ zu bewerten. Allerdings wird die Erkennung von Migration Sweetspots Typ 2 dadurch zeitlich nach hinten verschoben. Dies lässt sich nicht verhindern, ohne dass die Priorität der zu migrierenden Prozesse angehoben wird.

Für die Entwicklungs- und Evaluierungsphase der Implementierung musste eine Technik implementiert werden, um den Eintritt von Benutzer-Prozessen in neue Arbeitsbereiche im Linux-Kern erfassen zu können.

Für die Protokollierung der korrekten Implementierung wurde daher in das Kernel-Modul noch eine Kommunikations-Schnittstelle für die Übermittlung von Nachrichten von Benutzer-Prozessen zum Kernel-Modul über das `proc`-Dateisystem geschaffen. Damit erhalten die zu untersuchenden Prozesse eine Möglichkeit vordefinierte Nachrichten an den Linux-Kern zu verschicken, um mit dem Umweg über das Kernel-Modul ebenfalls in den Ftrace-Ausgabe-Puffer zu schreiben.

4.6 Probleme während der Implementierung

Obwohl `perf_event` schon seit dem Kernel v2.6.31 im Linux-Kern vorhanden ist, gibt es leider nur sehr wenig Anleitungen und Dokumentation über die Implementierung. Die meisten verfügbaren Dokumente beschränken sich auf Anleitungen zur Nutzung des Programms `perf` zu Benchmarking-Zwecken, um die Performance von Benutzer-

Prozessen zu messen. Kaum ein Dokument befasst sich mit der Anwendung der `perf_event`-Funktionen im Linux-Kern. Es gibt unzählig viele Funktionen mit uneindeutigen Namen und ohne einen Kommentar oder einer Erklärung im Quellcode. Es lässt sich schwer feststellen, was bestimmte Funktionen ausführen und was beim Aufruf zu beachten ist. Ein Beispiel dafür sind: `perf_read()`, `perf_read_hw()`, `perf_event_read()`, `perf_event_read_event()`, `perf_event_read_one()` und `perf_event_read_value()`. Alle genannten Funktionen kommen für das Auslesen der Performance-Counter in Frage, allerdings gibt es kein Dokument, welches beschreibt, wann welche Funktion anzuwenden wäre. Während der Implementierung kam es daher häufiger beim Aufruf einiger Funktionen zu einer Deadlock-Situation oder einem Kernel-Ooops.

`Perf_event` benutzt für die Konfiguration der Performance-Counter ein anderes Namensschema als beispielsweise in der Intel-Dokumentation, wodurch bei einigen Konfigurationseinstellungen ein Abgleich nötig wird. Manchmal ist es nicht ersichtlich, welche Caches für die Auswahl von Ereignissen auf bestimmten Prozessoren tatsächlich durch `perf_event` ausgelesen werden. Ein weiteres Problem ist das Debugging. Durch `trace_printk()` kann man einige Abläufe im Kern protokollieren, doch wenn man die genaue Stelle im Code nicht kennt und große Strukturen zu untersuchen sind, oder man einfach den Fehler nicht reproduzieren kann, wird es mühsam. Für jeden neuen Schritt muss der Kern neukompiliert und das System neugestartet werden. Dies ist sehr zeit- und arbeitsaufwendig.

Während der Implementierung kam es häufiger zu der Situation, dass die Programmierung der Performance-Counter durch `perf_event` an bestimmten Stellen der `__schedule()`-Funktion nicht durchgeführt werden konnte. Eine Ursache ist beispielsweise eine bereits durch `perf_event` durchgeführte Sperre des `perf_event_context`-Kontextes des Prozesses. Bei einer Programmierung der Counter zum falschen Zeitpunkt wurde nochmals versucht der Kontext zu sperren und ein Deadlock war die Folge.

5 Evaluierung

Zur Überprüfung und Auswertung der Implementierung aus dem letzten Kapitel, wurde eine Evaluierung durchgeführt. Zunächst wird im Abschnitt 5.1 das Testsystem beschrieben und einige Vorbemerkungen zur Unterstützung von weiteren Prozessoren gemacht. Im nachfolgenden Abschnitt 5.2 wird vorgestellt, wie Protokolldateien zur Evaluierung der Testläufe erstellt und wie die in diesem Kapitel gezeigten Cache-Zugriffs- und Migration-Sweetspot-Diagramme erzeugt werden. Eine kurze Analyse zur Überprüfung der Korrektheit der aufgezeichneten Daten wird ebenfalls behandelt. Die durchgeführten Messungen zur Evaluierung werden im Abschnitt 5.3 vorgestellt und ausgewertet.

5.1 Testsystem

Während der Entwicklungsphase wurde der modifizierte Linux-Kern nicht auf echter Hardware ausgeführt. Die Implementierung wurde nur in einer mit QEMU[1] aufgesetzten virtuellen Maschine getestet. Dadurch konnte unter anderem auch der in QEMU integrierte GDB-Server zum Debugging von Kernel-Funktionen genutzt werden. GDB erlaubt es auch virtuelle Maschinen zu einem Breakpoint anzuhalten und anschließend den Zustand des Systems zu protokollieren. Dies funktioniert auch im Linux-Kern, wenn der Kern-Quellcode und passende Debugging-Symbole in Form der vmlinux-Datei zur Verfügung stehen. Bei einem Systemabsturz oder bei Deadlock-Situationen kann die Ausgabe des System-Kernel-Puffers auf eine Konsole umgeleitet werden, was die Fehlersuche erheblich erleichtert, insbesondere weil der Linux-Kern in solchen Situation den Inhalt des Systemstacks in den System-Kernel-Puffer schreibt.

QEMU erlaubt es die CPU-Architektur der Testmaschine mit `-cpu host` weitestgehend auch in der virtuellen Maschine nachzubilden. Durch Aktivierung des im Linux-Kern enthaltenen KDM-Virtualisierungs-Moduls innerhalb von QEMU mit

[1] QEMU ist ein Computerpgoramm, das eine virtuelle Maschine bereitstellt. Ein solches Programm wird auch als VMM (Virtual Machine Monitor) bezeichnet.

`-enable-kvm`, werden die Hardware-Virtualisierungstechniken von Intel (VT) oder AMD (AMD-V) durch die VMM der Host-Maschine nutzbar. Damit erreicht die virtuelle Maschine annähernd die gleiche Performance, wie die Host-Maschine auf der sie ausgeführt wird. Es gibt jedoch einige Unterschiede, da weiterhin weite Teile der System-Peripherie (Festplatte, Netzwerkkarte …) nur emuliert werden und ein Teil der Prozessorleistung der Host-Maschine für die Ausführung der VMM benötigt wird.

Aus diesem Grund wurden alle Messungen auf einem realem Testsystem durchgeführt. Es wurde das gleiche Testsystem, wie für die Entwicklungsphase der Implementierung, bestehend aus einem Intel Core i5-2400-Prozessor mit vier Kernen (kein Hyper-Threading) und einer Taktfrequenz von 3,1 GHz, verwendet. Der modifizierte Linux-Kern v3.9.9 aus dem Kapitel 4 lässt sich auch auf beliebigen anderen Systemen mit x86-Prozessoren ab einer CPU mit Intel Pentium starten. Allerdings ist das Profiling für die Suche nach Migration Sweetspots nur für Prozessoren mit maximal vier logischen CPUs möglich. Dies liegt an den festen Größen der Arrays, die zum Zeitpunkt der Initialisierung der Performance-Counter benutzt werden.

5.2 Überprüfung des Profilings mit Hilfe von Ftrace

Bevor eine Evaluierung mit den Mikro-Benchmarks aus Abschnitt 4.1 durchgeführt werden kann, muss die Korrektheit der Aufzeichnung von Cache-Misses und Cache-References pro Prozess überprüft werden, wozu Ftrace benutzt wurde.

Mit der Standartkonfiguration der Kern-Kompilierungseinstellungen `defconfig` ist die Ftrace bereits verfügbar [Ros09]. Über das `debugfs`-Dateisystem gibt es aus dem Userland eine Schnittstelle, um auf Ftrace-Funktionen mit Root-Rechten zuzugreifen. Das debugfs-Dateisystem wird typischerweise in `/sys/kernel/debug/` eingehängt. Bei aktiviertem Ftrace wird darin ein `tracing` Ordner erstellt, der alle Steuerelemente zur Nutzung von Ftrace beinhaltet. Mit der Ftrace können Nachrichten, die mittels `trace_printk()` im Kern geschrieben wurden, zu einem späteren Zeitpunkt in der Standartausgabe ausgegeben und mit einer Pipe anschließend in eine Textdatei gespeichert werden. Dies wurde genutzt um Protokolle des Profilings von Prozessen zu erstellen. Die dazu nötigen Arbeitsschritte werden in Listing 5 festgehalten.

```
# cd /sys/kernel/debug/tracing
# echo printk-msg-only > trace_options
# echo "" > trace
   …
   Durchführung des Profilings …
   …
#cat trace > /direcoty/datei.txt
```

Listing 5: Shell-Befehle zur Erzeugung von Protokolldateien des Profiling von Prozessen

Nachfolgend wird beschrieben, welches Format von den Protokolldateien, die sich mittels der oben genannten Arbeitsschritte erzeugen lassen, benutzt wird. Ein Beispiel einer Protokolldatei zur Veranschaulichung ist in Listing 6 zu sehen.

```
Start tracking task with pid 5414
7692481 p1 7692481 560 1864
7692481 p2 7692481 3855 6952
7692481 p4 sweetspot2
10580473 signal1
12868597 p1 5176116 351 1491
12868597 p2 5176116 1700 2858
12868597 p4 sweetspot2
…
7042701322 p1 5174271 2723 7393
7042701322 p2 5174271 0 0
7042701322 p3 sweetspot1
…
Finish tracking task with pid 5414
```

Listing 6: Beispiel zum Aufbau einer Protokolldatei

Die erste Zeile „`Start tracking task …`“ wird durch die Funktion `kk_perf_setup_task()` erzeugt und erlaubt es den korrekten Start des Profilings für eines der Programme aus dem Mikro-Benchmark nachzuvollziehen. Nach Beendigung des Profilings, wenn der beobachtete Prozess letztmalig auf der CPU ausgeführt wurde, erfolgt durch die Funktion `kk_perf_release_task()` die Ausgabe der letzten Zeile „`Finish tracking task …`“.

Alle Zeilen dazwischen werden mit einer Ausnahme (`signal1`) durch die Funktion `kk_perf_log_task()` erzeugt. Die Protokolldatei besitzt einen tabellarischen Aufbau. Die Spalten sind durch Leerzeichen getrennt. In der ersten Spalte steht immer eine

Zeitangabe, die angibt wie viele Nanosekunden seit dem Start des Profilings für das protokollierte Ereignis vergangen sind. Alle Zeilen mit den gleichen Zeitangaben hintereinander sind Ausgaben, die einer Momentaufnahme am Ende eines Intervalls T entsprechen.

Eine Zeile von Typ `p1` (zweite Spalte) gibt die Summe der aufgetretenen Cache-Misses und Cache-References eines Intervalls T des Prozesses an. Die vorletzte Spalte beinhaltet die Cache-Misses und die letzte Spalte die Cache-References, die von zwei Performance-Countern gezählt werden. Die dritte Spalte zeigt an, wie lang das Intervall T war.

Den gleichen Aufbau besitzt eine Zeile vom Typ `p2`, jedoch werden dort die Cache-Misses und Cache-References während der Präemptierung gelistet. Eine Zeile mit dem Typ `p3` kennzeichnet einen erkannten Migration Sweetspot Typ 1 und eine Zeile mit dem Typ `p4` analog dazu einen Migration Sweetspot Typ 2.

Eine Zeile mit `signal1` kennzeichnet eine Nachricht, die der beobachtete Benutzer-Prozess an das neu implementierte Kernel-Modul geschickt hat, um den Beginn eines neuen Arbeitsbereiches in der Abarbeitung des Prozesses anzuzeigen.

5.2.1 Grafische Auswertung

Um die aus dem letzten Abschnitt erzeugten Protokolldateien besser auswerten zu können, wird eine grafische Darstellung der Protokolldaten benötigt. Eine sehr einfache Möglichkeit größere in Textformat vorliegende Messdaten zu visualisieren, bietet das skriptgesteuerte Programm Gnuplot [gnp86].

Mit Hilfe von selbst erstellten Konfigurationsdateien lassen sich über Gnuplot Zeilen eines bestimmten Typs aus der Protokolldatei auslesen und in Form einer Kurve in einem Diagramm darstellen. Abbildung 1 zeigt beispielhaft ein so erstelltes Cache-Zugriffs-Diagramm, das auf Daten einer Messung mit dem Programm `change` aus dem Mikro-Benchmark basiert. Die Funktion und der Aufbau des Programms wurde bereits in Kapitel 4 im Abschnitt 4.1 erklärt.

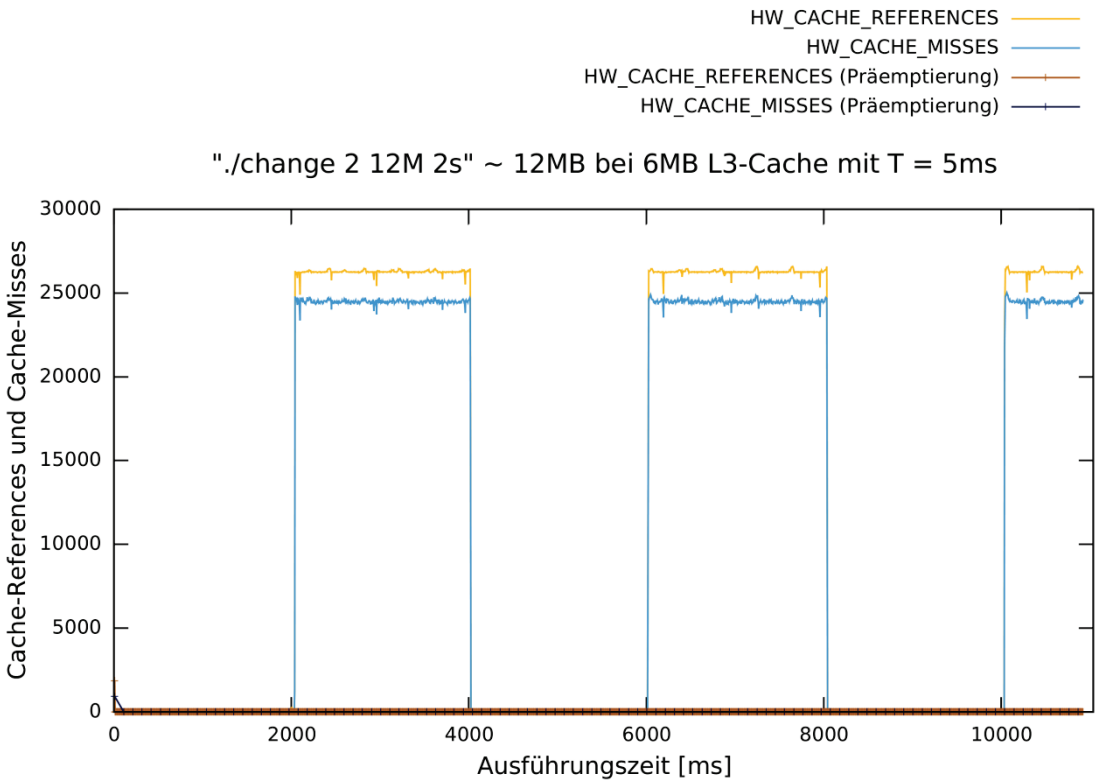

Abbildung 1: Cache-Zugriffs-Diagramm bei Ausführung des Programms change aus dem Mikro-Benchmark

Bei der Messung, die in Abbildung 1 gezeigt wird, wurde die Größe des Intervalls T auf fünf ms festgelegt. Man erkennt deutlich die unterschiedlichen Phasen der Abarbeitung. Innerhalb der ersten 2000 ms wird auf der CPU nur die `while`-Endlosschleife mit einer Abbruchbedingung (siehe Listing 7) ausgeführt. Dies ist die `cpuburner`-Phase. In dieser Phase wird nur auf eine einzelne Variable `change` zugegriffen und demzufolge treten nachdem die Variable einmal aus dem Speicher gelesen wurde, keine Cache-Misses mehr auf. Anschließend folgt für 2000 ms die `memeater`-Phase und die nächste `while`-Endlosschleife (siehe Listing 7) wird ausgeführt. Aufgrund der Größe von `size` passt der allokierte Speicherbereich von `buffer` nicht komplett in den 6 MB großen L3-Cache und man erkennt den sprunghaften Anstieg an Cache-Misses. Nach genau 2000 ms erfolgt ein Wechsel beider Phasen. Dies lässt sich im Diagramm aus Abbildung 1 nachvollziehen.

```
…
while (!change) { //cpuburner-Phase
}
…
while (!change) { //memeater-Phase
   for (unsigned long long i = 0; i < size; i++){
      buffer[i]++;
   }
}
…
```

Listing 7: Ausschnitt aus dem C++-Quellcode des Programms change

Aus Abbildung 1 ist zu erkennen, dass für die `memeater`-Phase eine obere Schranke von etwa 25000 LLC-Misses pro Intervall T = 5 ms existiert. Um zu prüfen ob die aufgenommenen Werte stimmen, wurde dieser Wert mit `perf` auf einen nicht modifizierten Linux-Kern nachgemessen und verglichen. Dazu wurde das Programm `memeater` für eine Sekunde ausgeführt und mit `perf stat –e cache-misses ./memeater 2 12M 1s` die aufgetretenen LLC-Misses gezählt. Die Gesamtzahl lag immer bei ungefähr 50 Mio., was entsprechend für einen Zeitraum von fünf ms ebenfalls rund 25000 LLC-Misses entspricht.

5.3 Messungen mit den Mikro-Benchmarks

Im Folgenden werden einige Szenarien betrachtet, um festzustellen, ob die Implementierung aus Kapitel 4 den Anforderungen für die Suche nach Migration Sweetspots entspricht. Es wurden nur Cache-Misses und Cache-References im Benutzer-Modus der CPU erfasst. Dadurch wird verhindert, dass für Migration Sweetspots Typ 2 Abläufe im Linux-Kern (Scheduling, Prozessverwaltung, virtueller Speicher usw.) die Messungen beeinflussen. Der Core-i5-Prozessor des Testrechners besitzt mehrere Cache-Prefetch-Funktionen [Hed13]. Im Einzelnen sind das:

Hardware Prefetcher: Diese Hardware-Einheit lädt Daten innerhalb einer Seite, die vorrausichtlich in naher Zukunft gebraucht werden, vorab aus dem Speicher in den L2-Cache. Dadurch treten weniger Cache-Misses auf, wenn der Speicherbus nicht komplett ausgelastet wird. Dies verhindert eine präzise Auswertung der Messergebnisse, ist aber

für die allgemeine Suche nach Migration Sweetspots nicht weiter relevant. Diese Funktion wurde für die Messungen im Bios deaktiviert, um besser Voraussagen treffen zu können, wann L3-Misses auftreten müssen. Lässt man den Hardware Prefetcher an, kann es zu Situationen kommen, wenn der L3-Cache durch einen Prozess bereits voll belegt ist und trotzdem beim Zugriff auf neue Speicherbereiche einer Seite keine LLC-Misses auftreten.

Adjacent Cache Line Prefetch: Lädt bei einem Cache-Miss nicht nur eine Cache-Line, sondern auch noch die gepaarte Cache-Line (zusammenhängender Speicherbereich) aus dem Speicher. Reduziert damit effektiv die Anzahl der auftretenden Cache-Misses. Dies verhindert ebenfalls eine präzise Auswertung der Messergebnisse und wurde für die Messungen ebenfalls im Bios deaktiviert.

DCU Prefetcher: Lädt ähnlich wie der Hardware Prefetcher Daten einer Seite, auf die häufiger zugegriffen wird, vorab in den L1-Cache. Die Daten können aus allen Ebenen der Cache-Hierarchie stammen u.a. auch aus dem Speicher. Diese Funktion konnte im Bios nicht deaktiviert werden und muss daher bei den Messungen mit berücksichtigt werden.

Während der Messungen wurde die grafische Oberfläche von der benutzten Linux-Distribution gestoppt. Alle weiteren parallel laufenden Dienste wurden allerdings nicht beendet, um eine durchschnittliche Leerlauf-Situation im Desktop-Betrieb zu simulieren.

5.3.1 Auftreten von Migration Sweetspots am Beispiel von matrixmult in Abhängigkeit der Größe der Arbeitsmenge

Es ist zu erwarten, dass Migration Sweetspots Typ 1 nur auftreten, wenn die Arbeitsmenge eines Prozesses groß genug ist, um innerhalb eines Intervalls T genug Cache-Misses zu verursachen, dass der Schwellwert S1 überschritten wird. Bei dieser Messung soll für das Programm matrixmult ermittelt werden, wie viele Sweetspots vom Typ 1 in Abhängigkeit von der Größe der Arbeitsmenge (Größe der Matrizen) auftreten.

Durchführung der Messung

Das Programm matrixmult wird für diese Messung mit unterschiedlichen Startparametern für die Matrizengröße gestartet. Die quadratischen Matrizen speichern Integer-Werte (4 Byte). Die beiden zu multiplizierenden Matrizen werden beim Programmstart initialisiert und bleiben während der gesamten Ausführungszeit unverändert. Die Multiplikation wird fünf Mal hintereinander ausgeführt. Für jede der Multiplikationen wird der Speicher der alten Zielmatrix deallokiert und neuer Speicher für die nächste Zielmatrix allokiert.

Demzufolge ändert sich bei jeder neuen Multiplikation die Arbeitsmenge des Prozesses um die Größe der Zielmatrix. Das sind demnach Zeitpunkte an denen erwartungsgemäß Migration Sweetspots Typ 1 auftreten können. Es soll geprüft werden, wie viele der Multiplikationen als Migration Sweetspots bei der Messung erkannt werden. Als Schwellwert S1 für einen Migration Sweetspot Typ 1 wird ein sehr geringes Miss-Verhältnis von S1 = 0,1 festgelegt.

Auswertung

Die Messergebnisse sind in Tabelle 8 zu sehen. Für die theoretische Anzahl der zu erwartenden Cache-Misses CM am Anfang einer neuen Multiplikation wird die nachfolgende Formel benutzt. Diese wurde genutzt, um zu überprüfen, ob die gemessenen Werte die passende Größenordnung haben. Die Herleitung ist wie folgt: Da die Matrizen Integer speichern, belegt eine Zielmatrix insgesamt $(\text{Größe der Zielmatrix})^2 \times 4\,\text{Byte}$ Speicher. Teilt man den Wert durch die Cache-Line-Größe, erhält man die Gesamtzahl der Cache-Lines, die die Zielmatrix im Cache brauchen wird. Dies entspricht auch der Anzahl der zu erwartenden Cache-Misses.

$$\text{CM} = \left(\frac{(\text{Größe der Zielmatrix})^2 \times 4\,\text{Byte}}{\text{Cache-Line-Größe}} \right)$$

Tabelle 8: Messergebnisse der Messung mit dem Programm matrixmult in Abhängigkeit der Matrizengröße

Größe der Matrizen	Benötigter Speicher für drei Matrizen (in MB)	Durchschnitt Miss-Verhältnis im Intervall T	Durchschnitt Miss-Verhältnis im ersten Intervall T nach Beginn einer neuer Multiplikation	Migration Sweetspots Typ 1 gefunden
500	2,86	<0,01	0,02	0 / 5
723	6	0,01	0,11	4 / 5
750	6,44	0,01	0,16	5 / 5
836	8	0,014	0,33	5 / 5
936	10	0,0625	0,52	5 / 5
1024	12	0,111	0,78	0 / 5

Bei einer kleinen Matrixgröße von beispielsweise 500 werden die Wechsel der Arbeitsmenge nicht als Migration Sweetspot erkannt. Laut der oben genannten Formel müssen mindestens 15625 Cache-Misses im ersten Intervall T nach Beginn einer neuen Matrixmultiplikation auftreten. Bezogen auf die gemessenen 36000 Cache-References im gleichen Intervall T, ergibt sich ein theoretisches Miss-Verhältnis von 0,43. Dieser Wert wird allerdings in den Messungen überhaupt nicht erreicht. Die Ursache ist höchstwahrscheinlich, dass nachdem der Speicher der letzten Zielmatrix deallokiert wurde, anschließend die gleichen Seiten für die neue Zielmatrix allokiert werden. Die meisten der benutzten Seiten stehen dann offenbar noch komplett im Cache und werden nicht verdrängt. Der Schwellwert S1 wird bei kleinen Matrixgrößen durchweg nicht überschritten.

Ab einer Matrixgröße von circa 720 passen die beiden zu multiplizierenden Matrizen und die Zielmatrix nicht mehr komplett in den L3-Cache (6 MB beim Core-i5-2400 Prozessor) und das ist auch der Grund wieso das durchschnittliche Miss-Verhältnis anfängt zu steigen. Der Schwellwert S1 wird nun bei Beginn einer neuen Multiplikation regelmäßig überschritten und ab einer Matrixgröße von 800 werden alle fünf Migration Sweetspots bei Beginn einer neuen Multiplikation gefunden. Beispielhaft dazu zeigt Abbildung 2 das Diagramm des LLC-Miss-Verhältnis einer Messung mit einer Matrixgröße von 836.

Bei weiter steigender Matrixgröße entspricht die Anzahl der beobachten Cache-Misses nach Beginn einer neuen Multiplikation auch tatsächlich den theoretischen Werten aus der Formel. Aufgrund der vielen durchgängigen Cache-Misses werden die Cache-Lines der alten Zielmatrix offenbar verdrängt, bevor das erste Mal auf die entsprechenden Speicheradressen zugegriffen wird.

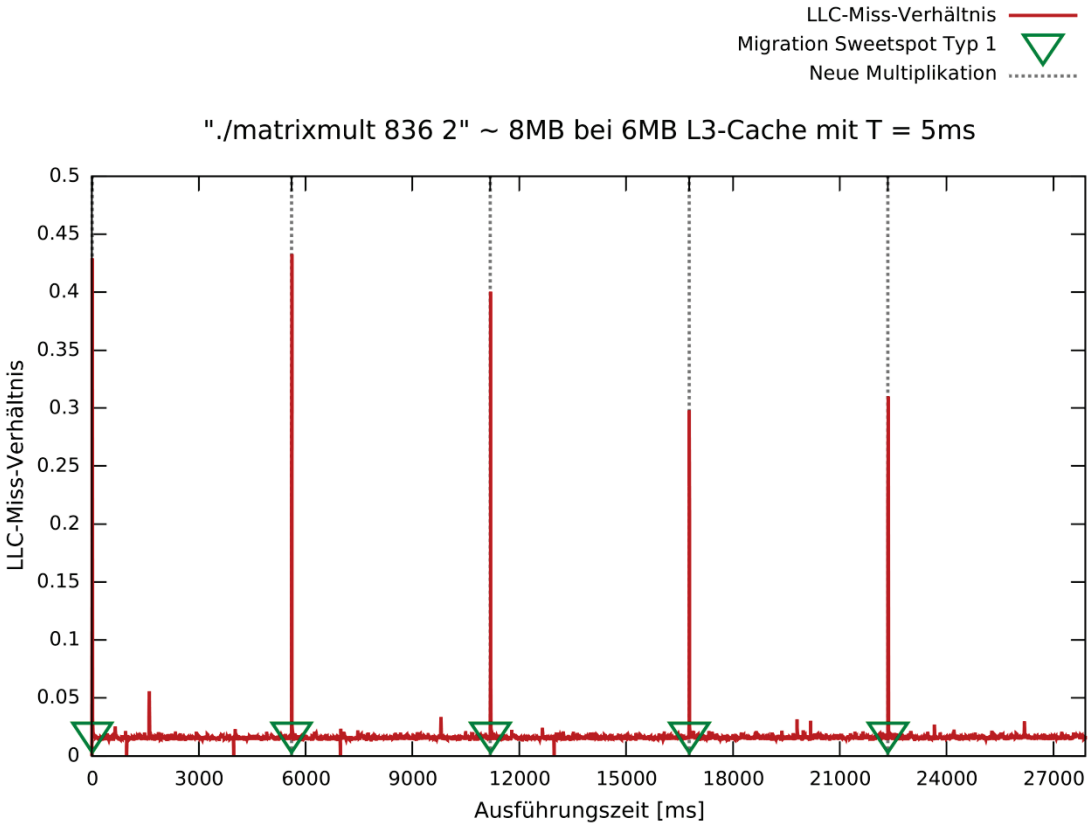

Abbildung 2: Diagramm des LLC-Miss-Verhältnis für eine Messung mit matrixmult mit Markierung der gefunden Migration Sweetspots Typ 1

Ab einer Matrixgröße von rund 1024 steigt das Miss-Verhältnis konstant über den Schwellwert S1. Dies weist darauf hin, dass die Matrixmultiplikation konstant verzögert wird, weil der Prozessor immer wieder neue Daten aus dem Speicher nachladen muss.
Es gibt somit keinen Grund mehr eine verzögerte Migration einzuplanen. Demzufolge können auch keine Migration Sweetspots Typ 1 mehr gefunden werden. Jedoch treten nun häufiger Migration Sweetspots Typ 2 auf, da immer mehr Cache-Lines mit Daten anderer Prozesse durch die Matrixmultiplikation verdrängt werden und andere Prozesse damit Daten aus dem Speicher nachladen müssen.

5.3.2 Parallele Ausführung matrixmult und memeater

Bei Überlastsituationen ist es wahrscheinlich, dass mehrere Prozesse sich gegenseitig behindern und um den Platz im Cache konkurrieren. Mit dieser Messung soll gezeigt werden, dass das Auftreten von Migration Sweetspots durch parallel laufende Prozesse beeinflusst werden kann. Auch Prozesse mit normalerweise wenigen Migration Sweetspots können durch eine solche Überlastsituationen Migration Sweetspots (sowohl Typ 1 als auch Typ 2) bekommen.

Durchführung der Messung

Die Messung beginnt mit der Ausführung von matrixmult mit einer relativ kleinen Matrixgröße von 400. Beginnend bei der sechsten Iteration wird nach jeder vierten Iteration parallel memeater für die Dauer einer Sekunde ausgeführt. Der Verlauf des Miss-Verhältnisses und die dazugehörige Verteilung der Migration Sweetspots für eine solche Messung ist in Abbildung 3 zu sehen. Es ergeben sich vier Überlastsituationen. Aus dem letzten Abschnitt ist bekannt, dass bei der Matrixmultiplikation von dieser Größe keine Migration Sweetspots Typ 1 zu erwarten sind.

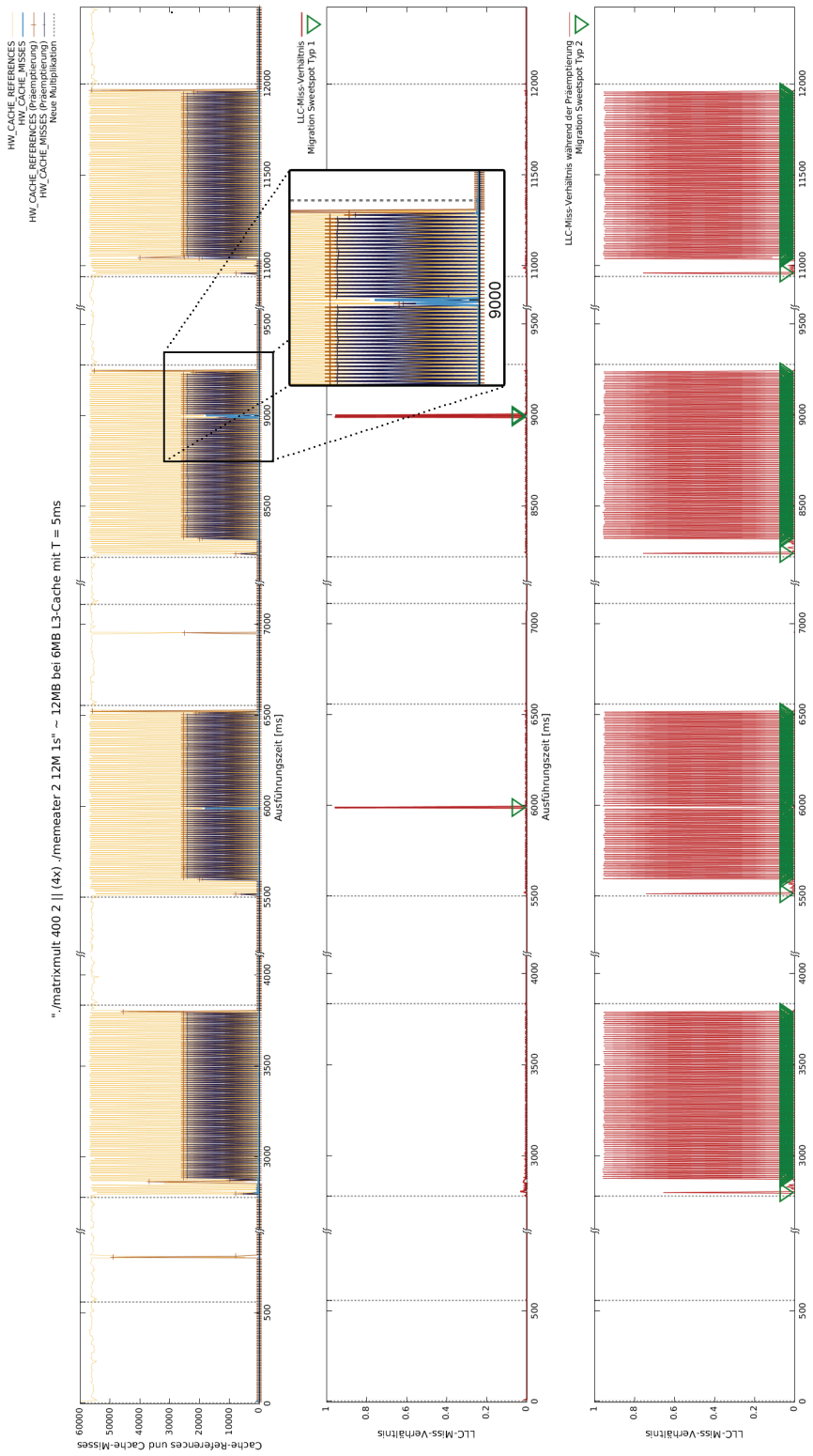

Abbildung 3: Cache-Miss-Diagramm und Migration-Sweetspot-Diagramme für eine Messung mit paralleler Ausführung von matrixmult und memeater

Auswertung

Während der Überlastsituationen kann beobachtet werden, wie beide Prozesse sich nach jedem Intervall T auf der CPU abwechseln. Die Ursache dafür ist, dass beide Prozesse CPU-intensive Prozesse sind. Demzufolge haben Sie die gleiche Priorität und besitzen die gleiche Zeitscheibenlänge (100 ms bei einer Priorität von 120). Da die zwei Prozesse damit in der gleichen Liste des `active`-Arrays stehen, werden Sie nach der Unterbrechung durch den Kernel-Thread des implementierten Kernel-Moduls abwechselnd ausgeführt, solange es keine anderen höher priorisierten rechenbereiten Prozesse gibt, oder einer der Prozesse seine Zeitscheibenlänge komplett aufgebraucht hat. Die Wahrscheinlichkeit, dass der `memeater`-Prozess während seiner Ausführung Cache-Lines des `matrixmult`-Prozesses verdrängt steigt. Dabei würde anschließend das Miss-Verhältnis des `matrixmult`-Prozesses sprunghaft ansteigen und bei Überschreitung des Schwellwerts S1 tritt ein Migration Sweetspots Typ 1 auf. Dies ist in der Abbildung 3 bei etwa 6000 ms und 9000 ms zu sehen.

Es lässt sich auch sehr gut erkennen, wie während der Ausführung von `memeater`, dauerhaft Migration Sweetspots Typ 2 für den `matrixmult`-Prozess zu beobachten sind. Dies kann als frühzeitige Warnung für die kommenden Cache-Misses des Prozesses gedeutet werden.

5.3.3 Migration zu einem Migration Sweetspot Typ 1

Um nachzuweisen, dass eine verzögerte Migration zu einem Migration Sweetspot Typ 1 eine Reduzierung der Laufzeitverzögerungen eines Prozesses bewirken kann, wird eine Messung mit einer modifizierten Version von `matrixmult` durchgeführt.

Durchführung der Messung

Das Programm matrixmult wird mit der Matrixgröße von 146 gestartet, was für drei Matrizen einen Speicherbedarf von etwa 250 KB entspricht. Die Größe wird so gewählt, damit die Arbeitsmenge für eine Matrixmultiplikation komplett in den L2-Cache einer CPU passt, weil der L2-Cache die letzte Cache-Ebene ist, die nicht über mehrere CPUs hinweg geteilt wird. Die Matrixmultiplikation (A•B = C) wird sehr oft hintereinander ausgeführt um sicherzugehen, dass die beiden zu multiplizierenden Matrizen A und B komplett im L2-Cache stehen.

Bei der letzten Wiederholung werden zwei unterschiedliche Varianten untersucht. Im Fall A wird mit `sched_setaffinity()` eine Migration auf eine andere CPU nach der Hälfte der Berechnung der Zielmatrix erzwungen. Die zweite Variante (Fall B) ist eine Migration direkt vor dem Anfang der Berechnung der Zielmatrix. Aus dem Abschnitt 5.3.1 ist bekannt, dass im Fall B die Migration zu einem Migration Sweetspot Typ 1 stattfindet.

In beiden Fällen wird mit Hilfe der Performance-Counter erfasst wie viele LLC-References und LLC-Misses während der Multiplikation im Durchschnitt auftreten. Beide Versuche werden mehrmals hintereinander abwechselnd ausgeführt. Die erfassten Werte sind in Tabelle 9 zu sehen.

Tabelle 9: Messergebnisse der Messung zur Migration zu einem Migration Sweetspot Typ 1 im Vergleich zu einer willkürlichen Migration

	Durchschnitt LLC-References vor der letzten Multiplikation	Durchschnitt LLC-Misses vor der letzten Multiplikation	Durchschnitt LLC-References während der letzten Multiplikation	Durchschnitt LLC-Misses während der letzten Multiplikation
Fall A (Migration nach der Hälfte der Matrix-multiplikation)	4314	<1	6658	2
Fall B (Migration zum Migration Sweetspot)	4243	<1	4301	2

Auswertung

Im Fall A muss für die Beendigung der Matrixmultiplikation von der neuen CPU noch auf mindestens die Hälfte der Daten der Matrix A, alle Daten der Matrix B und die Hälfte der Daten der Zielmatrix C zugegriffen werden. Diese Daten können nicht aus dem L1- oder L2-Cache der CPU, zu der migriert wurde kommen, sondern müssen aus dem gemeinsam genutzten LLC geladen werden. Es wird erwartet dass im Fall A deutlich mehr LLC-References auftreten.

Die durchschnittlichen LLC-References betragen in beiden Fällen für eine Multiplikation ohne Migration in etwa 4300. Im Fall A treten bei der letzten Multiplikation mit Migration nach der Hälfte der Berechnung etwa 50% mehr LLC-References auf. Das sind alles Speicherzugriffe, die nicht zu einem L1- oder L2-Hit geführt haben und damit musste auf einen Cache-Hit bzw. Cache-Miss im LLC gewartet werden.

Auf dem Testrechner dauert ein Speicherzugriff, der mit einem LLC-Hit beantwortet werden konnte laut Tabelle 4 mindestens 65 Takte länger als ein Speicherzugriff der zu einem L1- oder L2-Hit geführt hat, wenn die betreffende Cache-Line auf einem anderen CPU modifiziert worden ist, wovon hier auszugehen ist. Bei 2300 zusätzlichen LLC-References ergibt das etwa eine Verzögerung von mindestens 149500 Takten (65×2300). Im Fall B kommt es zu keiner nennenswerten Verzögerung durch zusätzliche LLC-References. Die Reduzierung der Laufzeitverzögerungen durch eine Migration zu einem Migration Sweetspot Typ 1 kann damit nachgewiesen werden.

5.3.4 Migration zu einem Migration Sweetspot Typ 2

Diese Messung erbringt den Nachweis, dass eine Migration zu einem Migration Sweetspot Typ 2 sich positiv auf die Ausführungszeit eines Prozesses auswirken kann. Es wird die gleiche Wirkung erzielt, wie bei einem Lastausgleich.

Durchführung der Messung

Die Messung ähnelt der Messung aus 5.3.2. Das Programm `matrixmult` wird mit unterschiedlichen Matrizengrößen gestartet. Nach drei durchgeführten Matrixmultiplikationen wird parallel das Programm `cpuburner` für eine Sekunde gestartet, wodurch eine Überlastsituation auf der benutzten CPU entsteht. Es wird die Gesamtausführungszeit für die Ausführung von 18 Matrixmultiplikationen gemessen. Einmal ohne Migration (Fall A) und ein zweites Mal mit einer Migration zu einer anderen CPU sofort nach Erkennung des ersten Migration Sweetspots Typ 2 nach dem Start von `cpuburner`. Die Migration wird wieder mit `sched_setaffinity()` durch `matrixmult` erzwungen. Die Messergebnisse sind in Tabelle 10 festgehalten. Die Anzahl der durchgeführten Matrixmultiplikationen vor und nach der Migration sind völlig willkürlich gewählt, ermöglichen aber eine gute Auswertung der Messergebnisse mit dem Cache-Zugriffs-Diagramm.

Tabelle 10: Messergebnisse zur Messung für die Migration zu einem Migration Sweetspot Typ 2

	Durchschnitt Gesamtausführungszeit von matrixmult (Matrixgröße 400)	Durchschnitt Gesamtausführungszeit von matrixmult (Matrixgröße 1024)
Fall A (keine Migration)	7187 ms	131236 ms
Fall B (Migration zu Migration Sweetspot Typ 2)	6685 ms	130744 ms
Verbesserung Gesamt-ausführungszeit Fall B im Vergleich zu Fall A	502 ms	492 ms

Auswertung

Es zeigt sich, dass im Fall A durch die zusätzliche Ausführung von `cpuburner` die Zeit in der `matrixmult` auf der CPU ausgeführt wird, halbiert wird. Dadurch verlängert sich die Ausführungszeit für eine Iteration von ca. 550 ms auf ungefähr den doppelten Wert 1050 ms. Das ergibt eine Differenz von 500 ms. Das ist zu sehen im Cache-Zugriffs-Diagramm in Abbildung 4.

Im Fall B wird der `matrixmult`-Prozess nicht durch den `cpuburner`-Prozess behindert, da er nach der Migration ungehindert von der anderen CPU ausgeführt wird. Die Gesamtausführungszeit verringert sich gegenüber dem Fall A um ca. 502 ms Sekunden. Das entspricht fast exakt der Differenz der Zeit, die im Fall A verloren ging. Für die Matrizengröße von 1024 ergibt sich eine analoge Situation. Die Verbesserung der Gesamtausführungszeit liegt im selben Größenbereich, wie bei einer Matrixgröße von 400.

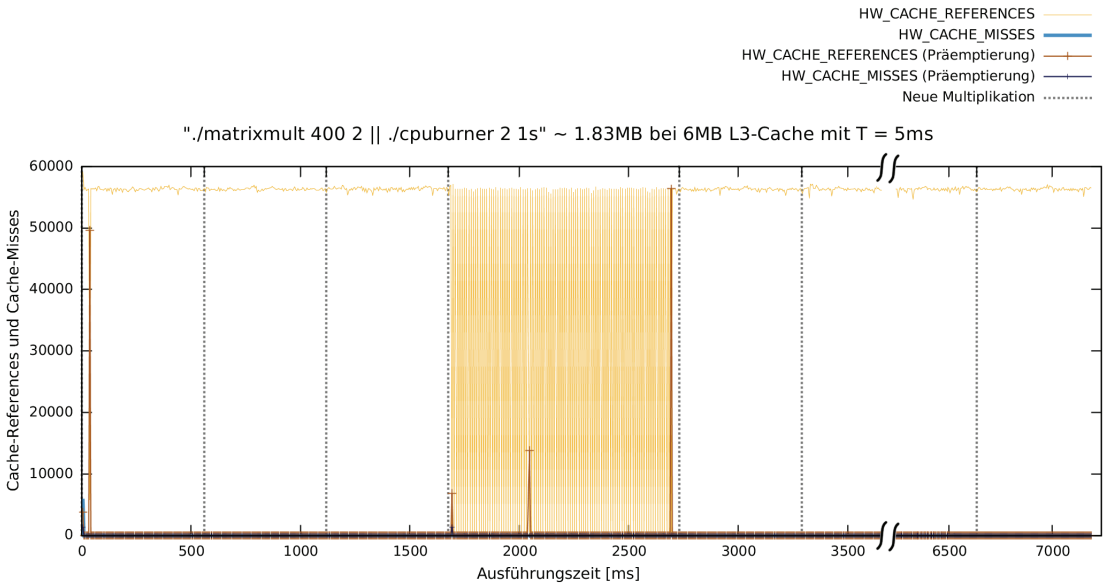

Abbildung 4: Cache-Zugriffs-Diagramm für eine Ausführung der Messung zur Migrationen zu einem Migration Sweetspot Typ 2 ohne Migration (Fall A)

6 Zusammenfassung

Migration Sweetspots bieten eine Gelegenheit die Scheduler moderner Betriebssysteme für Mehrkernprozessorsystem weiter zu optimieren. In dieser Arbeit wurde gezeigt, dass der Scheduler – auch ohne genauer Kenntnis der Algorithmen der von einem Prozessor ausgeführten Prozesse mit Hilfe von Performance-Counter für die Ermittlung und Beobachtung von aufgetretenen Cache-Misses und Cache-References – Migration Sweetspots finden kann.

Anhand einer ersten Implementierung für den Linux-Kern konnte gezeigt werden, dass die bereits im Kernel integrierte `perf_event`-Schnittstelle sich gut dafür eignet, um mit Performance-Countern nach Migration Sweetspots zu suchen. Mit nur zwei Performance-Countern pro logische CPU können für alle ausgeführten Prozesse Migration Sweetspots zur Laufzeit gesucht werden. Dafür werden LLC-Misses und LLC-References innerhalb eines kurzen Intervalls gezählt. In der Arbeit werden zwei unterschiedliche Typen von Migration Sweetspots definiert und analysiert. Migration Sweetspots Typ 1 können auftreten, wenn die Arbeitsmenge eines Prozesses wechselt und eine neue Abarbeitungsphase in der Ausführung des Prozesses beginnt. Mit Migration Sweetspots Typ 2 kann während der Präemptierung des Prozesses abgeschätzt werden, ob eine Migration bereits vor der erneuten Ausführung des Prozesses sinnvoll wäre. Beide Typen wurden im Evaluierungskapitel mit Zuhilfenahme mehrerer ausgewählter Programme eines Mikro-Benchmarks untersucht.

Es hat sich rausgestellt, dass bei Prozessen mit einer sehr kleinen Arbeitsmenge, die nur sehr wenige Cache-Lines im Cache belegen und diesen nicht komplett ausfüllen, kaum Migration Sweetspots Typ 1 zu erwarten sind. Eine verzögerte Migration bringt hier keine Vorteile. Bei Überlastsituationen, wenn andere Prozesse Cache-Lines mit Daten des beobachteten Prozesses verdrängen, können jedoch unerwartet Migration Sweetspots Typ 1 auftreten. Diese besonderen Situationen können ebenfalls in Migrations-Entscheidungen einbezogen werden.

Für Prozesse mit einer großen Arbeitsmenge ist das Auftreten von Migration Sweetspots Typ 1 unwahrscheinlich, wenn die Arbeitsmenge deutlich über der Größe des LLC liegt. In diesem Fall verliert der Cache seine positive Wirkung, da der Prozess ständig auf wechselnde Daten zugreift, die häufiger nicht mehr im Cache stehen. Es treten sehr viele Cache-Misses auf und der Prozess wird nur durch die Bandbreite des Speicherbuses limitiert. Eine Migration zu einer andren CPU bringt keine Vorteile.

Die größtmöglichen positiven Laufzeitverringerungseffekte durch eine Migration eines Prozesses zu einem Migration Sweetspot werden dort erzielt, wo die Arbeitsmenge normalerweise in den LLC passt, dann aber schlagartig verändert wird, so dass größere Datenbereiche aus dem Speicher nachgeladen werden müssen.

Der derzeitig in Linux verwendete Scheduler führt einen Lastausgleich in regelmäßigen Abständen ohne Rücksicht auf die Folgen für die nächsten Speicherzugriffe der zu migrierenden Prozesse durch. Durch die Einbeziehung der Ergebnisse bei der Suche nach Migration Sweetspots kann dieser Vorgang verbessert werden, indem Prozesse mit einem „heißen“ Cache erst dann migriert werden, wenn der Cache abgekühlt ist. Eine weitere Untersuchung mit einer Vielzahl von realen Anwendungen wäre lohnenswert, um die Wirksamkeit der Implementierung zu bestätigen. Für die Zukunft können noch einige weiterführende Arbeiten das Thema weiter vertiefen:

Kompatibilität zu perf_event. Es ist nicht auszuschließen, dass durch die Modifizierung des Linux-Kerns keine Inkompatibilitäten mit `perf_event` oder anderen Teilen des Kernels, die mit Performance-Countern arbeiten, entstanden sind. Es wäre auch noch zu untersuchen, ob die Stabilität des Kernels nicht durch die Änderungen am Scheduler beeinflusst wird.

Einbeziehung weiterer Performance-Counter: Die derzeitige Implementierung wertet nur das LLC-Miss-Verhältnis für die Suche nach Migration Sweetspots aus. Denkbar wäre auch eine andere Metrik für die Suche mit einzubeziehen, beispielsweise das TLB-Miss-Verhältnis oder die relative Anzahl der Seitenfehler oder das Prefetch-Miss-Verhältnis.

Anpassungen der Zeitkonstanten. Für die Suche nach Migration Sweetspots wird innerhalb der Implementierung eine periodische Unterbrechung (Intervall T) der laufenden Prozesse durch höher priorisierte Kernel-Threads durchgeführt. Es wäre noch zu untersuchen in wie weit das Intervall T sich verkürzen lässt, ohne dass der Overhead der Kontextwechsel die Laufzeitverbesserungen durch Migration Sweetspots wieder vernichtet.

Test mit weiterer Hardware. Interessant wäre auf auch die Messungen auf weiteren x86-Prozessoren mit anderen Cache-Größen zu wiederholen. Da sich Cache-Architekturen teilweise sehr stark unterscheiden, sollte das Auftreten von Migration Sweetspots direkt in Zusammenhang mit der Größe des LLC stehen. Außerdem wäre zu erwarten, dass bei einem Prozessor mit nicht geteiltem LLC eine Migration zu einem Migration-Sweetspot

deutlich mehr Auswirkungen auf die Laufzeit eines Prozesses hat, als bei einem von allen CPUs gemeinsam genutzten LLC.

Test und Messungen mit weiteren Anwendungen. In wieweit Migration Sweetspots sich auch bei den typischen Algorithmen wie beispielsweise A*-Algorithmus, Encodierung mit h264 oder gcc-Kompilierung in gängigen Anwendungen finden lassen, muss noch weiter untersucht werden.

III Literaturverzeichnis

[Aas05] Josh Aas. *Understanding the Linux 2.6.8.1 CPU Scheduler*. Februar, 2005.

[amd04] AMD Corporation. *AMD Demonstrates World's First X86 Dual-Core Processor*. August, 2004. http://www.amd.com/us/press-releases/Pages/Press_Release_89872.aspx.

[AMD11] AMD Corporation. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. May, 2011.

[BCC08] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Dai Yuehua, Yang Zhang, Zheng Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementaion*, 2008.

[Col13] Robert Colwell. *The Chip Design Game at the End of Moore's Law*. Hot Chips, August, 2013. http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.15-keynote1-Chipdesign-epub/HC25.26.190-Keynote1-ChipDesignGame-Colwell-DARPA.pdf.

[CSR99] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, Yale N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the international* symposium *on Computer Architecture*, 1999.

[Fre07] Free Software Foundation. *GNU General Public License*. Januar, 2007. http://www.gnu.org/licenses/gpl.html.

[gnp86] Gnuplot: . 1986. http://www.gnuplot.info/.

[Hed13] Ravi Hedge. Intel Software: *Optimizing Application Performance on Intel® Core™ Microarchitecture Using Hardware-Implemented Prefetchers*. 2013. http://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers/.

[Her11] Paul Herrmann. *Rechnerarchitektur: Aufbau, Organisation und Implementierung, inklusive 64-Bit-Technologie und Parallelrechner*. Wiesbaden, Vieweg+Teubner, 2011.

[HMN09] Daniel Hackenberg, Daniel Molka, Wolfgang E. Nagel. *Comparing Cache Architectures and* Coherency *Protocols on x86-64 Multicore SMP Systems*. Dezember, 2009.

[int08] Intel Corporation. *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500* processors. 2008.

[int11] Intel Corporation. *Intel® 64 and IA-32 Architectures: Volume 3B: System Programming Guide, Part 2*. Dezember, 2011.

[int13a] Intel Corporation. *Intel® Architecture Instruction Set Extensions Programming Reference*. Juli, 2013.

[int13b] Intel Corporation. *Intel® Xeon® Processor E5-2697 v2*. 2013. http://ark.intel.com/products/75283/.

[KLW04] Dongkeun Kim, Steve Shih-wei Liao, Perry H. Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, John P. Shen. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *International Symposium on Code Generation and Optimization*, 2004.

[KST11] Md Kamruzzaman, Steven Swanson, Dean M. Tullsen. *Inter-core Prefetching for Multicore* Processors *Using Migrating Helper Threads*. März, 2011.

[lif13] Linux Foundation. *Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing,* and *Who is Sponsoring It*. September, 2013. http://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2013.

[lin09] Linux Programmer's Manual. Linux.com: *Linux Programmer's Manual*. April, 2009. http://www.linux.com/learn/docs/man/.

[lin13] Linux Programmer's Manual. *PERF_EVENT_OPEN*. September, 2013. http://web.eece.maine.edu/~vweaver/projects/perf_events/perf_event_open.html.

[lka13a] The Linux Kernel Archives: *delays - Information on the various kernel delay / sleep mechanisms*. 2013. https://www.kernel.org/doc/Documentation/timers/timers-howto.txt.

[lka13b] The Linux Kernel Archives: *trace_printk*. 2013, . https://www.kernel.org/doc/htmldocs/device-drivers/API-trace-printk.html.

[lka99] The Linux Kernel Archives: *T H E /proc F I L E S Y S T E M*. Oktober, 1999. https://www.kernel.org/doc/Documentation/filesystems/proc.txt.

[Lov05] Robert Love. *Linux Kernel Development: Second Edition*. Indianapolis, Novell Press, 2005.

[Ros09] Steven Rostedt. LWN.net: Linux from the source: *Debugging the kernel using Ftrace - part 1*. Dezember, 2009. http://lwn.net/Articles/365835/.

[Sig10] Benoit Sigoure. Tsuna's blog: *How long does it take to make a context switch?* November, 2010. http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html.

[Tan09] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. München, Pearson Studium, 2009.

[Tor01] Linux Torvalds, David Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. New York, HarperCollins, 2001.

[VIA05] Neil Vachharajani, Matthew Iyer, Chinmay Ashok, Manish Vachharajani, David I. August, Daniel Connors. *Chip Multi-Processor Scalability for Single-Threaded Applications*. September, 2005.

[Wea13] Vince Weaver. *Linux* perf_*event Features and Overhead*. 2013 FastPath Workshop, April, 2013. http://web.eece.maine.edu/~vweaver/projects/perf_events/overhead/fastpath2013_perfevent_slides.pdf.

[ZS01] Craig Zilles, Gurindar Sohi. Execution-based Prediction Using Speculative Slices. In Proceedings *of the International Symposium on Computer*, 2001.

A Anlagen vom Typ1

A 1. Anlagenbeispiel 1

```
Some Code ...
```