Skalierbarkeit mikrokernbasierter Systeme

Volkmar Uhlig

Universität Karlsruhe
IBM T.J. Watson Research, Yorktown Heights, NY

1 Einleitung

Mikrokernbasierte Systeme lösen das Problem ständig wachsender Komplexität monolithischer Betriebssysteme durch Verlagerung der Systemfunktionalität vom privilegierten in den nicht-privilegierten Modus. Der Mikrokern fungiert dabei als reiner Ressourcenmultiplexer, und die Ressourcenmanagementstrategie befindet sich außerhalb des Kernes.

Mikrokerne haben mit wachsender Parallelität von Multiprozessorsystemen ein inhärentes Skalierbarkeitsproblem. Synchronisationsmechanismen mit geringem Laufzeitaufwand innerhalb des Kernes sollten grob-granulare oder keine Sperren einsetzen. Fein-granulare Synchronisation hingegen liefert bessere Skalierbarkeit für häufig parallel angefragte Ressourcen. Die Literatur unterscheidet bis zu zehn unterschiedliche Synchronisationsschemata.

Kernalgorithmen in monolithischen Betriebssystemen haben Zugriff auf höhere semantische Informationen, die es erlauben, die Implementierung für den individuellen Anwendungsfall zuzuschneiden. Diese Möglichkeit existiert jedoch nicht in mikrokernbasierten Systemen, da sich diese semantischen Informationen außerhalb des auf physischen Ressourcen operierenden Kernes befinden. In dieser Arbeit wird der Grad der Parallelität von kernkontrollierten Ressourcen als eine applikationsgesteuerte Ressourceneigenschaft eingeführt. Die Synchronisationsmechanismen für individuelle Ressourcen können dabei abhängig vom jeweiligen Anwendungsfall dynamisch und sicher von Applikationen zur Laufzeit angepaßt werden.

2 Kernskalierbarkeit

Der typische Ansatz zur Verbesserung der Skalierbarkeit von Betriebssystemen ist, mit Hilfe einer repräsentativen Last, die Verstopfungen zu identifizieren und grob-granulare, durch fein-granulare Sperren zu ersetzen. Dieser Ansatz skaliert nur beschränkt, da mit jeder weiteren Sperre die Kosten steigen. Mit wachsender Anzahl an Prozessoren saturiert das System ultimativ, entweder wegen Verstopfungen oder der hohen Kosten für Sperren. Basierend auf Warteschlangentheorie hat Unrau [Unr93] drei Designprinzipien für

skalierbare Systeme entwickelt: (1) das Betriebssystem muß die Parallelität der Applikation im Kern erhalten, (2) die Kosten für Systemoperationen müssen unabhängig von der Anzahl der Prozessoren sein und (3) das Betriebssystem muß die Speicherlokalität der Applikationen erhalten. Wenn man diese Entwurfsprinzipien auf Mikrokerne anwendet, erhält man sehr restriktive Designanforderungen. Da der Mikrokern kein Wissen über mögliche Ressourcenabhängigkeiten besitzt, müssen alle Systemressourcen (wie Speicher, Unterbrechungsquellen, etc.) unabhängig, parallel und in der feinstmöglichen Granularität verwaltet werden. Da der Kern außerdem nicht über Wissen um die maximale Parallelität verfügt, müßte er die am Besten skalierenden, aber damit auch die teuersten Synchronisationsprimitive verwenden. Dies steht jedoch im Konflikt zu einem anderen Designziel für Mikrokerne, nämlich ein System mit minimalen Kosten zu konstruieren.

Die Kernidee dieser Arbeit ist, den Mikrokern mit hinreichenden Kontextinformationen über den Grad der Parallelität und den zu erwartenden Zugriffsmustern auf Kernressourcen zu versorgen, so daß der Kern seine Synchronisationsprimitive sicher und dynamisch den Anforderungen und Gegebenheiten der jeweiligen Applikation anpassen kann. Ausgehend vom höchsten Grad der Skalierbarkeit (und damit den höchsten Kosten) werden die Primitive sukzessive für bessere Performanz optimiert.

Die betrachteten Optimierungen sind (1) die Vereinigung mehrerer kritischer Abschnitte in einen kombinierten kritischen Abschnitt, (2) selektive Wahl des Sperrenprimitives [LLG+92, GVW89] abhängig vom Speichersubsystem (z.B. Spinlocks und MCS-Locks [MCS91]), (3) Operationen, die primär auf nur einem Prozessor ausgeführt werden, sollten keine Sperren verwenden und (4) Entzug von Speicherressourcen aus dem virtuellen Adreßraum sollte minimale Kosten für die Kohärenz des *Translation Look-Aside Buffers* (TLB) haben.

Alle Optimierungen haben die Voraussetzung, daß die Sicherheit des Kernes nicht kompromittiert werden kann. Der Mikrokern muß daher mögliche Parallelität vermerken und mögliche Verletzungen des Vertrags zwischen Applikation und Kern verhindern.

2.1 Ressourcenverwaltung

Optimierungen für Multiprozessorsysteme verlangen eine Abwägung von besserer Performance gegen bessere Skalierbarkeit. Optimierungen sind inhärent plattformspezifisch und beschränkt durch die physikalischen Eigenschaften des Systems.

Standard-Speicherbussysteme skalieren nicht mit wachsender Anzahl Prozessoren was zu Strukturen wie Hyperwürfel [HJ86] und fetten Bäumen [Lei85] in Speichersystemen geführt hat. Diese Systeme geben Performance für geringe Komplexität auf und nur wenige Prozessoren haben eine direkte Speicherkommunikationsverbindung wie in Bussystemen. Da nichtuniformer Speicher höhere Kosten für entfernte Ressourcen hat, sollten Systeme diese Ressourcen mit wachsender Entfernung seltener nutzen. Bei der Zuweisung bevorzugen die typischen Lösungen daher daher lokale gegenüber entfernten Ressourcen, minimieren gemeinsam benutzte Ressourcen (z.B. durch Replikation) und plazieren häufig kommunizierende Applikationen auf benachbarte Prozessoren.

Parallelität in komponentisierten Systemen auszudrücken erfordert eine platzsparende Datenrepräsentation; die typischerweise verwendete Prozessor-Bitmaske skaliert nicht, da die Größe von der Anzahl der Prozessoren im System abhängt. Wenn man jedoch die sich natürlich ergebende Verwendung von Ressourcen, die durch die Struktur des Speichersubsystems bedingt ist, betrachtet, kann man den Freiheitsgrad möglicher Prozessorkombination für eine kompaktere Repräsentation einschränken. Die rekursive Zusammenfassung benachbarter Prozessoren in Cluster erlaubt es, auf der einen Seite eine große Zahl von Prozessoren mit wenig Speicherplatz zu repräsentieren, aber ebenso für kleine Cluster eine detaillierte Darstellung einer Untermenge weniger benachbarter Prozessoren zu erreichen. Beim Systemstart werden die Prozessornummern so vergeben, daß physikalisch benachbarte Prozessoren auch im numerischen Raum benachbart angeordnet sind. In der Datenrepräsentation wird neben der Prozessorbitmaske noch die Clustergröße als Zweierpotenz und ein Offset gespeichert. In einer Clustermaske von 32 Bit können somit von 16 individuellen bis zu 2^{256} Prozessoren dargestellt werden.

2.2 Adaptive Synchronisation

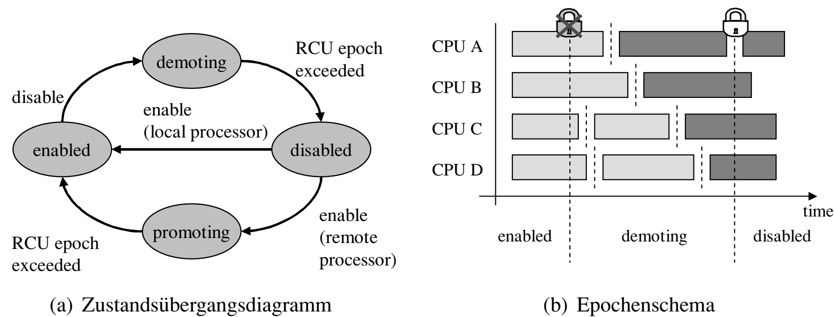
Die Erweiterung der Kernschnittstelle für Ressourcenverwaltung mit dem zusätzlichen Attribut der Prozessorkonkurrenz mit Hilfe einer Clustermaske erlaubt es, die Kernsynchronisationsprimitive den jeweiligen Applikationserforderungen entsprechend anzupassen. Dies betrifft sowohl die Sperrenprimitive als auch die Sperrengranularität. Für den Fall keiner Konkurrenz können Sperren sogar vollständig eliminiert werden.

Synchronisation im Kern wird zur Zusicherung von Konsistenz und zur Bestimmung der Ablaufreihenfolge verwendet. Die zwei primären Kernsynchronisationsschemen für Mehrprozessorsysteme mit gemeinsamem Speicher sind speicherbasierte Sperren und nachrichtenbasierte Synchronisation. Bei letzterem Verfahren wird Konsistenz durch Serialisierung der Ausführung auf einem einzelnen Prozessor erreicht. Ein adaptiver Synchronisationsmechanismus muß zur Laufzeit zwischen unterschiedlichen Verfahren umschalten können.

Dies kann durch Erweiterung der Sperren mit einer Aktivierungszustandsvariablen erreicht werden. Für Sperren im inaktiven Zustand wird dann die teure Sperrfunktion nicht ausgeführt. Bei der Transition vom aktiven in den inaktiven Zustand gibt es ein Zeitfenster, in dem die einzelnen Prozessoren unterschiedliche Sperrenzustände wahrnehmen. Dieses Zeitfenster ist nicht beschränkt, da die Speicherzugriffsdauer nicht deterministisch ist.

Das Synchronisationsverfahren *Read-Copy Update* löst ein ähnliches Problem mit Hilfe einer Markierung, die zwischen allen Prozessoren in einem sicheren Systemzustand zirkuliert wird. Nach einem vollen Umlauf der Markierung (bezeichnet als Epoche) ist garantiert, daß jeder Prozessor in einem sicheren Zustand war und demzufolge nicht versucht eine Sperre zu setzen. Am Ende des Umlaufs sind demzufolge alle Prozessoren synchronisiert und der Sperrzustand kann sicher umgesetzt werden.

Die Sperre enthält neben der eigentlichen Sperrvariablen demzufolge zwei zusätzliche Variablen, den Sperrenzustand und die Epoche. Der Kodepfad testet zuerst, ob die Sperre in einer Umschaltphase und die Epoche bereits abgelaufen sind; in diesem Fall wird die



(a) Zustandsübergangsdiagramm (b) Epochenschema

Abbildung 1: Zustandsübergangsdiagramm und Epochenschema für dynamische Sperren.

Umschaltung vollendet. Zusätzlich zu den zwei Zuständen *enabled* und *disabled* drücken die Zustände *promoting* und *demoting* die jeweiligen Zustandsübergänge aus. Wenn die Sperre im deaktivierten Zustand ist, kann die eigentliche Sperroperation übersprungen werden und damit die hohen Kosten für die atomare Speicheroperation und die Folgekosten durch Verunreinigung des Nah-Schnell-Speichers vermieden werden. Der folgende Kodeabschnitt verdeutlicht eine mögliche Implementierung dynamischer Sperren:

```
1  if (Lock.State != disabled) {
2    while TestAndSet(Lock.Spinlock) { /* wait */ }
3    if ( Lock.State != enabled && Lock.Epoch+1 < GlobalEpoch )
4      { /* perform lock state manipulation... */ }
5  }
6  /* critical section */
7  Lock.Spinlock = 0; /* lock release */
```

Basierend auf dem dynamischen Sperrprimitiv kann ebenso die Sperrgranularität angepaßt werden. Die jeweilige Entscheidung für grob- bzw. fein-granulare Sperren, d.h. wo und wie sperren, hängt von den Details der jeweiligen Kernalgorithmen ab. Feingranluare Sperren setzen eine teilbare Datenstruktur voraus, in der Teile unabhängig voneinander gesperrt werden können. Dies ist zum Beispiel der Fall für Hasch-Tabellen, in welchen die individuellen Einträge unabhängig voneinander sind. Die Granularität der Sperren kann mit Hilfe einer Kaskade von Sperren angepaßt werden. Um ein Teilobjekt zu sperren, müssen dabei immer alle Sperren akquiriert werden, jedoch können einige der (dynamischen) Sperren inaktiv sein. Blockierungen können durch Einhaltung einer Sperrenreihenfolge ausgeschlossen werden und die Sperren der Kaskade müssen in umgekehrter Reihenfolge der Akquisition wieder freigegeben werden.

Die Umschaltung der Sperrengranularität für ein teilbares Kernobjekt erfolgt in mehreren Schritten. Um Korrektheit zu garantieren, dürfen zu keinem Zeitpunkt zwei Prozessoren eine unterschiedliche Sperrgranularität nutzen. Eine sichere Umschaltung erfordert einen Zwischenschritt in dem beide Sperrengranularitäten—grob- und fein-granular—aktiviert sind. Nach Ablauf der Transitionsepoche haben alle Prozessoren die gleiche Sicht auf den Sperrenzustand, und die Sperren der vorherigen Granularität können deaktiviert werden. Es ist anzumerken, daß der zusätzlich erforderliche Speicher für eine zweistufige Sperrenkaskade identisch zu dem von dynamischen Sperren ist, d.h. zwei Bits für den Zustand

und der Speicher für den Epochenzähler. Da dynamische Sperren nur dann auf die Sperrvariable zugreifen, wenn die Sperre aktiviert ist, kann der notwendige Speicher dynamisch zugewiesen werden.

2.3 TLB Koheränz

Das Zusichern von TLB-Koheränz unterliegt ähnlichen Abwägungen wie grob- gegenüber fein-granularer Synchronisation. Nach der Aktualisierung von Seitentabelleneinträgen, die gemeinsam von mehreren Prozessoren benutzt werden, müssen die TLBs aller betroffenen Prozessoren aktualisiert werden. Die Aktualisierung wird durch Invalidierung der entsprechenden Einträge, typischerweise mit Hilfe einer Inter-Prozessor-Unterbrechung, erreicht. Die Kosten entfernter TLB-Invalidierungen sind dabei sehr hoch.

Für Funktionen, die von der vollständigen Durchführung von Seitenrechtenveränderungen abhängen, hat die Latenz für die entfernten TLB Aktualisierungen direkte Auswirkung auf die Gesamtlaufzeit. Typische Beispiele sind Rechteeinschränkungen für Kopieren-beim-Schreiben (z.B. für die UNIX Funktion „Gabeln“) und für den Seitenauslagerungsmechanismus. Durch Zusammenfassung mehrerer TLB-Invalidierungen können die Kosten für die Funktion drastisch reduziert werden, jedoch wird dabei die Laufzeit, abhängig von der Anzahl der Einträge, erhöht. Durch Zusammenfassung werden außerdem Abhängigkeiten zwischen Seiten erzeugt die, wie anfangs gefordert wurde, unabhängig und auf Seitengranularität sein sollen.

Um Korrektheit zu erreichen, müssen alle Rechteveränderungen zwischen Prozessoren synchronisiert werden. Mit Verzögerung der TLB-Invalidierung kann es passieren, daß andere Prozessoren bei Inspektion der Seitentabellen inkorrekter Weise annehmen, daß die Rechte bereits gesetzt und gültig sind. Solch Abhängigkeit erzeugt ein Performanzproblem, da billige und unabhängige Operationen nun deutlich länger Laufzeitcharakteristiken aufweisen und zusätzlich die Latenz für die entfernten TLB-Invalidierung enthalten.

Bei der optimierten TLB-Invalidierung werden dynamisch einzelne Speicherobjekte für die Laufzeit der Operation zu größeren zusammengefaßt. Durch Trennung von Rechteveränderung und TLB- Invalidierung kann diese Abhängigkeit wieder aufgebrochen werden.

Die Rechteveränderung erfordert *genau dann* eine entfernte TLB-Invalidierung, wenn ein anderer Prozessor möglicherweise einen alten (d.h. inkorrekten) Eintrag zwischengespeichert hat. Desweiteren ist eine Aktualisierung nötig, wenn ein anderer Prozessor bereits die Seitentabelle verändert, aber die TLB-Invalidierung noch nicht beendet hat. Die Rechte sind erst dann autoritativ nachdem alle Prozessoren, die Zugriff auf die Seitentabelle haben, ihre TLBs invalidiert haben.

Zur Verwaltung der TLB-Zustände wird pro Prozessor eine Aktualisierungsepoche eingeführt, wobei eine neue Epoche jeweils mit der Invalidierung der TLB Einträge der vorherigen Epoche beginnt. Nach Aktualisierung einer Seitentabelle während der Epoche n für einen speziellen Prozessor ist in der Epoche $n+1$ garantiert, daß alle Einträge der Seitentabelle autoritativ sind und der Prozessor keine veralteten mehr Einträge vorhält. Für

Seitentabellen, die von mehreren Prozessoren verwendet werden, ist dies nach Erhöhung aller TLB Epochen der Fall. Bei Aktualisierung der Zugriffsrechte für ein Speicherobjekt werden alle TLB-Epochen der Prozessoren mit Zugriffsrechten auf das Objekt, und damit potentiell aktiven TLB Einträgen, vermerkt. Basierend auf dieser Information kann ermittelt werden, ob weitere TLB-Invalidierungen nötig sind. Weitere Kostenreduzierungen können durch die Einführung einer Super-Epoche und genaue Buchführung ausstehender Operationen erreicht werden [Uhl05].

3 Anwendung und Leistungsanalyse für den L4 Mikrokern

In einer Leistungsanalyse wurden die Kosten für individuelle Mikrokernoperationen für den L4-Mikrokern mit Hilfe von Mikrobenchmarks analysiert. L4 stellt den Stand der Technik der weltweiten Mikrokernforschung dar und wird sowohl in einer Reihe von Forschungs- als auch Industrieprojekten eingesetzt.

Es wurden unterschiedliche Einzel- und Mehrprozessorlasten untersucht und die Kosten für Intra- und Inter-Prozessor-Kommunikation als auch für geringe gegen hohe Parallelität verglichen. Die Skalierbarkeit ist dabei nach Unraus Designprinzipien evaluiert. Außerdem wurden die zusätzlichen Kosten der Multiprozessorprimitive zur Basisperformance des Einprozessorsystems ermittelt.

Alle Messungen wurden auf einem *IBM xSeries 445 Server* mit acht 2.2 GHz *Pentium 4 Intel Xeon Prozessoren* durchgeführt. Die Prozessoren hatten jeweils zwei logische Prozessoren (HyperThreads). Das System bestand aus zwei Prozessorplatinen mit jeweils vier Prozessoren; die Platinen hatten separaten Speicher und waren mit IBMs nicht-uniformen Speicherbus verbunden.

Das relative strikte Ordnungsmodel für Speicherzugriffe von Intel-Prozessoren hat negative Performanz-Implikationen auf spekulative Ausführung und hat hohe Kosten für atomare Speicherinstruktionen. Die Kosten der atomaren Austauschinstruktionen (xchg) sind 125 Prozessortakte und die der atomaren Vergleichs-und-Austauschfunktion (lock cmpxchg) sind 136 Takte.

3.1 Interprozeßkommunikation

Interprozeßkommunikation (IPK) zwischen Programmfäden ist die zentrale Funktion eines jeden Mikrokerns. IPK stellt den Mechanismus zum sicheren und kontrollierten Wechseln zwischen Adreßräumen dar, ist der Delegationsmechanismus für Ressourcenrechte, der universelle Datenaustauschmechanismus zwischen nicht-vertrauenswürdigen Programmen und das Synchronisationsprimitiv für unabhängige Programmfäden. Liedtke hat gezeigt [Lie95], daß die direkten und indirekten Kosten (z.B. durch Verschmutzung des Nah-Schnell-Speichers) für einen IPK minimal sein müssen, da die Gesamtleistung eines mikrokernbasierten Systems direkt von diesem Primitiv abhängt. Der IPK für Mehrprozessorsysteme hat zwei Hauptanwendungsfälle, zur Realisierung von Klienten-Server-

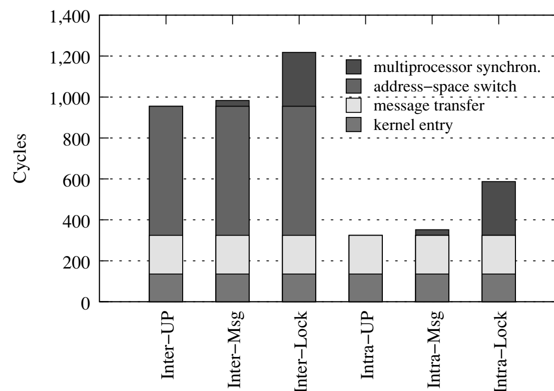


Abbildung 2: Aufschlüsselung der Kosten des IPK Primitives für Inter- und Intra-Adressraumkommunikation für Einprozessor (UP) und Mehrprozessor Systeme mit nachrichtenbasierter (msg) und sperrenbasierter (lock) Synchronisation. Kommunikation erfolgt auf gleichem Prozessor.

Systemen und zur Synchronisation und Signalisierung.

Klienten-Server-Kommunikation ist die kostenkritischste Operation, da in mikrokernbasierten Systemen alle Kernaufrufe eines monolithischen Systems durch zwei IPKs ersetzt werden. In den meisten Fällen blockiert der aufrufende Klient bis der Server den entfernten Funktionsaufruf beendet und das Ergebnis (in einer weiteren IPK) an den Klienten zurückliefert hat. In Mehrprozessorsystemen könnten theoretisch Aufrufe auf andere Prozessoren verteilt werden, was jedoch aufgrund der hohen Kosten für die Migration des Nah-Schnell-Speicher Inhalts und die Signalisierungsverzögerung untauglich ist.

Eine Evaluierung der Kosten für IPK auf dem gleichen Prozessor für unterschiedliche Synchronisationsprimitive hat gezeigt, daß das dynamische Sperrenschema die Synchronisationskosten fast vollständig eliminieren kann. Die architekturspezifischen Kosten für die TLB und L1-Nah-Schnell-Speicher Invalidierung sind dabei überaus dominant. Im Vergleich zum Einprozessorfall erzeugt die nachrichtenbasierte Synchronisation sehr geringe Zusatzkosten von 2,9% während die speicherbasierte Synchronisation 27,6% Zusatzkosten erzeugt. Für den Adreßraumlokalen Fall (also ohne Umschaltung des TLBs) sind die Zusatzkosten für genannte Fälle 8,6% gegenüber 81,2%. In Abbildung 2 ist eine Einzelaufstellung der Kosten gegeben.

Das Inter-Prozessor-IPK-Primitiv wurde für das typische Szenario der Schrankensynchronisation paralleler Programmiersprachen mit mehreren Arbeitsfäden, die von einem Koordinatorfaden eine Nachricht zugestellt bekommen, evaluiert. Das Szenario ist mit dem von Bull [BO01] vorgestellten Mikrobenchmark für die OpenMP PARALLEL Direktive zu vergleichen. Es wurde die Benachrichtigungslatenz der Arbeitsfäden für eine wachsende Anzahl Prozessoren gemessen. In Abbildung 3 werden die Kosten für nachrichtenbasierte denen der sperrenbasierten Synchronisation gegenübergestellt. Die IPK mit speicherbasierte Synchronisation hat einen klaren Geschwindigkeitsvorteil gegenüber der nachrichtenbasierten Variante mit einem Faktor von 2,5 für zwei Prozessoren bis zu 3,6 für 16

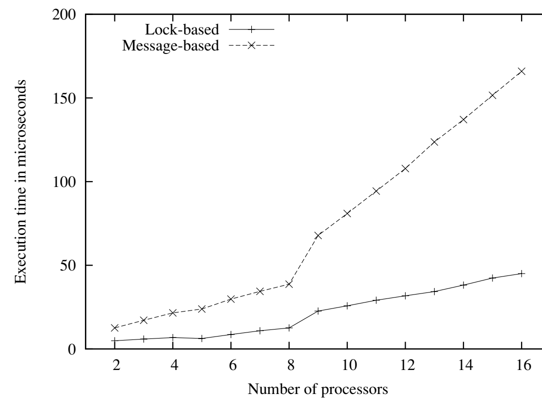


Abbildung 3: Zustellungslatenz für das Gablungs-Vereinigungs-Programmiermodell von OpenMP. Da die Latenz direkt proportional zur Gesamtausführungszeit ist, hat die speicherbasierte Synchronisation einen deutlichen Geschwindigkeitsvorteil.

Prozessoren.

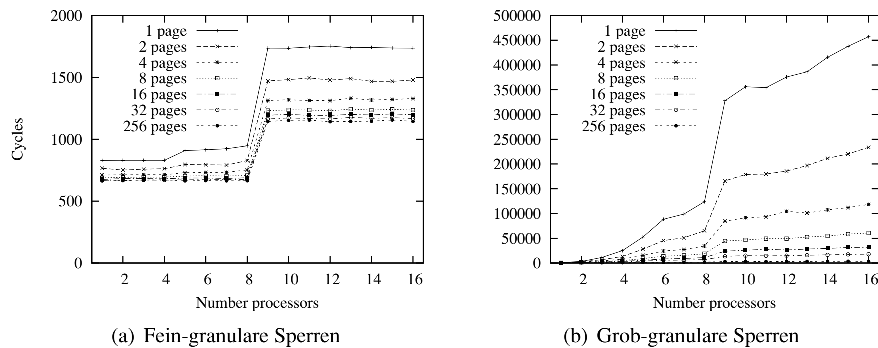
Die Messungen zeigen, daß die jeweilig optimale Synchronisationsmethode vom Anwendungsfall abhängt; sowohl speicher- als auch nachrichtenbasierte Synchronisation zeigen einen deutlichen Geschwindigkeitsvorteil für den jeweiligen Anwendungsfall. Nur mit Hilfe der dynamischen Anpassung kann das jeweilige Optimum erreicht werden.

3.2 Speicherverwaltung

Die Speicherressourcenverwaltung in L4 basiert auf der rekursiven Konstruktion von Adreßräumen und wird durch das Einblenden bereits verfügbaren Speichers in einen anderen Adreßraum realisiert. Ein Programmfaden kann mit Hilfe des IPK eine virtuelle Seite in seinem Adreßraum spezifizieren und die Rechte an einen anderen Programmfaden weiterleiten. Der Kern speichert die jeweilige Assoziation und die Rechte innerhalb einer Datenbank. Etablierte Seitenzugriffsrechte können zu jedem Zeitpunkt asynchron widerrufen werden.

Die rekursive Konstruktion der Adreßräume erzeugt einen gerichteten Abhängigkeitsgraphen in dem alle Adreßräume mit einem Uradreßraum verbunden sind. Da Rechte zu jedem Zeitpunkt in jeder Stufe der Hierarchie entzogen werden, kann es bei zu grob-granularer Synchronisation zu Verstopfungen kommen, wohingegen sehr hohe Laufzeitkosten und Verschmutzungseffekte im Nah-Schnell-Speicher bei fein-granularer Synchronisation auftreten.

Durch explizite Kontrolle über die Strukturierung der Datenrepräsentation in der Datenbank können beide Synchronisationsgranularitäten unterstützt werden. Zum Zeitpunkt der Rechteweitergabe können die Kommunikationspartner die Kernrepräsentation soweit beeinflussen, daß Operationen unabhängig voneinander und gleichzeitig ausgeführt werden



(a) Fein-granulare Sperren

(b) Grob-granulare Sperren

Abbildung 4: Paralleler Rechteentzug für fein- und grob-granular synchronisierte Speicherobjekte

können. Zur Wahrung der Unabhängigkeit gleichzeitig ausgeführter Veränderungen wird das TLB-Koheränzschema aus Abschnitt 2.3 verwendet.

Die Kosten und die Skalierbarkeit wurden in einem Benchmark ermittelt, der wiederholt die Schreibrechte auf Seiten entzieht. Diese Operation ist typisch für Kopieren-bei-Schreiben Szenarien, wie zum Beispiel für die Funktion „Gabeln", und ist identisch zum Entziehen von Speicher jedoch ohne die letztendliche Speicherfreigabe. Da die Kernrepräsentation unverändert bleibt, ist der Benchmark wiederholungsfähig. Die Messungen wurden 200 mal ausgeführt und es wurde anschließend darüber gemittelt.

Die Laufzeitkosten für fein-granulare Sperren sind pro Sperre 185 Takte höher als für grob-granulare Sperren. Im Benchmark wurden 24% höhere Laufzeitkosten für einen moderat-besetzten Adreßraum mit 128 Seiten und 32% für höhere Kosten für einen mit 256 Seiten besetzten Adreßraum gemessen. Dies muß im Vergleich zu typischen Linux gesehen werden: Bash nutzt 185 Seiten, Emacs 1513 Seiten und MySQL 4453 Seiten. Zusätzlich zu den höheren Laufzeitkosten führen individuelle Sperren noch zur Verschmutzung des Nah-Schnell-Speichers.

Im nächsten Benchmark wurde die Skalierbarkeit eines parallelen Rechteentzugs für eine wachsende Anzahl Prozessoren gemessen. In dem Benchmark wurde erst eine Menge von Seiten mehreren Fäden auf unterschiedlichen Prozessoren im Adreßraum eingeblendet. Danach entzieht ein Faden die Schreibrechte für eine zunehmende Anzahl von Seiten während alle anderen Fäden wiederholt die Rechte auf eine Seite entziehen. Im Benchmark wird die durchschnittliche Ausführungszeit für grob- und fein-granular Sperren ermittelt.

Die Ausführungszeit bei Verwendung fein-granularer Sperren ist fast konstant unabhängig von der Anzahl der Prozessoren wohingegen die Kosten für grob-granulare Sperren drastisch mit der Anzahl Prozessoren steigen (siehe Abbildung 3.2). Die beiden Anomalien im Graph sind auf das NUMA Speichersubsystem (Prozessor 5) und HyperThreading (Prozessor 9) zurückzuführen.

Ein weiterer Benchmark hat optimale Skalierbarkeit für Rechteentzug mit unabhängigen Sperren gezeigt, während für den Fall grob-granularer Sperren die Laufzeit linear mit der Anzahl der Prozessoren wuchs (siehe [Uhl05]).

4 Zusammenfassung

In dieser Arbeit wurden Mechanismen zur dynamischen Anpassung von Kernsynchronisationsprimitiven und zur TLB-Koheränz in Multiprozessorsystemen diskutiert. Die strikte Separierung von Rechteverwaltung und der eigentlichen Systemfunktionalität in mikrokernbasierten Systemen verhindert den Zugriff auf semantische Informationen, die für effiziente Wahl der Synchronisationsprimitive Vorraussetzung sind. Mit Hilfe einer effizienten Repräsentation der möglichen Parallelität, sicheren Mechanismen zur Adaption des Kernsynchronisationsverfahrens und der vollständigen Deaktivierung der Kernsperren, konnten die Kosten für die kritischen Kodepfade minimal gehalten werden. Dies wurde mit Hilfe eines sowohl in der Forschung als auch der Industrie weitverbreiteten Mikrokerns nachgewiesen.

Literatur

[BO01] J. Mark Bull und Darragh O'Neill. A Microbenchmark Suite for OpenMP 2.0. In *3rd European Workshop on OpenMP*, September 2001.

[GVW89] J. R. Goodman, M. K. Vernon und P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, April 1989.

[HJ86] C.-T. Ho und L. Johnsson. Distributed Routing Algorithm for Broadcasting and Personalized Communication in Hypercubes. In *International Conference on Parallel Processing (ICPP 1986)*, Seiten 640–648, 1986.

[Lei85] C. E. Leierson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, c-34:892–901, Oktober 1985.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proc. of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, Dezember 1995.

[LLG$^+$92] Daniel Lenoski, James Laudon, Gharachorloo Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz und Monica S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3), Marz 1992.

[MCS91] John M. Mellor-Crummey und Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Februar 1991.

[Uhl05] Volkmar Uhlig. *Scalability of Microkernel-Based Systems*. Dissertation, University of Karlsruhe, Germany, Mai 2005.

[Unr93] Ronald Unrau. *Scalable Memory Management through Hierarchical Symmetric Multiprocessing*. Ph.D. thesis, University of Toronto, Toronto, Ontario, Januar 1993.

Dr. Volkmar Uhlig erhielt sein Diplom in Informatik von der Technischen Universität Dresden verliehen. Er arbeitete beim IBM T.J. Watson Research Center in New York am SawMill Linux Projekt, einem mikrokernbasierten Betriebssystem. Von 2001 bis 2005 promovierte Herr Uhlig am Lehrstuhl für Systemarchitektur der Universität Karlsruhe. Sein Forschungsschwerpunkt war Skalierbarkeit des L4Ka Mikrokerns, welcher weltweit in Forschung und Industrie eingesetzt wird. In 2003 verweilte er für sechs Monate am Intel Microprocessor Research Lab (MRL) in Oregon und arbeitete im Bereich Prozessorvirtualisierung. Seit Mai 2005 ist Herr Uhlig dauerhaft als Forscher bei IBM Watson.