Synthetix

Code-Generierung mit Quasi-Invarianten

Hauptseminar WS 97/98
Aktuelle Trends in der Betriebssystem-Forschung

Gerhard Müller

17.12.1997

Inhaltsverzeichnis: Seite

1. Motivation

Heute kann ein Rechner aus einer Vielzahl von Hardwarekomponenten bestehen, wobei in einer Sparte (z.B. Grafikkarten) wieder viele unterschiedliche Modelle existieren können. Um ein solches System zu kontrollieren, ist es für Betriebssysteme nötig, viele Systemzustände und Unterschiede in Hard- und Software zu verwalten. Dies führte zu immer größeren Betriebssystemen. Durch die Notwendigkeit alle möglichen Systemzustände vor einem Systemaufruf zu erkennen und darauf aufbauend weiter zu arbeiten, kommt es häufig vor, daß bei einfachen Aufträgen an das System sehr viel mehr Zeit für eine Zustandserkennung verbraucht wird, als zur eigentlichen Abarbeitung nötig ist. Nun wäre es wünschenswert, so wenig Verlust wie möglich in Kauf nehmen zu müssen.

Im Synthesis-Projekt, dem „Vorgänger“ von Synthetix, wurde dies durch die Bereitstellung einer Anzahl von verschiedenen spezialisierten Diensten des Betriebssystems erreicht. Jeder spezialisierte Dienst baut auf Invarianten („feste Systemzustände“) auf, die für eine bestimmte Anwendung typisch sind und somit nicht bei jedem Aufruf erneut bearbeitet werden müssen.

Mit dem Synthetix-Projekt wurde Synthesis erweitert. Hierbei werden nicht nur „feste Zustände“, wie bei Synthesis, betrachtet, sondern auch solche Zustände, welche über einer relativ großen Zeitspanne hinweg konstant bleiben.

2. Was ist Spezialisierung (Specialization)

Spezialisierung kann man als partielle Berechnung eines Programms betrachten. Diese Berechnung stützt sich auf den Input des Programms, wie zum Beispiel konstante Eingabeparameter oder Umgebungsvariablen. Solche Invarianten lassen sich in jedem Programm finden, dessen Kontrollfluß durch die Auswertung der aktuellen Umgebung und des Systemzustands bestimmt ist.

Besonders die Betriebssysteme bieten eine Vielzahl von Invarianten. Spezialisierung bedeutet also, daß man allgemeinen Code dadurch vereinfacht, daß man schon feststehende Invarianten in den Code integriert. Dies führt dazu, daß sich die aufgewendete Zeit für die Analyse des aktuellen Zustands verringert. Begründung: Im spezialisierten Code wurden ja schon einige Invarianten des aktuellen Zustands fest definiert, welche nun nicht mehr ausgewertet werden müssen.

Die Invarianten lassen sich in zwei Kategorien einteilen, und zwar abhängig davon, ob sie vor oder während der Laufzeit verwendbar werden. Invarianten, welche vor der Laufzeit bekannt sind, können während der Compilierung durch herkömmliche Verfahren auf Source-Level verarbeitet werden. (Zum Beispiel: Ob eine FPU vorhanden ist; Wie groß der Prozessor-Cache ist; ...)

Für Betriebssysteme ist es sicher von Vorteil beide mögliche Arten von Invarianten zur Spezialisierung zu nutzen. Synthetix beschäftigt sich nun mit den Invarianten, welche während der Laufzeit erst handhabbar werden und versucht zur Laufzeit Spezialisierungsmechanismen zur Effizienzsteigerung einzusetzen.

Im weiteren sind für die Betrachtung auch solche Invarianten von Bedeutung, welche Ihren Zustand über einen längeren Zeitraum beibehalten.

Dies ergibt die folgende Unterscheidung:

* True Invariant, ist ein Zustand im System, welcher sich über die gesamte Zeit nicht verändert. (Im weiteren einfach Invariante benannt)
* Quasi-Invariant, ist ein Zustand im System, welcher über eine lange Zeit konstant bleibt, sich jedoch in der Zukunft ändern kann. (Im weiteren Quasi-Invariante benannt)

Während der Laufzeit kann es vorkommen, daß sich zusätzlich „neue“ Quasi-Invarianten festlegen lassen. Somit ist es sicherlich wünschenswert, einen schon spezialisierten Code während der Laufzeit an jedem Punkt, an dem mehr Information verfügbar wird, weiter spezialisieren zu können. Eine solche wiederholte Anwendung der partiellen Berechnung nennt man Incremental Specialization.

Falls Quasi-Invarianten in einem System ungültig werden, muß die korrekte spezialisierte oder sogar unspezialisierte Funktion für die weiteren Aktionen bereitgestellt werden. Eine Ersetzung der vorhandenen Funktion durch eine dem Systemzustand angepaßte Funktion nennt man Replugging.

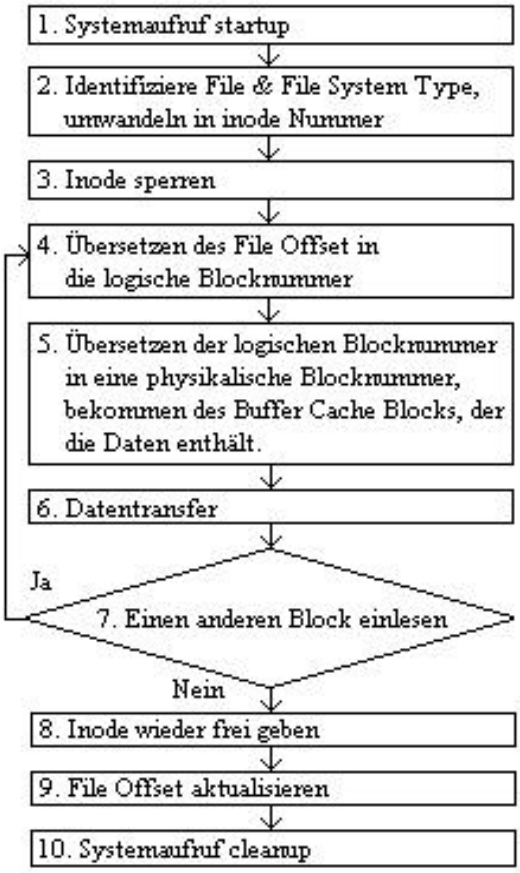
Falls Funktionen schon vor Ihrem eigentlichen Aufruf spezialisiert vorgehalten werden, dann kann es vorkommen, daß diese vor ihrem ersten Aufruf ersetzt werden müssen, da eine oder mehrere Quasi-Invarianten ungültig geworden sind. Wenn nun spezialisierte Funktionen schon vor dem Aufruf bereit gestellt werden, so redet man auch von einer Optimistic Specialization.

Im Moment wurde immer davon ausgegangen, den Code zur Laufzeit zu spezialisieren. Wenn man die Optimistic Specialization ausnutzt, kann man auch zu Code Templates greifen, die man schon kompiliert vorhalten kann. So können die Kosten der Code Generierung zur Laufzeit vermieden werden, was eine zusätzliche Entlastung bringt.

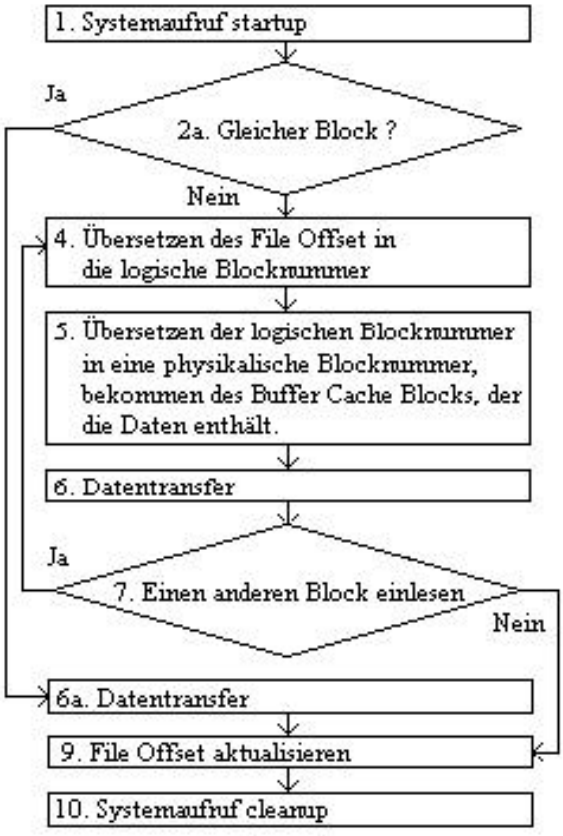
3. Als Beispiel ein spezialisiertes HP-UX read

read wurde ausgewählt, da es repräsentativ für viele Unix-Systembefehle ist. Außerdem ist der read-Code ein wohlverstandener und hoch optimierter Teil heutiger Betriebssysteme. Um sicherzustellen, daß die Performance-Resultate auf aktuelle Software anwendbar ist, wurde darauf geachtet daß die spezialisierte read Implementation das gegenwärtige Interface und Verhalten von HP-UX read einhält.

3.1. Das HP-UX read



HP-UX read



is_read

Um die Spezialisierung besser zu verstehen, betrachten wir zuvor das konventionelle HP-UX read.

1. **Systemaufruf startup**: Privilegierter Modus; Wechsel zum Kernel-Stack; User-Structure aktualisieren.
2. **Identifiziere File & File System Type**: Übersetzen der File Descriptor Nummer in einen Filedeskriptor. Übersetzen des Filedeskriptor in einen Zeiger auf das Open-File Objekt. Kontrolle, ob das File im richtigen Modus geöffnet wurde. vnode aus dem Open-File Objekt holen und in eine inode umrechnen.
3. **Inode sperren**, damit read/write Operationen die in verschiedenen Prozessen auf einem File ausgeführt werden nicht überlappen.
4. **Den Block identifizieren**: Übersetzen des File Offset in eine logische (File) Block Nummer, dann Übersetzen der logischen Block Nummer in eine physikalische (Disk) Block Nummer.
5. **Die virtuelle Adresse der Daten finden**: Den gewünschten physikalischen Block in dem Block-Buffer Cache finden und Berechnen der virtuellen Adresse der Daten anhand des File Offset. (In neueren UNIX-Systemen wird kein Block-Buffer Cache mehr verwendet, sondern ein Paging System.)

6. **Datentransfer = copyout**: Kopieren der benötigten Bytes von dem Buffer-Cache Block zu dem Buffer des Benutzers.
7. **Einen anderen Block einlesen**: Vergleiche die Anzahl der kopierten Bytes mit der Anzahl der angeforderten Bytes. Falls noch mehr Bytes benötigt werden gehe zurück zu 4.
8. **Den Inode frei geben**.
9. **File Offset aktualisieren**: Sperren des File Table, File Offset aktualisieren, File Table wieder frei geben.
10. **Systemaufruf cleanup**: Aktualisieren der Kernel Seiteninformation und wechsle zurück zum User Stack. Privilegierten Modus verlassen.

Die hier genannten Aktionen können in folgende Kategorien eingeteilt werden: Interpretation, Traversal, Locking oder Work. *Interpretation*: Überprüfung der Parameterwerte und des Systemzustands um darauf den Kontrollfluß aufzubauen. *Traversal*: Kann als Vorbereitung zur Interpretation angesehen werden; aufdecken des Systemzustandes, suchen von Datenstrukturen. *Locking*: Enthält alle Aktivitäten welche zur Synchronisation notwendig sind. *Work*: ist der eigentliche Zweck des Aufrufs. In diesem read lassen sich die Aktivitäten der Stufen 1,2,4,5,7 und 10 überwiegend als Interpretation und Traversal einordnen, die Stufen 3,8 und 9 überwiegend in Locking und nur die Stufe 6 und ein kleiner Teil der Stufe 9 können in die Kategorie Work gelegt werden. Ideal wäre natürlich eine Routine, welche ausschließlich oder überwiegend aus der Kategorie Work besteht.

3.2. Das spezialisierte is_read

Im Folgendem werden verschiedene Invarianten und Quasi-Invarianten behandelt, die in einer Leseoperation vorkommen. Eine Übersicht wird in der Tabelle am Ende dieses Abschnitts gegeben.

Die Invariante FS_CONSTANT kann aufgrund der File System Semantik von UNIX als konstant angesehen werden. Diese Invariante steht für folgenden, immer gültigen, Zustandswert: File Type, File System Type und Block Size. Hierdurch lassen sich die Kosten für Traversal in der Stufe 2 umgehen.

Durch die Quasi-Invariante SEQUENTIAL_ACCESS wird es möglich, daß man nach dem erstmaligen Aufruf der read-Routine mit dem gleichen Prozeß auf dem gleichen File die Stufen 2,3,4,5,7,8 und 9 umgehen kann. Dies liegt daran, daß man bei dem ersten Aufruf schon die virtuelle Adresse der zu kopierenden Daten erhalten hat. Diese kann nun bei jedem weiteren Aufruf direkt angesprochen werden. -> Spezialisierung der read-Routine direkt nach dem ersten Aufruf. Diese Spezialisation betrifft noch weitere Quasi-Invarianten. Und zwar darf read die Puffergrenzen nicht überschreiten und es darf nicht vorkommen, daß die Pufferersetzungsroutine die Daten verändert, welche durch die virtuelle Adresse bezeichnet werden.

Die Quasi-Invariante NO_HOLES ist ebenso für die oben ausgeführte Spezialisierung notwendig, da kontinuierliches sequentielles Lesen nur zu einer kontinuierlichen Bytekopie spezialisiert werden kann, wenn keine Lücken enthalten sind. Diese Lücken würden sonst eine Interpretation des Empty Block Pointers benötigen.

Die Quasi-Invarianten NO_INODE_SHARE und NO_FP_SHARE erlauben einen exklusiven Zugriff auf das File. Hierdurch können die Locking - Aktivitäten für inode und File Table in den Stufen 3,8 und 9 umgangen werden. Dies erlaubt zudem, daß man Informationen wie den File Pointer cachen darf. Dies bewirkt, daß man alle Interpretation-Traversal- und Locking-Aktivitäten in den Stufen 2,3,4,5,8 und 9 umgehen kann.

Ergebnis: Die Stufen 2,3 und 8 wurden eliminiert. Die Stufen 4 und 5 wurden für kleinere sequentielle Leseoperationen eliminiert. Die Restlichen Stufen wurden spezialisiert. Is_read wurde in Bezug auf in der folgenden Tabelle genannten Invariante (FS_CONSTANT) und Quasi-Invarianten spezialisiert.

Diese Implementation von is_read ist wie folgt spezialisiert: Für Files auf dem lokalen File System mit einer Blockgröße von 8 KB. Is_read wird zu dem Zeitpunkt aktiviert, an dem open aufgerufen wird, allerdings nur für Leseoperationen auf diesem File und nur von dem aufrufenden Prozeß. Das lesen eines anderen File-Typs wird von der herkömmlichen read-Funktion abgearbeitet.

Diese spezialisierte Funktion wird nur ausgeführt, wenn alle Invarianten und Quasi-Invarianten erfüllt sind. Außer dem is_read soll keine weitere spezialisierte Funktion angeboten werden, so daß bei einem anstehenden Replugging is_read sofort durch das HP-UX read ersetzt wird.

(Quasi-)Invarianten	Beschreibung	Einsparungen
FS_CONSTANT	Invariante File System Paramter	Vermeidet Stufe 2
NO_FP_SHARE	Kein Sharing des File Pointers	Vermeidet einen Teil der Stufe 9 und erlaubt das Cachen des File Offset
NO_HOLES	Keine Löcher im File	Vermeidet das Überprüfen auf leere Block-zeiger in der Inode-Struktur
NO_INODE_SHARE	Kein Sharing der Inode	Vermeidet die Stufen 3 und 8
NO_USER_LOCKS	Keine User-Level Locks	Kein Überprüfen von User-Level Locks nötig
READ_ONLY	Keine Schreiber	Erlaubt eine optimierte Fileüberprüfung
SEQUENTIAL_ACCESS	Aufrufe von is_read bekommen den File Offset von dem vorangegangenen is_read	Für kleine Leseoperationen werden die Stufen 2,3,4,5,7,8,9 vermieden.

4. Probleme und Lösungen anhand des Beispiels mit Replugging

Das Replugging muß immer dann eingeleitet werden, wenn Quasi-Invarianten ungültig werden. Um eine solche Veränderung zu erkennen, werden an den kritischen Stellen im System sogenannte Guards eingefügt. Diese haben die Aufgabe zu kontrollieren, ob und welche Quasi-Invarianten der spezialisierten Funktion sich geändert haben und den entsprechenden Replugger zu aktivieren.

Die zwei wichtigsten Punkte bei der Betrachtung von Replugging sind, einerseits die Effizienz beim Aufruf von spezialisierten Funktionen und andererseits die Kontrolle, während eines Replugging, über Prozesse, welche sich innerhalb von spezialisierten Funktionen befinden, und über Prozesse, welche ebenso Replugging durchführen wollen.

Typische Probleme sind die folgenden Fälle:

I Interaktionen zwischen dem Guard und dem Prozeß, dessen Code replugged werden soll.

II Ein Guard löst das Replugging einer spezialisierten Funktion aus, während sich ein anderer Prozeß in dieser Funktion blockiert hat. Zum Beispiel beim warten auf I/O Daten.

III Mehrere Guards wollen zur gleichen Zeit dieselbe spezialisierte Funktion mittels Replugging ersetzen.

Ein System, welches Replugging verwendet, muß diese Probleme korrekt lösen ohne daß dabei die Leistung erheblich geschwächt wird. Zumindest darf die Leistung nicht mit der Leistung eines Systems ohne Spezialisierung mittels Quasi-Invarienaten gleich sein, denn sonst würde diese Spezialisierung nichts mehr bringen.

4.1. Guards für is_read

Wie schon erwähnt müssen Guards an verschiedenen Stellen im System eingebaut werden. Sie gewährleisten, daß bei Änderungen an Quasi-Invarianten, gültig und ungültig werden, der richtige, für die betroffene Funktion verantwortliche, Replugger ausgeführt wird. Bei is_read muß dies an folgenden Stellen passieren.

Quasi-Invariant	HP-UX Systemaufrufe, in denen Quasi-Invarianten ungültig werden können
NO_FP_SHARE	creat, dup, dup2, fork, sendmsg
NO_HOLES	open
NO_INODE_SHARE	creat, fork, open, truncate
NO_USER_LOCKS	lockf, fcntl
READ_ONLY	open
SEQUENTIAL_ACCESS	lseek, readv, is_read, und bei einer Buffer-Cache Block Ersetzung

Die beiden Quasi-Invarianten READ_ONLY und NO_HOLES werden zum Beispiel nur in open überwacht, da diese nur ungültig werden können, falls genau das File auch zum Schreiben geöffnet wird, für das die spezialisierte Funktion angeboten wurde.

Die übrigen Quasi-Invarianten müssen an den Stellen überwacht werden, an denen Operationen auf die FileStrukturen (wie zum Beispiel den FileDescriptor) zugreifen und diese eventuell verändern.

4.2. Der Replugging Algorithmus

Als erstes betrachten wir die Lösung der genannten Probleme I und III. Einen Sonderfall hat man, falls ein System mit einem single-threaded kernel vorliegt, in dem keine Systemaufrufe im Kern blockiert werden können, dann fallen diese Probleme weg.

Die weiteren Betrachtungen werden allerdings auf ein Multiprozessorsystem ausgerichtet, bei dem Systemaufrufe unterbrochen werden können.

Um den Replugging Algorithmus zu vereinfachen werden im Weiteren folgende Annahmen gemacht.

(A1) Systemaufrufe werden im Kern nicht abgebrochen, so daß nicht überprüft werden muß, ob is_read abgebrochen wurde.

(A2) Es gibt nur einen Thread pro Prozeß, so daß mehrere Systemaufrufe nicht konkurrierend auf Prozeß-Level-Datenstrukturen zugreifen.

(A3) Daß nur ein Prozess innerhalb des spezialisierten Codes ausgeführt wird.

Um Problem I zu vereinfachen wird ein sogenannter Holding Tank eingeführt, der die Prozesse aufnimmt, welche bei einem anstehendem Replugging die Funktion noch nicht ausführen können. Diese Prozesse werden nach erfolgtem Replugging wieder aktiviert und können dann die ersetzte Funktion ausführen. Im Weiteren wird nur der Fall betrachtet, daß es eine spezialisierte Funktion und eine allgemeine Funktion gibt.

Dies wird vom aufrufenden Prozeß ausgeführt:

(1.) Überprüfe den File Deskriptor, ob der Zugriff auf eine spezialisierte Funktion führt. Falls nicht, verzweige aus dem schnellen Pfad aus.

2. Setze das inside-Flag.

3. Verzweige indirekt. Diese Verzweigung führt entweder zu einem Holding Tank oder zu einer spezialisierten Funktion. Diese Verzweigung wird vom Replugger gesetzt.

Die Leseoperation eines Prozeß:

1. Führe die Funktion zum lesen aus.

2. Lösche das inside-Flag.

Prozeß im Holding Tank:

1. Lösche inside-Flag.

2. Schlafe, bis der Replugger mit seiner Arbeit fertig ist.

3. Springe wieder zur Anfangsroutine.

Replugging Algorithmus:

1. Fordere per-process lock an, um andere Replugger zu blockieren. Es könnte sein, daß einige Guards konkurrierend aufgerufen wurden, um für denselben File Deskriptor zu bearbeiten.
2. Fordere per-file lock an, um ein Austreten aus dem Holding Tank zu blockieren.
3. Verändere den per-file Indirekt Pointer um die Leser zu dem Holding Tank zu leiten. (Dies Verändert die Aktion des Leserprozeß in Stufe 3 so, daß keine neuen Prozesse den spezialisierten Code betreten können.)
4. Spinwait auf das inside-Flag, bis dieses frei ist. Nun führen keine Prozesse mehr die spezialisierte Funktion aus.
5. Unterstütze Incremental Specialization in Bezug auf die Invariante, welche ungültig geworden ist.

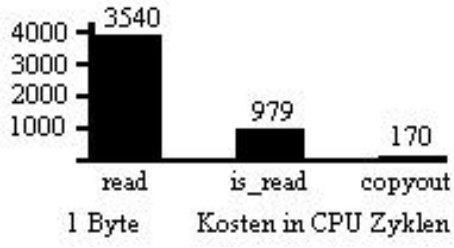
(6.) Setze den File Deskriptor richtig, eingeschlossen den Indikator, daß die Funktion nicht länger spezialisiert ist.

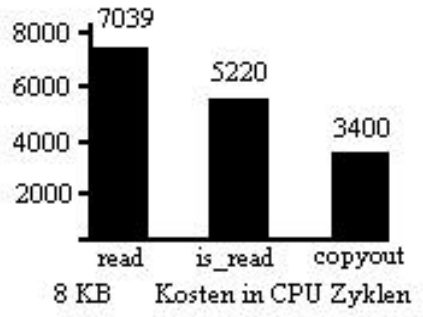
7. Gebe per-file lock frei, damit die Prozesse im Holding Tank wieder arbeiten können.
8. Gebe per-process lock frei, damit andere Replugger diese Funktion bearbeiten können.

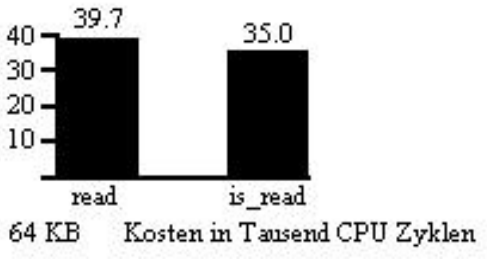
Mittels dieser Vorgehensweise können nun die oben genannten Probleme gelöst werden. In den meisten Fällen von Verallgemeinerung (Unspecialization) wird das allgemeinere read anstatt des is_read benutzt. In diesem Fall wird der File Deskriptor als unspezialisiert markiert und die is_read Funktion wird für den Garbage Collector markiert, welcher bei close aufgerufen wird.

5. Betrachtung der Performance

Die Umgebung für die Benchmarks war ein Hewlett-Packard 9000 Serie 800 G70 (9000/877) Zwei-Prozessor Server in Single-User Mode. Dieser Server war mit 128 MB Speicher ausgestattet. Die beiden PA7100 Prozessoren laufen mit 96 Mhz und jeder beinhaltet 1 MB Instruktionscache und 1 MB Datencache.







	1 Byte	8 Kbyte	64 Kbyte
read	3540	7039	39733
is_read	979	5220	35043
Einsparung	2561	1819	4690
% Verbesserung	72%	26%	12%
Speedup	x3.61	x1.35	x1.13
copyout	170	3400	---

Betrachtung zur Einsparung der Kosten beim Overhead

Das Testprogramm wurde folgendermaßen aufgebaut. Es besteht aus einer Schleife welche zuerst open ausführt, dann einen Zeitstempel holt, N Bytes einliest, einen erneuten Zeitstempel holt und zuletzt das File wieder schließt. Die Zeitstempel werden aus einem Prozessor Register ausgelesen, welches nach jedem Prozessortakt incrementiert wird.

Es wurden für drei verschiedene Lesegrößen 1 Byte, 8 KByte und 64 KByte Mikrobenchmarks nach dem oben genannten Testprogramm durchgeführt. Hierbei werden die Einsparungen in Bezug auf den Overhead bei einer Leseoperation deutlich. Als Vergleich für eine maximale Schranke der Einsparungsmöglichkeit wurden die Diagramme mit copyout ergänzt.

Leseoperationen, welche die Blockgröße überschreiten verlieren die Möglichkeit, einfach Daten kontinuierlich von der vorhergehenden Position in den aktuellen Buffercache Block schreiben zu können. Dies führt dazu, daß bei den 8 Kbyte Leseoperationen, welche die Blockgröße überschreiten, ein Performanceverlust sichtbar wird. Für noch größere Leseoperationen wird der Performancegewinn noch niedriger, da weniger Overhead bei den Leseoperationen vorhanden ist.

Allgemeine Betrachtung der Kosten für die Spezialisierung

Die oben genannten Performance-Gewinne lassen sich nur durch Kosten an anderen Stellen im System verwirklichen. Die meisten Kosten laufen bei open an, hier müssen nämlich 8 Invarianten und Quasi-Invarianten überprüft werden. Diese betragen in diesem Beispiel 90 Instruktionen und ein lock/unlock Paar. Falls eine Spezialisierung stattfinden kann, muß in open zusätzlich Speicher, für die spezialisierte Funktion, angefordert werden. Close muß im Vergleich zu open nur den Speicher für die spezialisierte Funktion wieder frei geben. Eine Speicheranforderung im Kern braucht 119 Takte und eine Freigabe 138 Takte.
Wenn man nun die beiden Varianten von open vergleicht, dann stellt man fest, daß das neue open teurer geworden ist. (open 5227 Takte; neues open 5582 Takte) Vergleicht man nun einen open-Aufruf und einen read-Aufruf zusammen in beiden Systemen, so stellt man fest, daß sich trotzdem eine Kostenreduzierung bei der Spezialisierung ergibt.

Weitere zusätzliche Kosten ergeben sich durch die Guards und das Replugging. Diese werden hier jedoch nicht mehr ausführlich betrachtet.
Spezialisierung führt zu einer Kostenreduzierung bei den spezialisierten Funktionen, benötigt aber zusätzlichen Aufwand an anderen Stellen im System (open ; close ; Guards ; Replugging). Bei dem vorliegenden Systemdesign kommen Guards jedoch nur an wenig benutzten Ausführungspfaden vor und daß Replugging wird nur in den seltenen Fällen aufgerufen, in welchen sich Quasi-Invarianten ändern.

Eine unformale Performance-Analyse dieser Kosten und ein Vergleich mit dem Gewinn läßt sich mit den folgenden Formeln stellen.

$$Overhead = \sum_i f^{\,i}_{syscall} * Guard^{\,i} + Open + Close + f_{Replug} * Replug$$

$$\text{Overhead} + f_{is} * is\_read < f_{is} * read$$

Jeder $Guard^{\,i}$ (in verschiedenen Kernaufrufen) wird $f^{\,i}_{syscall}$ mal ausgeführt.
Replug wird f_{Replug} mal ausgeführt.

f_{is} ist die Anzahl der Aufrufe von is_read.

In der hier getesteten Version von is_read wurden noch nicht alle möglichen Quasi-Invarianten ausgenutzt. Die hier benutzte is_read Funktion enthält also mehr Interpretation und Traversal -Arbeit, als unbedingt notwendig wäre.

6. Zusammenfassung

Incremental und Optimistic Specialization führt zu einer Reduzierung des Overheads in häufig genutzten Funktionen. Für die notwendigen Kontrollen und Spezialisierungsschritte wird ein zusätzlicher Aufwand an verschiedenen anderen Stellen im System nötig. Diese Stellen liegen jedoch in weniger genutzten Funktionen, so daß sich ein Gewinn bei den Gesamtkosten für einzelne „Funktionsabfolgen“ (open ? read ? read ... read ? close) einstellt.

Die größte Schwierigkeit bei der Konstruktion solcher Systeme, ist die Überwachung der Quasi-Invarianten mittels Guards. Jede Funktion, bei der eine mögliche Änderung der Quasi-Invarianten auftreten kann, muß überwacht werden.

Man könnte sich demnach vorstellen, eine solche Spezialisierung nur für einige wenige häufig genutzte Funktionen in Betriebssystemen zu realisieren, da jeder Gewinn durch Spezialisierung die Komplexität des Gesamtsystems erhöht.

7. Andere Spezialisierungen und Ausblick

Eine Spezialisierung nach dem Konzept von Synthetix wurde bereits in verschiedenen Bereichen realisiert:

- SUN RPC unter Solaris
- Signalverarbeitung unter LINUX
- Multimedia-Anwendungen (Speziell Verfahren zur Datenübertragung über Netze mit extremen Schwankungen der Übertragungszeit.)
- usw.

Die bisherigen Ergebnisse bei den verschiedenen Systemen hat gezeigt, daß eine weitere Betrachtung sinnvoll ist. Basierend auf den bisherigen Experimenten soll nun eine Zusammenstellung von Tools für C Programmierer entwickelt werden, welche diesem helfen sollen, mögliche Spezialisierungen zu entdecken und die Konsistenz zu sichern.

8. Literatur

[1] *Optimistic Incremental Specialization: Streamlining a Commercial Operating System*
Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, ...
Departement of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology

[2] *Fast Concurrent Dynamic Linking for an Adaptive Operating System*
Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, Jonathan Walpole
Departement of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology