# Implementierung eines effizienten Prozeßumschalters auf Benutzerebene

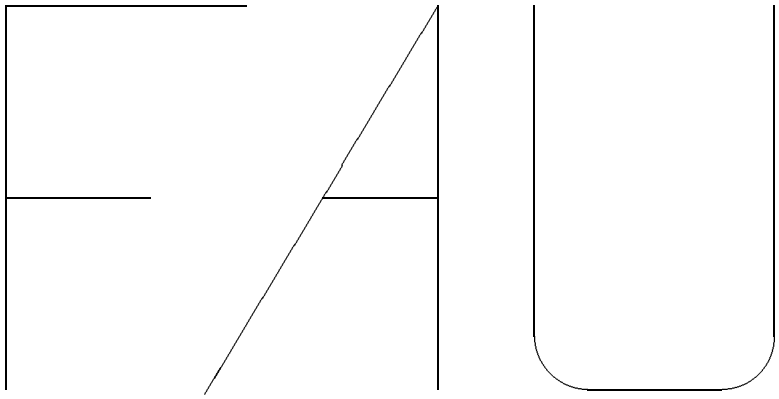
Uwe Reder

Februar 1995 02-95

Studienarbeit

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)



# Implementierung eines effizienten Prozeßumschalters auf Benutzerebene

Studienarbeit im Fach Informatik

vorgelegt von

*Uwe Reder*

geboren am 21. Juni 1969 in Ingolstadt

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

| | |
|---|---|
| Betreuer: | *Prof. Dr. rer. nat. Fridolin Hofmann*<br>*Dr. Ing. Claus-Uwe Linster*<br>*Dipl. Inf. Frank Bellosa* |
| Beginn der Arbeit: | 7. July 1994 |
| Abgabe der Arbeit: | 13. Februar 1995 |

Ich versichere, daß ich meine Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 13. Februar 1995

# Inhaltsverzeichnis

# 1 Motivation

Zur Lösung von komplexen mathematischen Problemen werden vermehrt leistungsfähige Mehrprozessorsysteme herangezogen. Deren spezielle Architektur erfordert ein Überdenken des traditionellen Programmiermodells. Man unterscheidet dabei zwischen zwei grundlegenden Architekturmodellen. Dies sind zum einen die nachrichtenbasierten Systeme und zum anderen die speichergekoppelten Systeme. Die einzelnen Knoten nachrichtenbasierter Systeme besitzen privaten Speicher. Zur Kommunikation verpacken die Prozessoren ihre Botschaften in Datenpakete und versenden diese über das Verbindungsnetzwerk. Erfordert eine Applikation einen häufigen Datentransfer zwischen den einzelnen Berechnungseinheiten, so erweist sich dieses Architekturmodell als nur bedingt brauchbar. Auch eine Migration von Prozessen auf andere Knoten innerhalb des Rechners ist aufgrund des privaten Speichers nicht trivial zu lösen. Aus diesen und anderen Gründen sollen im folgenden nur speichergekoppelte Systeme mit einem gemeinsamen Adreßraum betrachtet werden. Darunter versteht man, daß der Inhalt einer bestimmten Adresse, von allen Knoten im System aus betrachtet, identisch ist. Man spricht dann von Multiprozessorsystemen mit „global shared memory".

Eine hohe Skalierbarkeit des Gesamtsystems ist heute ein wichtige Forderung an speichergekoppelte Mehrprozessorsysteme. Daher ist es kaum noch sinnvoll, Software starr für $n$ Prozessoren zu parallelisieren. Vielmehr sollte durch eine vorgenommene Skalierung die Parallelität der Applikation dynamisch wachsen. Oftmals ist es aber nur mit erheblichem Programmieraufwand möglich oder gar überhaupt unmöglich, einen Algorithmus für die $n$ Prozessoren des Systems symmetrisch zu parallelisieren[1]. So kann der Zeitbedarf für ein Teilproblem meist gar nicht bestimmt werden, was mit dem Begriff der „symmetrischen Parallelisierung" nicht vereinbar ist. Das Ziel wird es also sein, unabhängig von der Anzahl der Prozessoren, eine Parallelisierung in eine möglichst hohe Anzahl von potentiell parallel abarbeitbarer Teilprobleme vorzunehmen. Diese Teilprobleme sind dynamisch zur Laufzeit auf den Prozessoren eines Mehrprozessorsystems zu verteilen. Für eine solche Aufgabe weisen herkömmliche Prozeßkonzepte einen zu großen Verwaltungsaufwand auf.

Nach dem Ausfall eines Prozessorknotens kann es bei fehlertoleranten Rechenanlagen auch zu einer Abwärtsskalierung kommen. Das heißt, daß das Gesamtsystem nach einer automatischen Umstrukturierungsphase die Arbeit ohne den ausgefallenen Knoten fortsetzt. In diesem Fall gelingt es einer fein granularen, dynamisch verteilbaren Applikation leichter, wieder eine symmetrische Parallelisierung zu erlangen.

Aber auch auf Monoprozessormaschinen kann es sinnvoll sein, eine Applikation hochgradig zu parallelisieren. In der Literatur wird hier auf Datenbanken, Netzwerk Server und auf graphische Benutzerschnittstellen verwiesen [PKBS91]. Beispielsweise liegt es bei letzteren nahe, jede Funktionseinheit der graphischen Oberfläche (widget) durch einen sehr leichtgewichtigen Prozeß zu verwalten. Wird eines dieser widgets zum Beispiel durch einen Mausklick aktiviert, so reiht man den Verwalterprozeß des ausgewählten widgets in die Warteschlange der laufbereiten Prozesse ein. Man muß sich natürlich auch hierbei darüber

[1] Eine Applikation ist genau dann „symmetrischen parallelisiert", wenn der Resourcenverbrauch der Teilkomponenten, ohne Interaktion des Betriebssytems oder anderer Applikationen, auf allen Knoten gleich hoch ist.

im klaren sein, daß konventionelle Prozeßverwaltungen für solche fein granularen Aufgaben wenig geeignet sind.

Eine gemeinsame Forderung all dieser Anwendungen ist eine höhere Effizienz bei der Erzeugung und Vernichtung solcher Prozesse sowie für deren Schedulingstrategien und den Kontextwechsel zwischen zwei Prozessen.

In dieser Arbeit soll eine Prozeßverwaltung vorgestellt werden, die imstande ist, die geforderte Effizienzsteigerung bereitzustellen.

# 2 Konzepte zur Prozeßverwaltung

Ein zentraler Bestandteil von Betriebssystemen ist die Prozeßverwaltung. Sie beinhaltet die Erzeugung, die Verwaltung während der Laufzeit und das Verwerfen eines Prozesses. Ein solcher Prozeß besteht jeweils aus einer Ablaufumgebung und mindestens einem Aktivitätsträger. Als aktive Gebilde benötigen Prozesse einen solchen Aktivitätsträger. Dieser hat die Aufgabe, die in der Ablaufumgebung enthaltenen Operationen in der festgelegten Reihenfolge zur Ausführung zu bringen. Ist ein Aktivitätsträger inaktiv, so hat er immer eine Aktivität zur Ausführung anstehen [Hof91]. Eine Applikation ist parallelisiert, falls ihr zu irgendeinem Zeitpunkt gleichzeitig mehrere Aktivitätsträger zugerechnet werden.

Die Effizienz paralleler Bearbeitung ist also in hohem Maße von den, vom Betriebssystem zur Verfügung gestellten Grundoperationen abhängig. Sind die Kosten für die Erzeugung und Verwaltung dieser Parallelität hoch, so zeigt selbst eine grob granular gestaltete Applikation kein befriedigendes Laufzeitverhalten. Dennoch kann selbst eine sehr fein granular gestaltete Applikation hervorragende Performanz erzielen. Dazu ist eine hohe Effizienz bei der Verwaltung dieser Parallelität anzustreben [MuSG93].

Eine weitere, an die Prozeßverwaltung gestellte Anforderung, ist die anwendungsorientierte Anpaßbarkeit. Es ist unwahrscheinlich, daß ein einziges Konzept in der Lage ist, alle Bedürfnisse einer Applikation optimal zu erfüllen.

## 2.1 Schwergewichtige Prozesse

Ein „Schwergewichtiger Prozeß“ besteht aus einer Ablaufumgebung und *genau einem* Aktivitätsträger. Die Ablaufumgebung stellt im wesentlichen einen virtuellen Adreßraum mit Verwaltungsinformationen für zugeteilte Betriebsmittel dar. Eingesetzt werden diese Schwergewichtigen Prozesse zum Beispiel im Betriebssystem UNIX.

Die Kommunikation zweier Prozesse ist beispielsweise durch ein teilweises Überlappen ihrer Adreßbereiche realisiert. Man spricht von „shared memory“ Segmenten.

Da bei der Erzeugung von Schwergewichtigen Prozessen der gesamte Adreßraum des initiierenden Prozesses dubliziert wird, entstehen schon zu diesem Zeitpunkt erhebliche Kosten[2]. Beim Verwerfen eines solchen Prozesses muß auch der Adreßraum verworfen werden. Dies wiederum führt zu einem nicht unerheblichen Aufwand.

Aber auch bei einer Prozeßumschaltung muß, selbst innerhalb einer Anwendung, eine Adreßraumumschaltung erfolgen. Dies erfordert das Neuladen der Register der MMU[3]. Während der eigentlichen Arbeit des Prozesses entstehen aber noch weitere Folgekosten dadurch, daß dessen Daten aus dem Prozessorcache und dem TLB[4] verdrängt wurden. Die zeitkritische Größe ist dabei der für den virtuellen Adreßraum wieder aufzubauende TLB.

[2] Allerdings müssen nicht alle Bereiche wirklich kopiert werden. Read-only Bereiche, wie das Text- und Datensegment werden durch mapping in den Adreßraum des neuen Prozesses eingeklinkt.

[3] Die Memory Management Unit (MMU) ist Teil der Hardware eines Rechners und erledigt die Abbildung von virtuellen Adressen auf physikalische Adressen. Als Grundlage für diese Abblidung dient die Seiten-Kachel Tabelle (SKT).

[4] Der Translation Lookaside Buffer (TLB) ist der Cache der MMU. Einmal durchgeführte Adreßumrechnungen werden dort gespeichert. Ohne TLB müßte die CPU bei jedem Speicherzugriff erneut die Zuordnung zwischen virtuellem und physikalischem Adreßraum bestimmen.

Neuere Prozessoren, wie der SPARC Prozessor von Sun, verfügen daher über mehrere TLBs. Dies geschieht mit dem Ziel, jedem Prozeß seinen eigenen TLB zu garantieren.

Alle Operationen im Zusammenhang mit Prozeßerzeugung, Prozeßvernichtung, Scheduling und Kontextwechsel werden dabei starr durch den Kern vorgenommen und können nicht anwendungsspezifisch durch den Benutzer angepaßt werden.

Selbst für eine grob granulare Parallelisierung sind Schwergewichtige Prozesse also kaum geeignet.

## 2.2 Kernel Threads (Leichtgewichtige Prozesse)

Aus den Erfahrungen mit Schwergewichtigen Prozessen entstand die Idee, die Ablaufumgebung und den Aktivitätsträger zu trennen. Dadurch wurde es möglich, mehrere Aktivitätsträger in einer Ablaufumgebung zu beherbergen. Der Mach Kern ist ein klassischer Vertreter dieser Vorgehensweise. Die Ablaufumgebung wird unter Mach als „Task“ bezeichnet. Eine solche Task besteht aus einem virtuellen Adreßraum und Verwaltungsinformationen für zugeteilte Betriebsmittel. Threads ihrerseits stellen die Aktivitätsträger dar. Jeder Thread wird durch einen Programmzähler sowie einen eigenen Stack beschrieben und arbeitet auf einem unabhängigen Registersatz. Eine Task kann beliebig viele solcher Threads in sich aufnehmen [Kle92].

Die Umschaltung zwischen ihnen ist im Vergleich zu den Schwergewichtigen Prozessen erheblich billiger. Da alle Threads einer Anwendung in einem gemeinsamen virtuellen Adreßraum ablaufen, besteht bei einem Threadwechsel innerhalb dieser Anwendung nicht die Notwendigkeit einer Adreßraumumschaltung. Nur der Registersatz, der Stapelzeiger und der Befehlszählerstand müssen gesichert werden.

Die Operationen zur Verwaltung dieser Threads, als auch deren Verwaltungsstrukturen, sind dennoch weiterhin Teil des Betriebssystems und können daher nur durch relativ teure Betriebssystemaufrufe erreicht werden [KlRi93].

Auch bei den Kernel Threads besteht keinerlei Möglichkeit zur anwendungsorientierten Anpassung. Ein Schritt in diese Richtung ist allerdings das unter Mach bekannte „handoff scheduling“. Ein sich blockierender Kernel Thread wird dadurch in die Lage versetzt, dem Scheduler im Kern einen Nachfolgerthread vorzuschlagen [MuSG93]. Damit kann die Schedulingstrategie in den Benutzeradreßraum verlegt werden. Der Kontextwechsel selbst findet trotzdem weiterhin im Kern statt.

## 2.3 User-Level Threads

Kernel Thread Implementierungen bieten meist nur ein einziges Modell an. Dieses Modell ist für alle denkbaren Applikationen tragfähig. Dafür werden allerdings hohe Kosten in Kauf genommen, obwohl dem Anwender oft ein weniger allgemeines Modell genügen würde. Beispielsweise ist es für eine Applikation die keine Fließkommaberechnungen durchführt überflüssig, Fließkommaregister während eines Kontextwechsels zu sichern [MuSG93]. Ein beim Programmierer der Applikation vorhandenes Wissen kann also nur sehr selten in die Threadverwaltung einfließen. Für eine fein granulare Applikation besitzen demnach selbst Kernel Threads kein annähernd optimales Laufzeitverhalten. Aus diesem Grund ist die

anwendungsorientierte Anpaßbarkeit eine zentrale Forderung an User-Level Threads. Im Mittelpunkt stehen dabei die freie Wahl der Schedulingstrategie, die Möglichkeit den Prozessorkontext zu verkleinern und die Datenstruktur der Warteschlange für das gegebene Problem zu optimieren.

Die Funktionen zur Verwaltung von User-Level Threads werden in Form einer Bibliothek zur Verfügung gestellt. Diese Bibliothek wird mit der Applikation gebunden. Alle Mechanismen und Datenstrukturen dieser Threads liegen daher außerhalb des Kerns, im virtuellen Adreßraum eines Prozesses beziehungsweise einer Task. Zur Ausführung einer Verwaltungsoperation benötigt man also keinen teuren Aufruf in den Adreßraum des Betriebssystemkerns.

Die erzeugten User-Level Threads nutzen die in der Ablaufumgebung vorhandenen Verwaltungsstrukturen gemeinsam. Öffnet zum Beispiel ein User-Level Thread eine Datei, so kann ein anderer Thread aus ihr lesen [StSh92]. Einzig und allein sind die von der Thread Bibliothek angebotenen Operationen privat. Da, wie beschrieben, alle User-Level Threads einer Applikation in einem gemeinsamen Adreßraum liegen, bestehen zwischen ihnen auch keinerlei Schutzmechanismen.

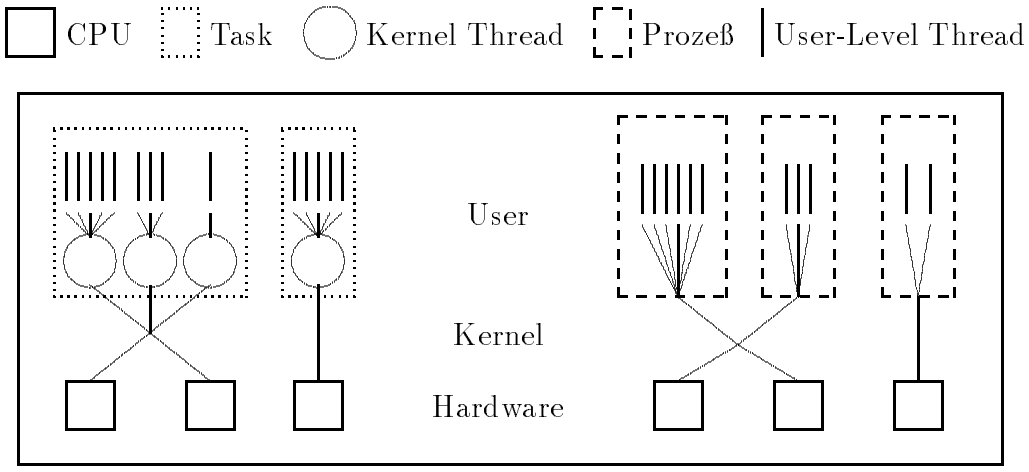



Abbildung 1: *Kernel Threads (links) und Schwergewichtige Prozesse (rechts) als Basis für User-Level Threads*

Die Zuteilung der einzelnen User-Level Threads an den Prozessor geschieht durch die Bibliothek. Die Basis hierfür bilden Schwergewichtige Prozesse beziehungsweise Kernel Threads. Diesen wird durch den Betriebssystemkern wiederum ein Prozessor zugeteilt und entzogen. Hierdurch entsteht eine zweistufige Prozessorzuteilung. Man bezeichnet dies als „two-level Scheduling“.

Probleme erwachsen dabei daraus, daß kein Informationsfluß zwischen den beiden Stufen stattfindet. Prozeßumschalter herkömmlicher Betriebssysteme besitzen kein Wissen über die Existenz weiterer Prozeßumschalter auf Benutzerebene. Daher werden deren Bedürfnisse im Kern ignoriert. Spezielle Schedulingstrategien für User-Level Threads werden so

unterlaufen und möglicherweise wertlos. [StSh92] schlägt daher vor, bestimmte User-Level Threads dem Prozeßumschalter im Kern bekannt zu machen. Solch eine Vorgehensweise ist zu wählen, falls Threads zeitkritische Ereignisse verfolgen sollen. Beispielsweise kann dies bei der Überwachung des Mauszeigers durch einen User-Level Thread nötig werden. Weitaus problematischer als das Unterlaufen von Schedulingstrategien ist allerdings die Blockierung eines User-Level Threads im Kern. Da ein konventioneller Kern einen User-Level Thread nicht als eine eigenständige Einheit wahrnimmt, kommt es zur Blockierung des gesamten Prozesses, obwohl eventuell weitere User-Level Threads laufbereit gewesen wären. Es wäre hier also denkbar, daß der Kern sich bei einer solchen Blockierung an den zuständigen Scheduler im Benutzeradreßraum wendet. Dieser könnte den blockierten Thread geeignet ablegen und einen Nachfolger bestimmen.

Ein weiteres Problem des two-level Schedulings ist auf Benutzerebene der Mangel an Wissen über Ereignisse im Kern. Sind zum Beispiel Prozessoren eines Mehrprozessorsystems ungleichmäßig stark belastet, so kann dies eine User-Level Thread Bibliothek meist nicht erkennen [MuSG93].

Lösungsansätze finden sich bei Anderson [And92] und Tucker/Gupta [TuGu89]. Auch im Betriebssystem Psyche werden Mechanismen zur Unterstützung eines Informationsflusses zwischen den Schedulerhierachien vorgestellt [MuSG93].

Neben dem two-level Scheduling existiert noch ein ähnliches Problem. Moderne Betriebssysteme für Mehrprozessorsysteme realisieren oftmals eine dynamische Lastverteilung durch Migration von Kernel Threads. Es handelt sich dabei um eine grob granulare Verteilung. Eine zweite, feinere Lastverteilung auf Benutzerebene wird dadurch empfindlich gestört. Dieses Phänomen könnte analog zum two-level Scheduling als „two-level Migration“ bezeichnet werden. Meist erlaubt der Kern aber eine Bindung der Kernel Threads an einen bestimmten Prozessorknoten.

# 3 User-Level Threads

## 3.1 Initialisierung und Start eines User-Level Threads

Für die Initialisierung eines User-Level Threads ist unbedingt ein Speicherbereich, der später als Stapel fungiert und eine Adresse die beim Start des Threads angesprungen wird, erforderlich. Weiterhin sollte es möglich sein, ein oder mehrere Argumente zu übergeben. Der Befehlszähler und die Argumente werden bei der Initialisierung auf dem Stapel geeignet abgelegt, so daß diese während des Startvorgangs richtig interpretiert werden können. Daneben sind Verwaltungsstrukturen denkbar, aber nicht zwingend notwendig. Solche Verwaltungsstrukturen können Threadidentifikatoren auf Basis eines Zählers, Threadzustände, Zeitstempel und ähnliches mehr enthalten.

User-Level Threads können zu einem Zeitpunkt nur einen einzigen Zustand haben und nur einmal in einer Warteschlange aufgenommen sein. Wird dagegen verstoßen, so ist der erste Thread in der Lage Daten auf seinem Stapel zu verändern, ohne daß weitere Threads mit einem identischen Stapel dies erkennen könnten.

## 3.2 Modelle für einen Kontextwechsel zwischen User-Level Threads

Die Hauptaufgabe eines User-Level Thread Pakets besteht in der Bereitstellung der Mechanismen zum Wechseln des Kontextes. Dies geschieht durch eine `block()` Funktion, die der Thread direkt aufruft oder in die er hineingezwungen wird. Wurde der Zustand des blockierten Threads aufgrund einer Schedulingentscheidung restauriert, so tritt dieser wieder aus der `block()` Operation aus.

Innerhalb von `block()` wird der augenblickliche Zustand des Threads gesichert. Danach wird ein anderer Thread, durch Restaurierung seines gesicherten Zustandes, reaktiviert. Der Zustand eines Threads in der `block()` Funktion ist durch die Prozessorregister, die Rücksprungadresse sowie den aktuellen Stapelzeiger charakterisiert. Eine Referenz auf den blockierten Thread muß in der `block()` Operation oder in deren Umfeld in eine Warteschlange eingereiht werden, da sich dieser ja nur temporär blockiert und zu einem späteren Zeitpunkt wieder ablaufen soll. Der Thread, der durch `block()` restauriert werden soll, muß aus der Warteschlange der lauffähigen Threads entnommen werden. Ist damit zu rechnen, daß mehrere Aktivitätsträger parallel beziehungsweise quasiparallel auf einer Warteschlange arbeiten, so ist für geeignete Synchronisation zu sorgen. Weiterhin ist es wichtig, daß ein Thread erst dann wieder in einer ungesperrten Warteschlange auftaucht, sobald alle seine relevanten Daten abgelegt sind und ein Stapelwechsel erfolgt ist. Eine Sperre (Lock) sollte aus Effizienzgründen nur so kurz wie möglich gehalten werden, um eine potentiell parallele Abarbeitung nicht unnötig zu behindern.

Im folgenden sollen einige Modelle zum Kontextwechsel vorgestellt werden.

### 3.2.1 Scheduler Threads

Der Scheduler Thread ist ein gewöhnlicher User-Level Thread mit eigenem Kontext und Stapel. Er befindet sich aber nie in einer Warteschlange mit anderen User-Level Threads,

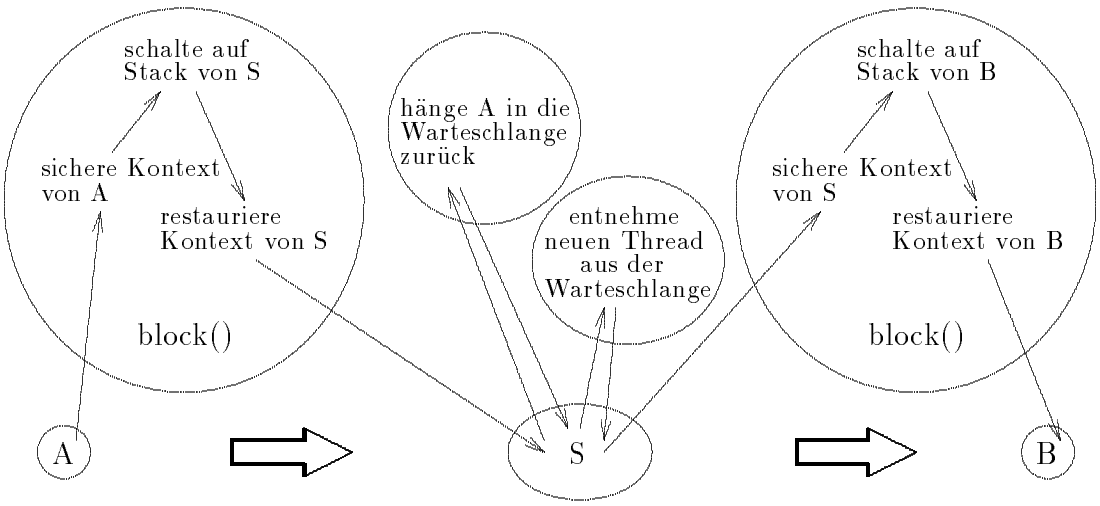



Abbildung 2: *Scheduler Thread*

sondern ist nur der `block()` Funktion bekannt. In einer Applikation existiert zu jedem Zeitpunkt für jeden Prozessor höchstens ein Scheduler Thread. Mehrere Prozessoren können aber auch einen Scheduler Thread gemeinsam nutzen. Blockiert sich ein User-Level Thread der Applikation, so wird sein Kontext im Rahmen der `block()` Funktion gesichert. Es kommt zu einer Umschaltung auf den Stapel des Scheduler Threads, der daraufhin seinen Kontext restauriert und aus der `block()` Funktion im Scheduler Thread zurückkehrt. Im Scheduler Thread wird der blockierte Thread in eine Warteschlange zurückgehängt und ein neuer, aus der Warteschlange der laufbereiten Threads, entnommen. Daraufhin ruft der Scheduler Thread nun seinerseits die `block()` Operation auf. Der folgende Kontextwechsel, vom Scheduler Thread zum zu deblockierenden Thread der Anwendung, erfolgt auf die gleiche Art und Weise, wie vorher der Kontextwechsel vom blockierten Thread zum Scheduler Thread. Diese Vorgehensweise erfordert also zwei Kontextwechsel, wie Abbildung 2 nochmals verdeutlicht.

Laut [Kep93] können sich beim Einsatz von Scheduler Threads auf Multiprozessorsystemen, deren CPUs keinen privaten Speicher besitzen, Probleme ergeben. Die Schwierigkeit würde darin liegen, an Prozessoren gebundene Scheduler Threads effizient zu lokalisieren. Vorausgesetzt sei, daß die User-Level Threads der Applikation von allen Prozessorknoten aus eine einzige `block()` Funktion gemeinsam nutzen. Diese Prämisse erlaubt die Migration von Threads zwischen den Knoten des Systems. Weiterhin basiert die These aus [Kep93] auf der Annahme, daß eine hohe Anzahl von Multiprozessorarchitekturen das Bestimmen der lokalen Prozessornummer nicht erlauben. Das eigentliche Problem tritt beim Kontextwechsel von einem Thread der Applikation zum Scheduler Thread auf. Gemäß [Kep93] ist ein Thread nur unter erheblichen Effizenzeinbußen in der Lage den Prozessorlokalen Scheduler als Nachfolger anzugeben. Dazu sollen im folgenden zwei Anmerkungen gemacht werden.

1. Der hier aufgezeigten Schwierigkeit liegt ein generelles Problem zugrunde, welches sich nicht ausschließlich auf das Modell des Scheduler Threads beschränkt. Wird der Versuch unternommen, einen Nachfolger aus einer Prozessor-lokalen Warteschlange zu entnehmen, so führt dies unabhängig vom verwendeten Kontextwechselmodell zur gleichen Problematik.

2. Es besteht die Möglichkeit jedem User-Level Thread der Applikation den lokalen Scheduler bekannt zu geben. Dazu würde bei der Initialisierung eines Threads der knotenlokale Scheduler in die Verwaltungsstrukturen des Threads eingetragen. Wird der Thread auf einen anderen Prozessorknoten migriert, so obliegt es dem Anfordernden die Referenz auf den neu zuständig gewordenen Scheduler Thread abzuändern. Während eines Kontextwechsels käme es daher zu keinen Effizienzeinbußen.

### 3.2.2 Preswitch

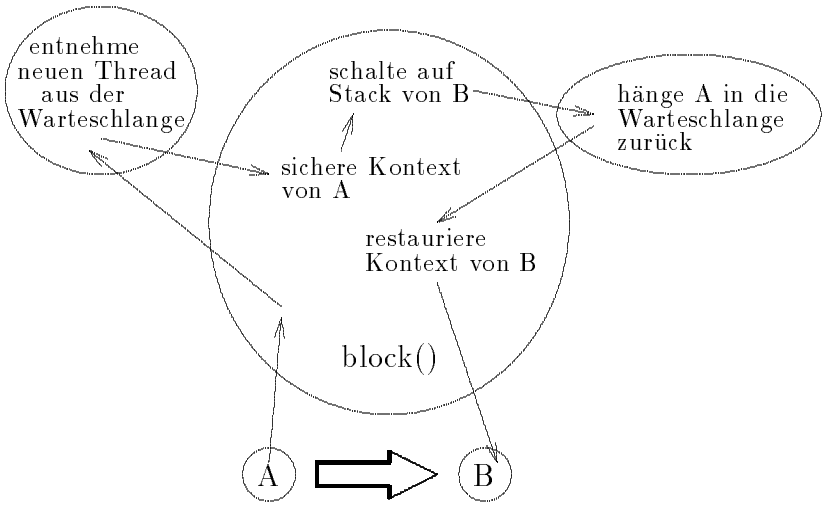



Abbildung 3: *Preswitch*

Das Zurückhängen des blockierten User-Level Threads in die Warteschlange erfolgt durch Aufruf einer Funktion, auf dem Stapel des zu deblockierenden Threads, innerhalb der `block()` Funktion.

Im Detail heißt das also, daß der sich blockierende Thread A in der `block()` Funktion zunächst seinen Nachfolger B aus der Gruppe der ablaufbereiten Threads bestimmt und daraufhin seinen Kontext sichert. Da die Selektion des Nachfolgers innerhalb eines Funktionsaufrufs geschieht, wird der Kontext von Thread A in der `block()` Operation nicht verändert. Das Bestimmen des Nachfolgers kann natürlich auch außerhalb der `block()` Funktion geschehen. Da ein Nachfolger also bereits bekannt ist, wird zur Ausführung der weiteren Aktionen sein Stapel verwendet. Der gesamte Kontext von A ist danach eingefroren. Auf dem Stapel von B kann nun problemlos ein Funktionsaufruf getätigt werden,

infolgedessen A in eine Warteschlange eingereiht wird. B restauriert nun seinen Kontext und kehrt aus der `block()` Funktion, die ihn beim Aufruf blockierte, zurück. Diese Vorgehensweise ist in Abbildung 3 graphisch dargestellt und benötigt wie alle folgenden Modelle nur einen Kontextwechsel.

Kritisch ist in diesem Konzept ein Kontextwechsel auf sich selbst. Solch ein Kontextwechsel ist immer dann erforderlich, falls sich ein laufbereiter Thread blockiert und die Warteschlange der laufbereiten Threads leer ist. Wie oben beschrieben, muß im Preswitch Model ein zu deblockierender Thread vorliegen, auf dessen Stapel dann der blockierte in eine Warteschlange eingetragen wird. Da zu diesem Zeitpunkt aber kein solcher Thread in der Warteschlange der laufbereiten Threads ist, wäre die `block()` Funktion verklemmt. Dieses Problem ist dadurch zu entschärfen, daß die Funktion, welche für die Bereitstellung des Nachfolgers zuständig ist, vor die `block()` Funktion gezogen wird. Findet sich kein Nachfolger, so ist die bevorstehende Blockierung abzubrechen. Falls der Thread zum Zeitpunkt der Blockierung nicht laufbereit war, muß er die unerfüllte Bedingung, die zu seiner Blockierung führte, erneut prüfen. Falls die Überprüfung wieder negativ ausfällt, blockiert sich der Thread ein weiteres Mal. Es entsteht also ein „Spin Lock“. Eine weitere Vorgehensweise ist die Einführung eines Idle Threads. Dieser wird immer dann aktiviert, wenn die Warteschlange zum Zeitpunkt der Entnahme eines Nachfolgers leer ist.

### 3.2.3 State in Register

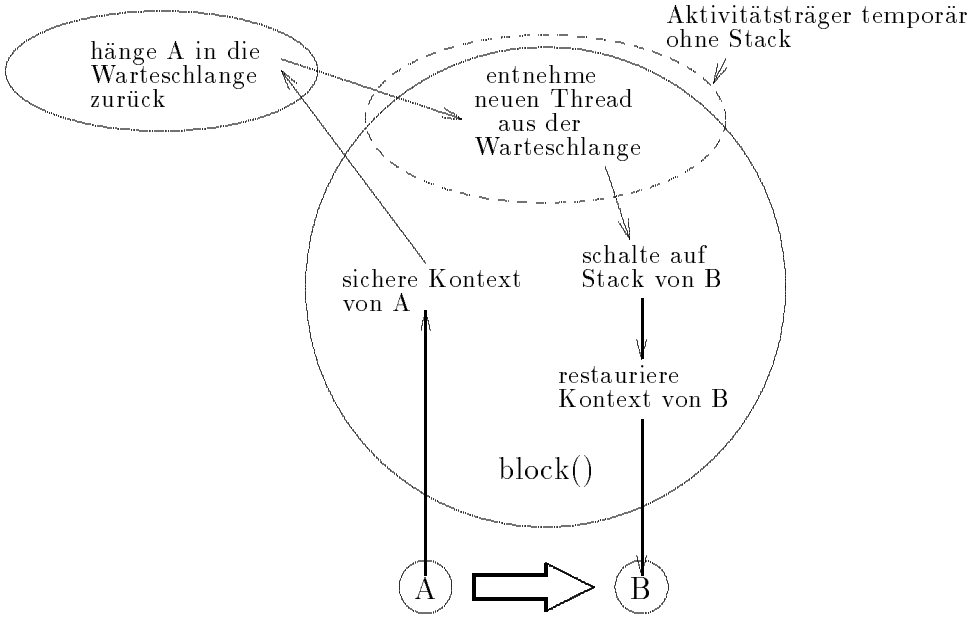



Abbildung 4: *State in Register*

Diese Art der Threadumschaltung (Abbildung 4) stellt eine Methode dar, die dem intuitiven Weg einer Umschaltung am nächsten folgt. Wird die `block()` Operation von einem

Thread aufgerufen, so wird versucht, so schnell wie möglich diesen Thread einzufrieren und in einer Warteschlange zu plazieren. Danach kann dessen Stapel ohne weiteres Sperren der Warteschlange nicht mehr genutzt werden. Da aber auch noch kein Nachfolger gefunden wurde, schließt sich auch die Möglichkeit aus, dessen Stack zu nutzen. Eine Strategie kann hier der vollkommene Verzicht auf einen Stapel und bloßes Nutzen der Prozessorregister sein. Daraus entstehen drei schwerwiegende Probleme. Der Sachverhalt, daß die nötige Schedulingentscheidung auf der Menge der Prozessorregister getroffen werden muß, stellt das erste dieser Probleme dar. Es ist nicht möglich einzelne Register temporär auf dem Stapel zwischenzuspeichern. Prozessoren, die nur über einen kleinen Registersatz verfügen, hätten hier wohl schon Probleme. Weiter sind Prozeduraufrufe ohne Stapel meist nicht zu realisieren und falls doch, nur unter strikten Restriktionen. In der PA-RISC Architektur von Hewlett-Packard kennt man sogenannte „leaf" Prozeduraufrufe, bei denen es unter bestimmten Umständen nicht nötig ist, einen Stapelbereich zu reservieren [HPC91]. Da der Einsatzbereich solcher Funktionen jedoch sehr beschränkt ist und dadurch die Portierbarkeit auf andere Architekturen beeinträchtigt wird, impliziert dies, daß der Code für die Schedulingentscheidung direkt und fest in der `block()` Funktion liegen muß. Das ist mit einer hohen Flexibilität, die von User-Level Threads im allgemeinen erwartet wird, nicht zu vereinbaren. Zum dritten bestehen von Seiten verschiedener Architekturen (SPARC) und Betriebssystemen (Mach 3.0) Restriktionen, die einen Aktivitätsträger auch temporär nicht ohne Stack dulden [Kep93].

### 3.2.4 Locking

Diese Methode geht ähnlich wie „State in Register" vor. Sie erlaubt aber durch Sperren der Warteschlange, in die der blockierte Thread aufgenommen wurde, eine Weiterverwendung des alten Stapels. Ist ein Nachfolger gefunden und dessen Stapel als aktueller Stapel übernommen worden, kann die Warteschlange freigegeben werden. Dies erfordert aber auch im günstigsten Fall ein relativ langes Halten des Locks. Ist das Sperren und Freigeben der Semaphore innerhalb der `block()` Funktion implementiert, so wird die Portierbarkeit als auch die Flexibilität herabgesetzt [Kep93]. Mit einem Verschieben der Synchronisation nach außen, verlängert man nochmals die Zeit für das Halten des Locks.

### 3.2.5 Stateless Schedulers

Dieser Ansatz basiert wiederum auf der „State in Register" Methode und stellt neben dem „Locking" einen weiteren Vorschlag zur Bereitstellung eines Stapels dar. Dadurch werden die Restriktionen, die durch das Fehlen eines Stapels auferlegt wurden, entschärft. Die Idee ist hier, einen Speicherbereich als Stapel während des Kontextwechsels auszuweisen (Abbildung 5). Auf diesem Stapel können dann Funktionsaufrufe getätigt werden. Es wird also nicht zu einem Scheduler Thread geschalten, sondern nur ein Stapel für Funktionsaufrufe bereitgestellt. Der Nachteil eines weiteren Kontextwechsels wird dabei umgangen. Da hier der Scheduler nur eine Funktion ist und damit über das Funktionsende hinaus keinen Status hat, spricht man von „Stateless Schedulers".

Wie schon bei den Scheduler Threads, erwähnt [Kep93] auch hier die Schwierigkeiten

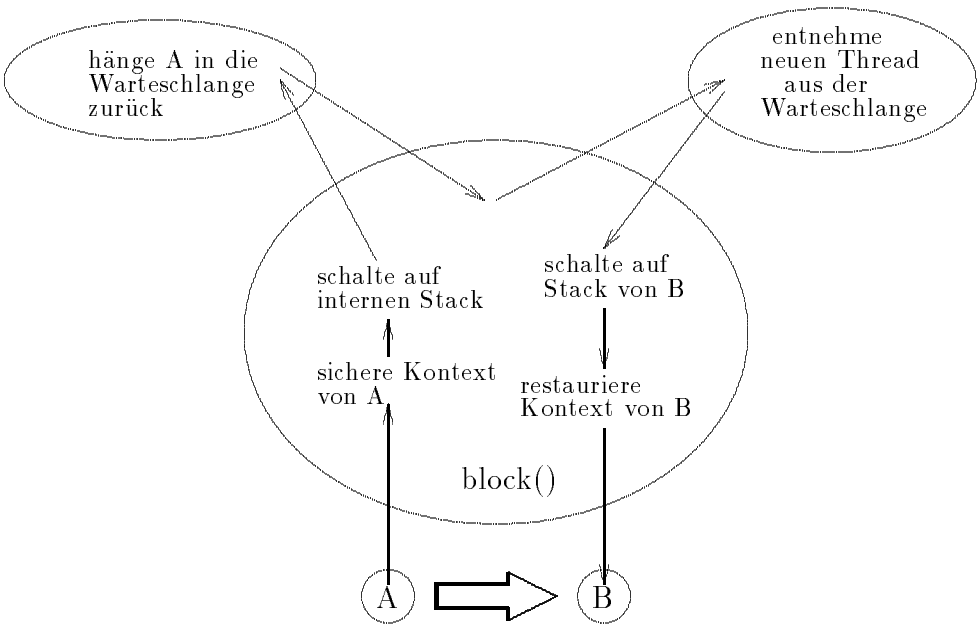



Abbildung 5: *Stateless Schedulers*

die mit dem Einsatz auf Multiprozessorsystemen ohne privaten Speicher verbunden sind. Wird auf jedem Prozessor ein unabhängiger Stateless Scheduler instanziiert, so sind die einzelnen internen Stapel seperat anzulegen. Die Zeiger auf diese Stapel verhalten sich im Bezug auf die obige Problematik, äquivalent zu den Referenzen auf die Scheduler Threads. Die in Kapitel 3.2.1 getroffenen Aussagen und Anmerkungen gehen daher sinngemäß auf die Stateless Scheduler über.

## 3.3 Verwerfen eines User-Level Threads

Das Verwerfen eines User-Level Threads kann durch einen vereinfachten Kontextwechsel implementiert werden. Dabei entfällt das Sichern des Kontextes völlig. Das Zurückschreiben des Threads in eine Warteschlange wird ersetzt durch die Freigabe des Stapelbereichs und eventuell vorhandener thread-eigener Datenstrukturen. Dabei entfällt natürlich auch die sonst nötige Synchronisation. Die anschließende Umschaltung auf seinen Nachfolger vollzieht sich analog zum Kontextwechsel.

Für ein flexibles Thread Paket wäre es dabei wünschenswert, daß die Freigabe der thread-eigenen Strukturen benutzerspezifisch als Funktion zu implementieren ist. Dies kann aber nur dann funktionieren, wenn der Aufruf dieser Funktion nicht mehr auf dem Stapel des zu verwerfenden Threads stattfindet. Der Stapel würde andernfalls in der benutzerdefinierten Funktion verworfen, obwohl die Funktion sich noch auf diesem Stapel in Ausführung befindet. Unter den oben genannten Schwierigkeiten muß, entweder ganz auf den Stapel verzichtet werden, oder vor dem Aufruf der Funktion bereits ein neuer Stack bekannt sein. Die Möglichkeit des „Lockings“ entfällt, da bei diesem Modell weiterhin auf dem alten Stapel

gearbeitet wird. Wie oben beschrieben, ist dies jedoch nicht möglich.

In Bezug auf die Schedulingentscheidung gehen die Probleme der vorgestellten Kontextwechselmodelle auf die von ihnen abgeleiteten Konzepte für das Verwerfen über.

# 4 Das Programmiermodell des QuickThreads Pakets

Das QuickThreads Paket [Kep93] stellt Basisoperationen zur Initialisierung, Start, Vernichtung und zum Kontextwechsel zwischen Threads bereit. Es handelt sich dabei also nicht um ein User-Level Thread Paket für Endanwender, sondern dient bei der Generierung anderer Thread Pakete als Basis. Diese auf Basis der QuickThreads generierten Thread Pakete sollen im folgenden als „aufsetzende Thread Pakete“ bezeichnet werden (Abbildung 6). Um eine wirklich breite Basis für Thread Pakete aller Art zu schaffen, entscheidet sich der Autor der QuickThreads für Flexibilität durch Anbieten des kleinsten Nenners, der möglichst vielen aufsetzenden Thread Paketen gemein ist. Das Anlegen der thread-eigenen Stapelbereiche, die Schedulingstrategien, die Warteschlangenverwaltung und die Definition der zugehörigen Datenstrukturen ist bewußt dem zu implementierenden Thread Paket überlassen.

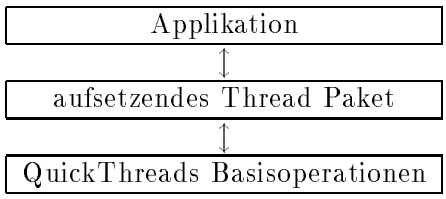



Abbildung 6: *Das QuickThreads Paket als Basis für aufsetzende Thread Pakete*

Die Headerdatei `qt.h` definiert die QuickThreads Schnittstelle. Alle Programme, die das QuickThreads Paket nutzen, müssen `qt.h` einbinden. Die QuickThreads Funktionen selbst sind zu einer Bibliothek namens `libqt.a` gebunden. Diese ist mit der aufsetzenden Anwendung durch die Compiler Option `-lqt` zu linken.

Jeder Thread besitzt einen eigenen Stapel. Zur eindeutigen Identifizierung eines Threads dient sein aktueller Stapelzeiger. Dieser ist vom Typ `qt_t*`.

## 4.1 Zustandsübergänge im QuickThreads Paket

QuickThreads können sich in fünf verschiedenen Zuständen befinden. Diese Zustände sind „uninitialisiert“, „initialisiert und lauffähig“, „laufend“, „blockiert und lauffähig“ oder „verworfen“. Allokiert ein aufsetzendes Thread Paket einen Speicherbereich zur späteren Verwendung als Stapel für einen Thread, so definiert der Zeiger auf diesen Speicherbereich bereits einen uninitialisierten Thread [Kep93]. Der Aufruf von `QT_ARGS()` beziehungsweise `QT_VARGS()` initialisiert den Stapel des Threads. Der Thread ist jetzt ablaufbereit, aber noch in keine Warteschlange aufgenommen. Veranlaßt das aufsetzende Thread Paket dies, so befindet sich dieser im Zustand „blockiert und lauffähig“. Wird dieser Thread aufgrund einer Schedulingentscheidung aus der Warteschlange der laufbereiten Threads entnommen und ein Kontextwechsel eingeleitet, so geht er in den Zustand „laufend“ über. In diesem verbleibt der Thread bis er wieder blockiert oder gänzlich verworfen wird.

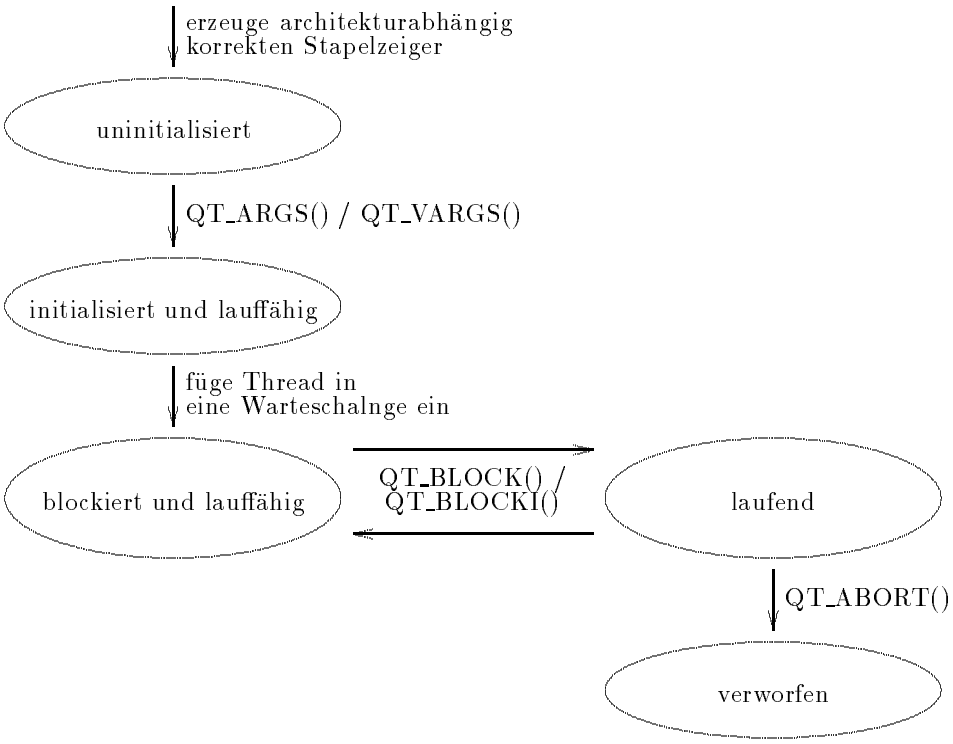



Abbildung 7: *Die möglichen Zustände eines QuickThreads*

## 4.2 Erzeugung eines uninitialisierten Threads

```
int QT_STKALIGN;
qt_t *QT_SP(void *sto, int size);
```

Nachdem das aufsetzende Thread Paket einen Speicherbereich für den Stapel des zu erzeugenden Threads bereitgestellt hat, muß der Stapelzeiger innerhalb dieses Bereichs architekturabhängig gesetzt werden. Bei Prozessoren, deren Stapel von höheren Adressen abwärts zu niedrigeren Adressen wächst, ist der Stapelzeiger auf das Ende des Speicherbereichs zu richten und umgekehrt auf den Anfang desselben. Dies erledigt das Makro `QT_SP()`. Das Argument `sto` ist dabei die Adresse des Speicherbereichs und `size` dessen Länge. Weiter ist korrektes Alignment zu beachten. Hierzu stellt das QuickThreads Paket in `QT_STKALIGN` den Wert für das architekturabhängige Alignment des Stapels bereit. Der Stapelzeiger muß also so korrigiert werden, daß er ohne Rest durch `QT_STKALIGN` teilbar ist.

## 4.3 Initialisierung eines QuickThreads

Bei der Initialisierung wird dem Thread eine Aufgabe zugewiesen. Die Aufgabe setzt sich aus einer Threadfunktion und den ihr zu übergebenden Argumenten zusammen. Ferner kann

der Rahmen, in dem die Threadfunktion ablaufen soll, beschrieben werden. Für diesen Aufgabenbereich stellt das QuickThreads Paket zwei Varianten bereit.

```
typedef void *(qt_userf_t) (void *u);
typedef void (qt_only_t) (void *u, void *t, qt_userf_t *userf);

typedef void *(qt_vuserf_t) (...);
typedef void (qt_startup_t) (void *t);
typedef void (qt_cleanup_t) (void *t, void *vuserf_return);

qt_t *QT_ARGS(qt_t *sp, void *u, void *t, qt_userf_t *userf,
              qt_only_t *only);

qt_t *QT_VARGS(qt_t *sp, int nbytes, void *vargs, void *t,
               qt_startup_t *startup, qt_vuserf_t *userf,
               qt_cleanup_t *cleanup);
```

Mit `QT_ARGS()` wird dem aufsetzenden Thread Paket ein Makro zur Verfügung gestellt, mit dessen Hilfe genau ein Argument an den Thread der Applikation übergeben werden kann. Die Funktion `QT_VARGS()` bietet dagegen die Möglichkeit eine unbegrenzte Anzahl an Argumenten zu übergeben.

Das Argument `sp` ist sowohl für `QT_VARGS()` als auch für `QT_ARGS()` ein Zeiger auf den Stapel eines uninitialisierten Threads, wie er in Abschnitt 4.2 eingeführt wurde. Relativ zu dieser Referenz wird die Initialisierung des Stapels vorgenommen. Das Ergebnis dieser Funktionen ist ein Zeiger auf das Ende des initialisierten Stapels.

Nachdem ein mit `QT_ARGS()` initialisierter Thread deblockiert wurde, durchläuft dieser eine Startsequenz, infolge derer die Funktion `only()` aufgerufen wird. Diese ist vom aufsetzenden Thread Paket zu implementierenden. Sie hat typischerweise die Aufgabe, eine Ablaufumgebung für die Threadfunktion (`userf()`) aufzubauen, sie aufzurufen und die Umgebung nach deren Rückkehr wieder aufzulösen. Dabei ist `t` das Argument für `only()`, während `u` das Argument für `userf()` darstellt. Vor dem Verlassen von `only()` muß `QT_ABORT()` ausgeführt werden. Diese QuickThreads Funktion verwirft den aktuellen Thread und schaltet zu seinem Nachfolger. Da `QT_ABORT()` den Thread verwirft, kehrt `QT_ABORT()` niemals zurück. Die Rückkehr von `only()` zu seinem Aufrufer ist nicht erlaubt und führt bei einem Verstoß zu einer Fehlerbehandlung.

Die beliebig vielen Argumente der Threadfunktion werden für `QT_VARGS()` durch die Argumente `vargs` und `nargs` repräsentiert. `vargs` ist dabei der Zeiger auf eine Argumentenliste der Länge `nargs`. Es handelt sich dabei um die *stdargs* C Funktionalität. Das Feld, das `vargs` referenziert, wird bei der Initialisierung auf den uninitialisierten Stapel des Threads kopiert. Wird ein mit `QT_VARGS()` initialisierter Thread deblockiert, so wird ebenso wie bei `QT_ARGS()`, eine Startsequenz durchlaufen. Diese Startsequenz ruft nacheinander die Funktionen `startup()`, `userf()` und `cleanup()` auf. `QT_VARGS()` werden dazu Referenzen auf diese Funktionen übergeben. Ihre Implementierung ist Teil des aufsetzenden Thread Pakets. Dabei ist `startup()` der Konstruktor und `cleanup()` der Destruktor der Umge-

bung. Weiter erhält `QT_VARGS()` das Argument `t`, welches an `startup()` und `cleanup()` weitergereicht wird.

Die `userf()` Funktion ist im Gegensatz zu `QT_ARGS()` nicht mehr auf ein einziges Argument beschränkt. Ihr steht die Verwendung beliebig vieler Argumente offen, welche beim Aufruf von `QT_VARGS()` durch den Zeiger `vargs` repräsentiert werden.

Die `cleanup()` Funktion fordert nach Prototypendefinition noch ein weiteres Argument. Es handelt sich dabei um den Zeiger `vuserf_return`, der den Rückgabewert der `userf()` Funktion darstellt. Dieses Argument kann beispielsweise bei der Speicherung des Ergebnisses durch `cleanup()` eingesetzt werden. `cleanup()` hat zusätzlich die Aufgabe den abgearbeiteten Thread mit `QT_ABORT()` zu verwerfen. Die `cleanup()` Funktion kehrt deshalb auch niemals zu ihrem Aufrufer zurück.

Da Makros in C durch den Präprozessor textuell ersetzt werden, ist kein Funktionsaufruf zu `QT_ARGS()` erforderlich. Weiterhin sind keine exzessiven Kopierarbeiten nötig. Daher ist `QT_ARGS()` sehr effizient. `QT_VARGS()` dagegen ist als Funktion implementiert und muß möglicherweise zahlreiche Argumente aus dem `vargs` Feld auf den Stapel kopieren.

Wie dargelegt, verwenden `QT_ARGS()` und `QT_VARGS()` verschiedene Aufrufsemantiken. Würde `QT_VARGS()`, in Anlehnung an `QT_ARGS()`, die Funktion `userf()` über eine Zwischenfunktion (`only()`) aufrufen, so müßten für den Aufruf von `userf()` sämtliche Argumente aus dem Stapelbereich der Startsequenz in den der `only()` Funktion kopiert werden. Dies ist besonders bei einer hohen Zahl an Argumenten nicht tragbar. Die vier von `QT_ARGS()` übergebenen Argumente können allerdings bei den meisten Architekturen noch in Argumentregistern durchgereicht werden. Bei `QT_ARGS()` war die Intuition des Autors wohl die, an die Grenze des Machbaren vorzustoßen. Überträgt man das Modell von `QT_VARGS()` auf `QT_ARGS()`, so ist aber ein dritter Funktionsaufruf unumgänglich. Dadurch wäre die Laufzeit negativ beeinflußt worden. Die unterschiedliche Aufrufsemantik ist vom Standpunkt des Programmierers zwar nicht sinnvoll, aus Effizienzgründen aber dennoch vertretbar.

## 4.4 Kontextwechsel im QuickThreads Paket

Von den vorgestellten Methoden für einen Kontextwechsel, wird vom QuickThreads Paket das Preswitching verwandt. Mit diesem Modell sind alle anderen Modelle, bis auf das „State in Register“, nachbildbar. Dies trägt zu einer hohen Flexibilität des Pakets bei. Die QuickThreads weichen von dem in 3.2.2 dargestellten Preswitching-Modell insofern ab, daß die `block()` Funktion mit dem neuen Thread als Argument aufgerufen wird und dieser nicht erst innerhalb von `block()` ermittelt wird.

```
typedef void *(qt_helper_t)(qt_t *old, void *a0, void *a1);

void *QT_BLOCK(qt_helper_t *helper, void *a0, void *a1,
               qt_t *newthread);
void *QT_BLOCKI(qt_helper_t *helper, void *a0, void *a1,
                qt_t *newthread);
```

Für das Einleiten eines Kontextwechsels verfügt das Paket über zwei Alternativen. Zur Sicherung *aller* relevanten Prozessorregister des blockierten Threads steht dem aufsetzenden Thread Paket die Funktion `QT_BLOCK()` zur Seite, während `QT_BLOCKI()` nur die relevanten Integer Register ablegt. Benutzt eine Anwendung keine Fließkommaarithmetik so kann, anstatt `QT_BLOCK()` die Funkion `QT_BLOCKI()` eingesetzt werden. Da der Unterschied zwischen `QT_BLOCK()` und `QT_BLOCKI()` nur in der Anzahl der gesicherten Register besteht, sollen die beiden Funktionen hier gemeinsam beschrieben werden. Ist im folgenden von einer „`block()` Funktion“ die Rede, so ist dieser Terminus sowohl durch `QT_BLOCK()` als auch durch `QT_BLOCKI()` ersetzbar.

Vor dem Aufruf einer `block()` Funktion muß das aufsetzende Thread Paket einen Nachfolger für den vor seiner Blockierung stehenden Thread finden. Typischerweise geschieht dies durch Entnahme aus der Warteschlange der laufbereiten Threads. Diese ist vom aufsetzenden Thread Paket zu implementieren. Wird eine Warteschlange von mehreren physikalischen beziehungsweise virtuellen Prozessoren parallel genutzt, so ist für die nötige Synchronisation zu sorgen.

Da im QuickThreads Paket jeder blockierte Thread eindeutig durch seinen eingefrorenen Stapelzeiger beschrieben wird, ist dieser der `block()` Funktion zur Spezifikation des Nachfolgers zu übergeben. Hierzu dient das Argument `newthread`.

In der `block()` Funktion werden im ersten Schritt die durch die jeweilige Semantik vereinbarten Register auf den Stapel des sich blockierdenden Threads geschrieben. Danach erfolgt der Wechsel auf den durch das Argument `newthread` beschriebenen Stapel. Hier ist der Kontext des zu deblockierenden Threads abgelegt. Bevor dieser allerdings restauriert wird, ist für das Zurückhängen des blockierten Threads in eine adequate Warteschlange zu sorgen. Das Argument `helper`, das der `block()` Funktion übergeben wird, ist eine Referenz auf eine Funktion die dies bewerkstelligt. Sie ist vom aufsetzenden Thread Paket zu implementieren. Die `helper()` Funktion wird mit drei Argumenten aufgerufen. Das erste von ihnen ist der Zeiger auf den Stapel des gerade blockierten Threads (`old`). Die beiden weiteren Argumente (`a0`, `a1`) sind frei benutzbar. Sie wurden bereits der `block()` Funktion als Argumente übergeben und werden unverändert an `helper()` weitergereicht. Typischerweise dient eines dieser Argumente dem Verweis auf eine Warteschlange, in die der blockierte Thread einzuhängen ist. Nach dem Verlassen der `helper()` Funktion wird der Kontext des zu deblockierenden Threads restauriert. Über die ebenfalls wiederhergestellte Rücksprungadresse kehrt der Aktivitätsträger aus der `block()` Funktion, mit deren Aufruf sich der jetzt deblockierte Thread blockierte, zurück.

In allen momentanen Implementierungen können Threads, die alle Register und Threads die nur die Integer Register sichern, in einer gemeinsamen Warteschlange liegen. Dies wird dadurch realisiert, daß das Retten der Register in zwei ineinander verschachtelten Funktionsaufrufen geschieht. Die auf dem Stapel abgelegten Rücksprungadressen dirigieren dabei die Restaurierung. Dieses Verhalten ist allerdings in [Kep93] nicht dokumentiert. Vielmehr schlägt der Autor im obigen Dokument im Kapitel „Tuning“ sogar vor, die beiden getrennten Routinen aus Effizienzgründen zusammenzulegen. Würde dann `QT_BLOCK()` aufgerufen, so führte dies nach dem Sichern aller Register automatisch auch zu einem Restaurieren aller Register des Nachfolgerthreads, obwohl dieser sie eventuell gar nicht gesichert hat. Die

Folge wäre, daß der Stapelzeiger in den davorliegenden Stack Frame rutscht und sich der Thread damit in einem inkonsistenten Zustand befinden würde.

## 4.5 Verwerfen eines QuickThreads

```
void *QT_ABORT(qt_helper_t *helper, void *a0, void *a1,
               qt_t *newthread);
```

Das Verwerfen eines Threads ist vom Kontextwechsel abgeleitet. `QT_ABORT()` ist daher auch bis auf das Sichern des Kontextes identisch mit den `block()` Funktionen. Aus naheliegenden Gründen entfällt beim Verwerfen eines Threads das Sichern seines Kontextes. `QT_ABORT()` kehrt niemals zu seinem Aufrufer zurück.

# 5 Portierung des QuickThreads Pakets

Ziel des praktischen Teils dieser Studienarbeit war die Portierung des QuickThreads Pakets auf die PA[5]-RISC 1.1 Architektur von Hewlett-Packard und speziell auf den Multiprozessor Convex SPP.

## 5.1 Merkmale der PA-RISC Architektur

PA-RISC ist eine von Hewlett-Packard definierte Standardisierung einer Prozessorfamilie und deren Programmiermodell. Als Basis hierfür diente das RISC[6] Konzept. Jeder Prozessorbefehl hat unter PA-RISC eine feste Länge von 32-Bit, was das Bereitstellen des nächsten Befehls für die Pipeline erleichtert. Um den Folgebefehl in die Pipeline einzuleiten, ist es nicht erforderlich den Typ des aktuellen Befehls und damit dessen Länge zu kennen. Weiter ist der Befehlssatz stark registerorientiert ausgelegt. So greifen nur „load“ und „store“ Anweisungen auf den Speicher zu. Beispielsweise vermerkt der „branch subroutine“ Befehl die Rücksprungadresse nicht direkt auf dem Stapel, sondern übergibt sie der Prozedur in einem Register. In dem Registersatz eines PA-RISC Prozessors sind deshalb auch bestimmte Register für Spezialaufgaben reserviert (Abbildung 8) [HPB91]. Für die Speicherung von

| Register | Synonym | Beschreibung |
|---|---|---|
| `%r2` | `%rp` | Rücksprungadresse |
| `%r23` | `%arg3` | Integer Argumentübergaberegister 3 |
| `%r24` | `%arg2` | Integer Argumentübergaberegister 2 |
| `%r25` | `%arg1` | Integer Argumentübergaberegister 1 |
| `%r26` | `%arg0` | Integer Argumentübergaberegister 0 |
| `%r27` | `%dp` | Datenzeiger |
| `%r28` | `%ret0` | Integer Rückgaberegister 0 |
| `%r29` | `%ret1` | Integer Rückgaberegister 1 |
| `%r30` | `%sp` | Stapelzeiger |
| `%r31` | `%mrp` | Rücksprungadresse bei Millicode Funktionsaufrufen |
| `%sr1` | `%sret` | Space Register Rückgaberegister |
| | `%sarg` | Space Register Argumentübergaberegister |
| `%fr4` | `%fret` | Fließkomma Rückgaberegister |
| | `%farg0` | Fließkomma Argumentübergaberegister 0 |
| `%fr5` | `%farg1` | Fließkomma Argumentübergaberegister 1 |
| `%fr6` | `%farg2` | Fließkomma Argumentübergaberegister 2 |
| `%fr7` | `%farg3` | Fließkomma Argumentübergaberegister 3 |

Abbildung 8: *Verteilung von Spezialaufgaben an Register unter PA-RISC*

Integern stehen 32 32-Bit Register zur Verfügung. Der Prozessor verwaltet weiter 32 64-Bit

[5] PA steht für „Precision Architecture“.

[6] RISC ist die Abkürzung für „Reduced Instruction Set Computer“.

Fließkommaregister, 8 32-Bit Space Register und 32 32-Bit Kontrollregister für Systemstatusinformationen. Dabei lassen sich die Fließkommaregister teilen, so daß 64 Register zu je 32 Bit benutzt werden können. Die oben erwähnten Space Register helfen dem eigentlichen 32-Bit Prozessor, je nach Prozessortyp, virtuelle 48-Bit, 56-Bit oder sogar 64-Bit Adreßräume anzusprechen.

Der jüngste Sproß der sieben Mitglieder zählenden PA-RISC Familie ist der superskalare HP7100 Prozessor. Seine maximale Taktfrequenz beträgt momentan 125 Megahertz. Das direct mapped Daten- und Instruktionscaching geschieht außerhalb des Prozessors. Die Größe des Datencaches kann dabei, je nach Implementierung, zwischen vier Kilobyte und zwei Megabyte liegen, der Instruktionscache zwischen vier Kilobyte und einem Megabyte. Der Datencache ist so aufgebaut, daß die CPU nach einem cache-miss solange weiterarbeiten kann, bis das Zielregister als Quellregister für eine andere Operation benötigt wird [Snu93].

Nach dieser allgemeinen Einführung sollen speziellere Eigenschaften der PA-RISC Architektur untersucht werden.

### 5.1.1 „caller-saves“ und „callee-saves“ Register

Werden Konventionen für Prozeduraufrufe aufgestellt, unterscheidet man meist zwischen „caller-saves“ und „callee-saves“ Registern. Benutzt eine Prozedur callee-saves Register, so ist sie verpflichtet, diese unmittelbar nach ihrem Aufruf zu sichern und vor dem Rücksprung zum Aufrufer zu restaurieren. Da jede weitere aufgerufene Prozedur so vorgeht, ist die Datensicherheit dieser Register gewährleistet. Sollen caller-saves Register einen Unterprogrammaufruf unverändert überdauern, so müssen diese vor dem Aufruf der Prozedur gesichert und nach der Rückkehr restauriert werden. In caller-saves Registern werden also prinzipiell temporäre Werte gehalten. Abbildung 9 zeigt die Aufteilung der noch nicht für spezielle Aufgaben reservierten Prozessorregister in caller-saves und callee-saves Register [HPC91].

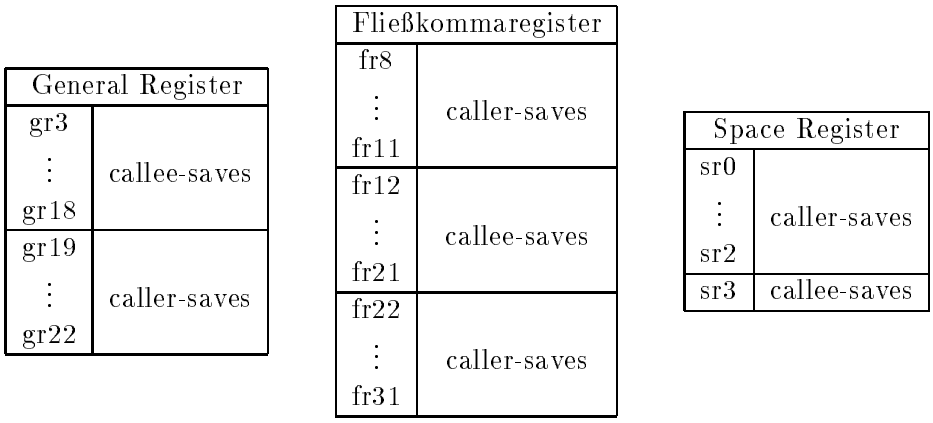



Abbildung 9: *Die Aufteilung der Register in caller-saves und callee-saves*

### 5.1.2 Organisation des Stapels

Man unterscheidet in der PA-RISC Architektur sogenannte „leaf“ und „non-leaf“ Prozeduren. Bei „non-leaf“ Prozeduren handelt es sich um Prozeduren die weitere Prozeduraufrufe tätigen. „leaf“ Prozeduren erzeugen keine weiteren Unterprogrammaufrufe.

Jede „non-leaf“ Prozedur erzeugt auf dem Stapel einen neuen Stapelbereich (Stack Frame). Dagegen benötigen „leaf“ Prozeduren nicht in jedem Fall einen Stack Frame. Anfang und Ende eines solchen Stack Frames müssen nach PA-RISC Konvention einem 64 Byte Alignment unterliegen. Dieses strikte Alignment wurde aus Effizienzgründen gewählt, um damit das Zusammenspiel mit dem Cache, dessen cache-lines maximal 64 Byte lang sind [HPC91], zu optimieren.

Jeder Stack Frame (Abbildung 10) besteht aus einem fest durch PA-RISC vorgegebenen Basisteil und einem optionalen Teil. Dieser Basisteil besteht aus dem sogenannten Frame Marker und dem Bereich für Fixed Arguments. Im Frame Marker können Verwaltungsdaten über Prozeduraufrufe hinweg gespeichert werden. Hier wird, falls erforderlich, auch die Rücksprungadresse abgelegt. Die vier 32-Bit Werte im Fixed Arguments Feld werden nicht für die Argumentübergabe genutzt, sondern dienen der aufgerufenen Prozedur zum eventuell nötigen Sichern der ersten vier Argumente. Die Übergabe dieser Argumente geschieht in den vier Argumentregistern. In dem optionalen Teil können weitere Argumente für eine aufzurufende Prozedur abgelegt werden. Dabei legt der Aufrufer diese weiteren Argumente auf seinem Stapel ab. Der Aufgerufene muß sie von dort herunterlesen. Weiter läßt sich hier Platz für die Sicherung der callee-saves Register bereitstellen.

Der Stapel wächst unter PA-RISC von niedrigen Adressen zu höheren.

### 5.1.3 Dynamisch bindbare Zeiger (Procedure Labels)

Um dynamisches Linken zur Laufzeit zu ermöglichen, setzen PA-RISC Compiler spezielle Zeiger, sogenannte „Procedure Labels“, ein [HPC91]. Dynamisches Linken ist immer dann erforderlich, wenn zur Übersetzungszeit das Ziel eines Prozeduraufrufs noch nicht bekannt ist oder es sich während der Laufzeit dynamisch ändern kann. Dies tritt zum Beispiel bei der Verwendung von Shared Libraries auf. Das Ziel des Prozeduraufrufs muß dann zur Laufzeit eingesetzt werden. Da sich selbstmodifizierender Code schon wegen der read-only Einlagerung verbietet, verwendet man eine doppelte Verzeigerung über einen Zeiger im BSS-Segment auf die Prozedur. Um aber nicht zwischen dynamischen und festen Prozeduraufrufen unterscheiden zu müssen, ermöglichen Procedure Labels eine transparente Verwendung. Dies wird durch eine geeignete Codierung der Adresse erreicht. Da unter PA-RISC Zeiger auf Maschineninstruktionen immer einem 4 Byte Alignment unterliegen, stehen die beiden untersten Bits für eine solche Codierung frei. Ein dynamischer Sprungbefehl muß die Codierung auswerten und vor dem konkreten Sprung diese Bits löschen, um das korrekte Alignment wiederherzustellen. Abbildung 11 zeigt ein Procedure Label. Ein gesetztes L-Bit signalisiert, daß die durch die 30 höherwertigen Bits repräsentierte, Referenz auf einen Eintrag verweist, in dem das eigentliche Sprungziel vermerkt ist. Ist das Bit zurückgesetzt, so handelt es sich um eine direkte Einsprungadresse. Das X-Bit charakterisiert die Adresse als lokal oder extern. Für externe Prozeduraufrufe dient das Space Register

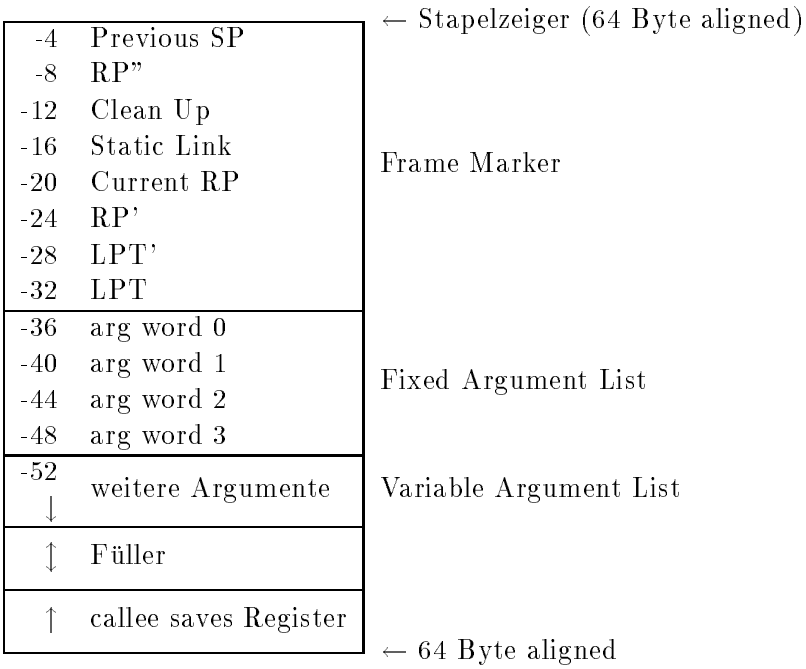



Abbildung 10: *Aufbau eines Stack Frames unter PA-RISC*

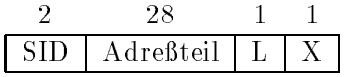



Abbildung 11: *Aufbau eines Procedure Labels*

mit der Nummer SID+4 als Basis. Innerhalb lokaler Prozeduraufrufe ist der SID-Eintrag ein Teil der Adresse. Auf externe Prozeduraufrufe soll aber hier nicht näher eingegangen werden.

### 5.1.4 Millicode Funktionen

CISC[7] Prozessoren enthalten in ihrem Befehlssatz zum Teil hochkomplexe Instruktionen. RISC Prozessoren beschränken sich dagegen auf unbedingt notwendige Grundoperationen, die direkt durch spezielle Schaltkreise der Hardware realisiert sind. Da aber trotzdem komplexe Instruktionen benötigt werden, stehen RISC Compiler vor einem Platz/Zeit Problem. Werden Anweisungssequenzen die komplexe Instruktionen nachbilden, direkt ohne Prozeduraufruf in den Code aufgenommen (inlining), so wächst die Länge des erzeugten Programmcodes sehr stark an. Andererseits führen Bibliotheksaufrufe zu einem Effizienzeinbruch,

[7]CISC ist die Abkürzung für „Complex Instruction Set Computer“.

der den Geschwindigkeitsvorteil von RISC Prozessoren zunichte machen würde. PA-RISC bietet daher die Möglichkeit der Verwendung spezieller Prozeduren mit weniger strengen Prozeduraufrufkonventionen. Aufrufe solcher Prozeduren werden von Hewlett-Packard in Anlehnung an die Microprogrammierung komplexer Instruktionen auf CISC Prozessoren als „Millicode Calls“ bezeichnet [HPC91]. Effizienzvorteile gegenüber herkömmlichen Prozeduraufrufen unter PA-RISC entstehen vor allem dadurch, daß eine Millicode Funktion nicht durch Konventionen gezwungen ist, einen Stack Frame anzulegen. Dadurch ist eine Prozedur die nur Millicode Aufrufe tätigt eine leaf Prozedur und benötigt ihrerseits ebenfalls keinen Stack Frame. Hieraus folgt allerdings, daß diese Prozedur keine Möglichkeit hat das Rücksprungadressenregister zu sichern. Um die Rücksprungadresse dieser Prozedur beim Aufruf einer Millicode Funktion trotzdem nicht zu zerstören, existiert ein eigenes Register für die Rücksprungadresse aus Millicode Aufrufen. Weiter ist die strikte Festlegung der vier Argumentregister aufgehoben. Jedes, nicht schon für andere Aufgaben belegtes Register, kann als Übergaberegister fungieren. Die Limitierung auf vier Argumentregister ist ebenfalls aufgehoben.

Eine wichtige Millicode Funktion ist `$$dyncall`. `$$dyncall` bildet eine Befehlssequenz für einen Unterprogrammaufruf mit indirekter Adreßangabe [HPC91]. Von einer indirekten Adreßangabe spricht man immer dann, wenn die Adresse nicht in irgendeiner Form im Befehl selbst codiert ist, sondern aus einem Prozessorregister entnommen wird. Das Ziel eines solchen Unterprogrammaufrufs ist damit dynamisch zur Laufzeit festlegbar. Dabei ist unter PA-RISC zu beachten, daß eine Einsprungadresse immer ein Procedure Label (siehe 5.1.3) darstellt, aus dem die eigentliche Adresse errechnet werden muß. Sieht man von der Decodierung des Procedure Labels ab, vollzieht sich der Unterprogrammaufruf in zwei Schritten. Dazu nutzen zwei Sprungbefehle aus dem Befehlssatz des Prozessors ihre unterschiedliche Funktionalität, um gemeinsam die oben beschriebene Befehlssemantik zu simulieren. Die „Branch and Link“ Instruktion speichert die Rücksprungadresse und ruft pc-relativ `$$dyncall` auf. In `$$dyncall` veranlaßt die „Branch Vectored“ Operation einen indirekten Sprung zu der aus dem Procedure Label gewonnenen Adresse. Die aufgerufene Funktion kehrt über die von „Branch and Link“ in einem Register hinterlassene Rücksprungadresse zum Aufrufer zurück. Wegen der notwendigen Zweistufigkeit des Aufrufs kann die Millicode Funktion `$$dyncall` nicht mittels inlining in den Code eingebunden werden.

### 5.1.5 Rückverfolgung des Aktivitätsträgerpfades (Backtracing)

Das Backtracing ([HPC91]) erfolgt entlang der aufgebauten Stack Frames. Um Effizienzeinbußen während der Laufzeit weitgehend zu vermeiden, hat ein Compiler unter PA-RISC die Aufgabe, Informationen über jede einzelne Funktion sowie über deren Stapelbereich abzulegen. Diese Informationen umfassen die Start- und Endadresse der Funktion und die Größe des von der Funktion angelegten Stapelbereichs. Weiter werden Angaben darüber gemacht, ob die Rücksprungadresse und der Wert des Stapelzeigers, den dieser vor dem Aufruf der Funktion hatte, gesichert wurden. Über die abgelegte Rücksprungadresse kann so ein Stapelbereich einer Funktion zugeordnet werden. Das Rückwärtsdurchlaufen der Stapelbereiche liefert damit einen kompletten Aufrufpfad.

Werden nach dem Aufruf einer Funktion die Argumentübergaberegister in den Fixed

Arguments Bereich gesichert, so ist es während des Backtracings möglich, diese korrekt wiederzugeben. Weiter besteht durch Ablegen des Stapelzeigers zur Laufzeit die Möglichkeit, auch Aktivitätsträger mit nicht kontinuierlichem Stapel zurückzuverfolgen.

## 5.2 Anpaßung an die PA-RISC Architektur

Während der Portierung auf die PA-RISC Architektur wurden diverse Fehler innerhalb des Pakets entdeckt. Sie sind hauptsächlich im Zusammenhang mit dem Umstand zu sehen, daß unter PA-RISC der Stapel aufwärts wächst. Im QuickThreads Paket waren auch für diesen Fall Vorkehrungen getroffen. Da aber die hier vorliegende Portierung bislang die einzige ist die diesen Code nutzt, war dieser aufgrund von unzureichenden Testmöglichkeiten sehr fehlerhaft. Auf diese Fehler soll im weiteren nicht eingegangen werden. Die nötigen Korrekturen wurden dem Autor bekannt gemacht. Sie sind, ebenso wie die eigentliche Portierung auf PA-RISC, Teil der zweiten QuickThreads Veröffentlichung und frei über `ftp.cs.washington.edu` (`/pub/qt-002.tar.gz`) erhältlich.

Das QuickThreads Paket besteht zum einen aus dem eigentlichen QuickThreads Kerncode, der zur Bibliothek `libqt.a` gebunden wird. Hierfür sind die architekturunabhängigen Dateien `qt.c` und `qt.h`, sowie die architekturabhängigen Dateien `md/`*arch*`.s` und `md/`*arch*`.h` relevant.

Ein weiterer Teil des Pakets ist ein, auf Basis der QuickThreads erzeugtes, Thread Paket (Simple Threads). Die Dateien `stp.c` und `stp.h` werden dazu compiliert und mit `libqt.a` gelinkt. Es entsteht die Bibliothek `libstp.a`.

Für die Grundoperationen der QuickThreads sowie für die der Simple Threads können Zeitmessungen durchgeführt werden. Für diesen Zweck wird das Programm `run` erzeugt. Zusammen mit dem shell-script `time/raw` ermöglicht es einen automatischen Test- und Meßlauf. In dem Verzeichnis `tmp/` finden sich Programmläufe der einzelnen Portierungen auf der jeweiligen Architektur.

Zum Erzeugen des gesamten QuickThreads Pakets wird mittels `./config` *machtype* eine architekturabhängige Quelldateibasis geschaffen. Für PA-RISC ist *machtype* durch `hppa` beziehungsweise `hppa-cnx-spp` zu ersetzen. `hppa-cnx-spp` unterscheidet sich lediglich durch die Wahl des Compilers von der `hppa` Standardumgebung. Der Aufruf des Kommandos `make` erzeugt das Paket.

### 5.2.1 Programmiertechnischer Aufbau des QuickThreads Pakets

Dem Benutzer der QuickThreads Bibliothek stehen die fünf Basisoperationen `QT_ARGS()`, `QT_VARGS()`, `QT_BLOCKI()`, `QT_BLOCK()` und `QT_ABORT()` zur Verfügung. Genau betrachtet sind dies Makros, hinter denen sich Funktionen verbergen. Diese Funktionen haben den gleichen Namen wie die korrespondierenden Makros, werden allerdings klein geschrieben. Das Makro `QT_BLOCKI()` wird also beispielsweise auf die Assemblerfunktion `qt_blocki()` abgebildet. Einzig und allein das Makro `QT_ARGS()` ist ein „echtes“ Makro, das in den fortlaufenden Instruktionsfluß ohne Unterprogrammaufruf eingebunden wird.

### 5.2.2 qt_blocki (md/hppa.s) — Kontextwechsel mit Sicherung der callee-saves General Register und des callee-saves Space Registers

Den Kern jedes Thread Pakets stellt die Routine zum Wechseln des Kontextes dar. Die Notwendigkeit in der Routine auf genau definierte Prozessorregister und als Spezialfall den Stapelzeiger zugreifen zu müssen, zwingt hier zur Assemblerprogrammierung. Die Portierung auf PA-RISC Assembler soll im folgenden analysiert werden.

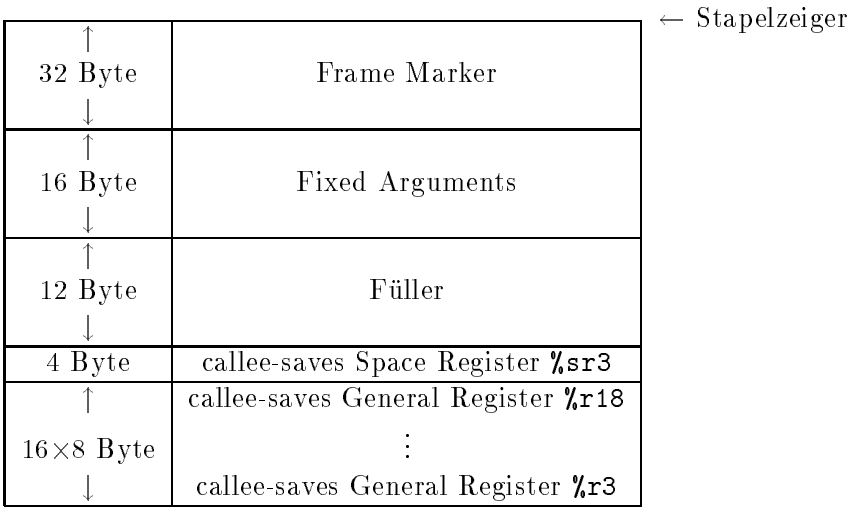



Abbildung 12: *Struktur eines durch* `qt_blocki` *aufgebauten Stack Frames*

In einem ersten Schritt muß der Umfang des zu sichernden Kontextes bestimmt werden. Ruft ein User-Level Thread aus der Applikation heraus eine der `block()` Funktionen auf, so sichert dieser beim Aufruf die von ihm verwendeten caller-saves Register. In der aufgerufenen `block()` Funktion sind nur noch die callee-saves Register ungesichert. Da die `block()` Funktion aber keine Aussage darüber machen kann, welche Register im folgenden verändert werden, muß sie alle callee-saves Register sichern. Der Benutzer hat allerdings die Möglichkeit, durch die Wahl zwischen `QT_BLOCKI()` und `QT_BLOCK()`, der Bibliothek begrenzte Hilfestellung zu geben.

Neben den in der QuickThreads Spezifikation vorgesehenen Integer Registern, sichert die PA-RISC Portierung von `QT_BLOCKI()` auch das callee-saves Space Register `%sr3`. Das QuickThreads Paket unterscheidet nur zwischen Registern für Fließkommazahlen und solchen für ganzzahlige Werte.

`QT_BLOCKI()` erzeugt einen Stapelbereich, wie er in Abbildung 12 dargestellt ist.

```
1 qt_blocki
2       .PROC
3       .CALLINFO   CALLER, ENTRY_GR=18, ENTRY_SR=3, FRAME=8, SAVE_RP
4       .ENTRY
```

Diese drei Assemblerdirektiven[8] ([HPB91]) leiten die `qt_blocki` Prozedur ein. Sie dienen im wesentlichen dazu, das in 5.1.5 beschriebene Backtracing zu unterstützen. Dem Assembler fällt es zu, Verwaltungsstrukturen dafür anzulegen. Die Direktive `.ENTRY` markiert die Startadresse der Routine, `.EXIT` deren Endadresse. Interessanter ist die Direktive `.CALLINFO`. Ihre Argumente beschreiben die Prozedur im Bezug auf die Stapelbenutzung. Neben der Start- und Endadresse einer Prozedur ist auch die Größe des Stack Frames von essentieller Bedeutung. Das Argument `CALLER` erklärt dem Assembler, daß es sich um eine non-leaf Prozedur handelt. non-leaf Prozeduren legen zumindest den Basisstapelbereich an. Es handelt sich dabei um den Frame Marker und den Fixed Arguments Bereich (vgl. 5.1.2). `ENTRY_GR=18` und `ENTRY_SR=3` teilen dem Assembler weiter mit, daß in dem Stapelbereich der Prozedur auch die callee-saves Register (vgl. 5.1.1) `%r3` bis `%r18` sowie `%sr3` abgelegt werden sollen. Sofern noch weiterer Speicher auf dem Stapel erforderlich ist, wird dies mit `FRAME=`*size* erklärt. Solch zusätzlicher Speicher kann für die Übergabe von mehr als vier Argumenten oder wie hier zum Erzwingen des korrekten Alignments genutzt werden.

```
 5       stw         %rp,-20(%sp)
```

Der Prozessorbefehl `stw` (store word) legt ein General Register im Speicher ab. Hier wird die Rücksprungadresse (`%rp`) relativ zum aktuellen Stapelzeiger (`%sp`) abgelegt. Die Sicherung erfolgt in den Frame Marker des Aufrufers (vgl. 5.1.2 und Abbildung 10).

```
 6       stwm        %r3,128(%sp)
 7       stw         %r4,-124(%sp)
 8       stw         %r5,-120(%sp)
 9       stw         %r6,-116(%sp)
10       stw         %r7,-112(%sp)
11       stw         %r8,-108(%sp)
12       stw         %r9,-104(%sp)
13       stw         %r10,-100(%sp)
14       stw         %r11,-96(%sp)
15       stw         %r12,-92(%sp)
16       stw         %r13,-88(%sp)
17       stw         %r14,-84(%sp)
18       stw         %r15,-80(%sp)
19       stw         %r16,-76(%sp)
20       stw         %r17,-72(%sp)
21       stw         %r18,-68(%sp)
```

In diesen 16 Zeilen werden alle callee-saves General Register gesichert. Der Prozessorbefehl `stwm` (store word and modify) legt das Prozessorregister `%r3` auf dem Stapel ab und addiert 128 Byte auf den Stapelzeiger auf. Diese 128 Byte bilden den neuen Stack Frame. Daraufhin werden alle weiteren General Register in aufsteigender Reihenfolge relativ zum Stapelzeiger abgelegt.

[8] Assemblerdirektiven sind keine Befehle aus dem Befehlssatz des Prozessors. Sie stellen Anweisungen an den Assembler dar.

Die Größe des Stapelbereichs ergibt sich aus 32 Byte für den Frame Marker, 16 Byte für die Fixed Arguments List, 16×4 Byte für die callee-saves General Register und 1×4 Byte für das callee-saves Space Register. Die Summe dieser Einzelbereiche ist also 116 Byte. Da der Stapel unter PA-RISC auf 64 Byte Grenzen gerundet werden muß, ergeben sich 128 Byte für die Gesamtlänge.

```
22      mfsp        %sr3,%r3
23      stw         %r3,-64(%sp)
```

Für die Ablage eines Space Registers im Speicher gibt es keinen eigenen Assemblerbefehl. Daher muß das Space Register vorher in ein General Register transferiert werden. Diese Aufgabe wird durch den Befehl `mfsp` (move from space register) erledigt. Danach erfolgt die Speicherung wie gewohnt.

Der nun folgende Abschnitt (Zeile 24–30) hat die Aufgabe die `helper()` Funktion aufzurufen. Im Argumentregister `%arg0` wurde `qt_blocki` ein Funktionszeiger auf die, vom aufsetzenden Thread Paket implementierte, `helper()` Funktion übergeben. In den Registern `%arg1` und `%arg2` sind Argumente für die `helper()` Funktion eingelagert. `%arg3` beherbergt den Zeiger auf den Stapel des neuen Threads.

```
24      copy        %arg0,%r22
25      copy        %sp,%arg0
26      copy        %arg3,%sp
27
28      .CALL
29      bl          $$dyncall,%mrp
30      copy        %mrp,%rp
```

Zum Aufruf der `helper()` Funktion wird, wie in 5.1.4 beschrieben, eine Millicode Funktion namens `$$dyncall` eingesetzt. `$$dyncall` erwartet als Argument ein Procedure Label (vgl. 5.1.3) in Register `%r22`. Die durch dieses Procedure Label spezifizierte Funktion wird von `$$dyncall` aufgerufen. Vor dem Aufruf der `helper()` Funktion muß allerdings noch das erste Argument gesetzt werden und der Stapelzeiger auf den Stapel des zu deblockierenden Threads gelenkt werden. Das erste Argument für die `helper()` Funktion ist der alte Stapelzeiger. Es ist die Aufgabe der Funktion, diesen in eine thread-private Datenstruktur zu sichern. Durch den `copy`[9] Befehl wird der Inhalt des `%sp` Registers in das `%arg0` Register kopiert (Zeile 25). Die beiden weiteren Argumente der `helper()` Funktion werden unverändert durchgereicht. In Zeile 26 wird das Stapelzeigerregister mit einem Zeiger auf den Stapel des zu deblockierenden Thread geladen. Damit ist die Umschaltung auf den neuen Stapel bereits vollzogen, so daß der folgende Prozeduraufruf schon vollständig auf dem neuen Stapel abläuft.

`bl` (branch and link) springt daraufhin `$$dyncall` an. Der in Zeile 30 folgende `copy` Befehl ist während der Ausführung des `bl` Befehls bereits in der Pipeline und wird daher

[9] Im eigentlichen Sinne ist „`copy` *quelle*,*ziel*“ kein Prozessorbefehl, sondern wird bei der Assemblierung textuell durch den Prozessorbefehl „`or` *quelle*`,0,`*ziel*“ ersetzt.

noch vor `$$dyncall` ausgeführt. Seine Aufgabe ist es die Rücksprungadresse, die in das `%mrp` (millicode return pointer) Register geschrieben wurde, in das `%rp` Register zu kopieren. Dies ist nötig, da die `helper()` Funktion als normale Prozedur über das `%rp` Register zurückspringt.

Der Thread, der `QT_BLOCK()` aufgerufen hat, ist nun gänzlich blockiert, so daß der Kontext des Nachfolgers restauriert werden kann. Die Restaurierung verläuft in umgekehrter Reihenfolge wie die Sicherung des Kontextes.

```
31      ldw         -64(%sp),%r3
32      mtsp        %r3,%sr3
```

Die Zeilen 31 und 32 stellen das callee-saves Space Register wieder her. Dazu wird mit dem Befehl `ldw` (load word) relativ zum neuen Stapelzeiger gelesen. Der gesicherte Inhalt des Space Registers wird in ein General Register geladen und von dort aus mit `mtsp` (move to space register) in das Space Register `%sr3` zurückkopiert.

```
33      ldw         -68(%sp),%r18
34      ldw         -72(%sp),%r17
35      ldw         -76(%sp),%r16
36      ldw         -80(%sp),%r15
37      ldw         -84(%sp),%r14
38      ldw         -88(%sp),%r13
39      ldw         -92(%sp),%r12
40      ldw         -96(%sp),%r11
41      ldw         -100(%sp),%r10
42      ldw         -104(%sp),%r9
43      ldw         -108(%sp),%r8
44      ldw         -112(%sp),%r7
45      ldw         -116(%sp),%r6
46      ldw         -120(%sp),%r5
47      ldw         -124(%sp),%r4
48
49      ldw         -148(%sp),%rp
50
51      bv          %r0(%rp)
52      ldwm        -128(%sp),%r3
53
54      .EXIT
55      .PROCEND
```

Daran schließt in den Zeilen 33 bis 47 und 52 die Restaurierung der General Register an. Die Wiederherstellung des `%rp` Registers ist hier vorgezogen, da die Rücksprungadresse vor dem `bv` (branch vectored) Befehl bekannt sein muß. Der Umstand, daß bei einem Sprung der nachfolgende Befehl auch noch in die Pipeline einfließt, wird zur Restaurierung des letzten Registers und zur Korrektur des Stapelzeigers genutzt (Zeile 52).

### 5.2.3 `qt_block` (`md/hppa.s`) — Sicherung der callee-saves Floatingpoint Register und Aufruf von `qt_blocki`

Arbeitet ein Thread auf Fließkommaregistern, so müssen diese bei einem Kontextwechsel ebenfalls gesichert werden. Dazu dient die QuickThreads Funktion `QT_BLOCK()`. Diese ist direkt mit der Assemblerroutine `qt_block` assoziiert. `qt_block` ruft nach der Speicherung der callee-saves Fließkommaregister `qt_blocki` als Unterroutine auf.

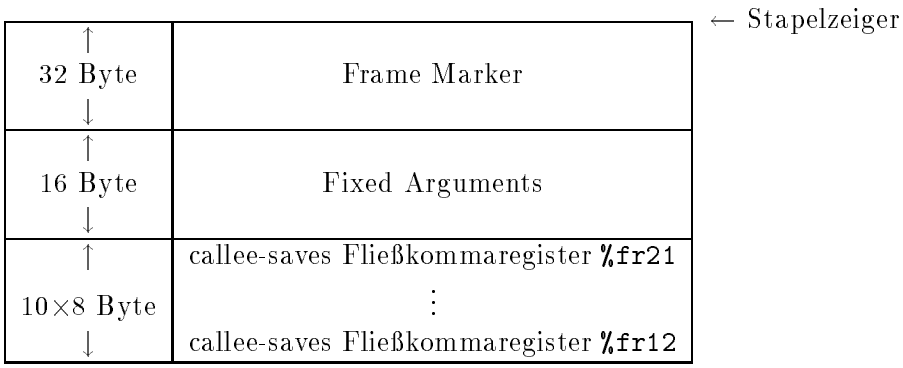



Abbildung 13: *Struktur eines durch* `qt_block` *aufgebauten Stack Frames*

Für den Stack Frame dieser Prozedur sind, wie in Abbildung 13 dargestellt, insgesamt 128 Byte vorgesehen. Diese 128 Byte setzen sich aus 32 Byte Frame Marker, 16 Byte Fixed Arguments sowie 10×8 Byte zur Speicherung der callee-saves Fließkommaregister zusammen.

```
56 qt_block
57       .PROC
58       .CALLINFO   CALLER, FRAME=0, SAVE_RP, ENTRY_FR=21
59       .ENTRY
60
61       stw         %rp,-20(%sp)
62
63       fstds,ma    %fr12,8(%sp)
64       fstds,ma    %fr13,8(%sp)
65       fstds,ma    %fr14,8(%sp)
66       fstds,ma    %fr15,8(%sp)
67       fstds,ma    %fr16,8(%sp)
68       fstds,ma    %fr17,8(%sp)
69       fstds,ma    %fr18,8(%sp)
70       fstds,ma    %fr19,8(%sp)
71       fstds,ma    %fr20,8(%sp)
72       fstds,ma    %fr21,8(%sp)
```

Mit dem `fstds,ma` (floating-point store doubleword short and modify after) Befehl werden die Fließkommaregister auf dem Stapel abgelegt. Der Stapelzeiger wird nach jeder Speicherung um acht Byte angehoben.

```
73      .CALL
74      bl          qt_blocki,%rp
75      ldo         48(%sp),%sp
```

Obige Befehlssequenz leitet den Sprung zu `qt_blocki` ein. Vor dem eigentlichen Eintritt in `qt_blocki` wird der Inhalt des Stapelzeigers um die Länge des Frame Markers und der Fixed Arguments erhöht. Die Argumente werden unangetastet von `qt_block` an `qt_blocki` weitergereicht.

```
76      ldo         -48(%sp),%sp
77
78      fldds,mb    -8(%sp),%fr21
79      fldds,mb    -8(%sp),%fr20
80      fldds,mb    -8(%sp),%fr19
81      fldds,mb    -8(%sp),%fr18
82      fldds,mb    -8(%sp),%fr17
83      fldds,mb    -8(%sp),%fr16
84      fldds,mb    -8(%sp),%fr15
85      fldds,mb    -8(%sp),%fr14
86      fldds,mb    -8(%sp),%fr13
87
88      ldw         -28(%sp),%rp
89
90      bv          %r0(%rp)
91      fldds,mb    -8(%sp),%fr12
92
93      .EXIT
94      .PROCEND
```

Nach einer Deblockierung wird der Stapel in gewohnter Weise wieder aufgelöst. Zur Restaurierung der Fließkommaregister dient der `fldds,mb` (floating-point load doubleword short and modify before) Befehl.

### 5.2.4 `qt_abort` (`md/hppa.s`) — Verwerfen eines QuickThreads

Die Routine zum Verwerfen eines Threads ist identisch zur `qt_blocki` Prozedur. Lediglich die Einsprungsadresse liegt nun direkt vor dem Aufruf der `helper()` Funktion. Der Teil in dem der Kontext des blockierten Threads gesichert wird, ist in `qt_abort` also ausgeklammert.

Daher wäre es prinzipiell möglich, den selben Code für `qt_blocki` und `qt_abort` zu verwenden. Diese Idee ist in bisher allen Implementierungen verwirklicht. Der Backtracing

Mechanismus von PA-RISC gibt als Resultat dieser Vorgehensweise anstatt `qt_blocki` aber immer `qt_abort` als aufgerufene Funktion aus. Deshalb existiert in der PA-RISC Portierung hier eigenständiger Code für `qt_abort`.

### 5.2.5 Stapelaufbau durch `QT_ARGS()`

`QT_ARGS()` hat die Aufgabe einen uninitialisierten Thread durch Anlegen eines geeigneten Stapelbereichs zu initialisieren. Diese Initialisierung täuscht eine vorangegangene Blockierung mit der Rücksprungadresse `qt_start` vor. Die Argumente `only`, `userf`, `t` und `u` (vgl. 4.3) werden so auf dem Stapel abgelegt, daß sie sich nach einer Restaurierung durch `qt_blocki` in den Registern `%r15` bis `%r18` befinden. Der Rücksprung in den vermeintlich wieder deblockierten Thread geschieht durch den Prozessorbefehl `bv` am Ende von `qt_blocki`. Der Funktionszeiger der `QT_ARGS()` übergeben wurde, kann ein Procedure Label sein, das durch den Prozessorbefehl nicht ausgewertet wird. Wie in Abschnitt 5.1.3 erläutert, ist der Adreßteil eines Procedure Labels dessen L-Bit gesetzt ist, nicht ein Zeiger auf eine Adresse im Text Segment, sondern vielmehr ein Zeiger auf einen Zeiger in das Text Segment. Ist das L-Bit aktiv, so muß daher über das Procedure Label erst auf den eigentlichen Zeiger zugegriffen werden. Es ist also sicherzustellen, daß die eigentliche, uncodierte Startadresse auf dem Stapel abgelegt wird. Zur Erledigung dieser Aufgabe wurde in `md/hppa.h` das Makro `QT_PA_RISC_READ_PLABEL()` definiert.

| | | | |
|---|---|---|---|
| | | | ← `%sp` |
| Stack Frame 1 | 32 Byte | Frame Marker | |
| | 16 Byte | Fixed Arguments | |
| | 12 Byte | Füller | |
| | 4 Byte | callee-saves Space Register `%sr3` | |
| | 16×8 Byte | callee-saves General Register `%gr18` | `u` |
| | | callee-saves General Register `%gr17` | `t` |
| | | callee-saves General Register `%gr16` | `userf` |
| | | callee-saves General Register `%gr15` | `only` |
| | | ⋮ | |
| | | callee-saves General Register `%gr3` | |
| Stack Frame 0 | 32 Byte | Frame Marker: Previous SP | |
| | | ⋮ | |
| | | Frame Marker: Current RP | `qt_start` |
| | | ⋮ | |
| | | Frame Marker: LPT | |
| | 16 Byte | Fixed Arguments | |
| | 16 Byte | Füller | |

Abbildung 14: *Struktur eines durch* `QT_ARGS()` *aufgebauten Stack Frames*

Wie Abbildung 14 zeigt, werden auf dem Stapel zwei Stack Frames erzeugt. Stack Frame 1 stellt den Teil dar, den eine fiktive `QT_BLOCKI()` Funktion, aufgerufen vom Inhaber

des Stack Frames 0 hinterließ. Bei der Deblockierung wird Stack Frame 1 abgetragen. Dagegen bleibt Stack Frame 0 erhalten und dient der Funktion `qt_start` als Stapelbereich.

### 5.2.6 `qt_start` (`md/hppa.s`) — Startfunktion für Threads mit fester Argumentenzahl

Bei der Initialisierung eines Threads durch `QT_ARGS()` ist dessen Stapel so beschrieben worden, als ob sich der Thread unmittelbar vor dem Label `qt_start` blockiert hätte. Die Argumente für diesen Thread wurden dabei auf dem Stapel so abgelegt, daß diese nach einer Deblockierung durch `qt_blocki` in den Registern `%r15` bis `%r18` vorgefunden werden.

```
 95 qt_start
 96        .PROC
 97        .CALLINFO   CALLER, FRAME=16
 98        .ENTRY
 99
100        copy        %r18,%arg0
101        copy        %r17,%arg1
102        copy        %r16,%arg2
103
104        copy        %r15,%r22
```

Die durch `qt_blocki` restaurierten Argumente werden durch `qt_start` in die Argumentübergaberegister kopiert beziehungsweise im Fall des Registers `%r15` als Funktionszeiger auf `only()` interpretiert. In `%arg0` wird dabei das Argument für die Threadfunktion, in `%arg1` das Argument für die Umgebungsverwaltung und in `%arg2` ein Funktionszeiger auf die Threadfunktion übergeben.

```
105         .CALL
106         bl          $$dyncall,%mrp
107         copy        %mrp,%rp
108
109         bl,n        qt_error,%r0
110
111         .EXIT
112         .PROCEND
```

Die durch den Zeiger in Register `%r22` charakterisierte `only()` Funktion wird wie gehabt aufgerufen. Kehrt diese Funktion verbotenerweise zurück, so wird die Fehlerbehandlungsfunktion `qt_error()` involviert.

### 5.2.7 Stapelaufbau durch `QT_VARGS()`

`QT_VARGS()` geht im Grunde genauso wie `QT_ARGS()` vor. Wieder werden zwei Stack Frames angelegt, von denen der obere bei der Deblockierung durch `qt_blocki` entfernt wird.

← %sp

| | | | |
|---|---|---|---|
| Stack Frame 1 | 32 Byte | Frame Marker | |
| | 16 Byte | Fixed Arguments | |
| | 12 Byte | Füller | |
| | 4 Byte | callee-saves Space Register `%sr3` | |
| | 16×8 Byte | callee-saves General Register `%gr18` | `t` |
| | | callee-saves General Register `%gr17` | `startup` |
| | | callee-saves General Register `%gr16` | `userf` |
| | | callee-saves General Register `%gr15` | `cleanup` |
| | | ⋮ | |
| | | callee-saves General Register `%gr3` | |
| Stack Frame 0 | 32 Byte | Frame Marker: Previous SP | |
| | | ⋮ | |
| | | Frame Marker: Current RP | `qt_vstart` |
| | | ⋮ | |
| | | Frame Marker: LPT | |
| | 16 Byte | Fixed Arguments: arg word 0 | Argument 0 |
| | | Fixed Arguments: arg word 1 | Argument 1 |
| | | Fixed Arguments: arg word 2 | Argument 2 |
| | | Fixed Arguments: arg word 3 | Argument 3 |
| | $(n-3) \times 4$ Byte | Variable Arguments | Argument 4 |
| | | ⋮ | ⋮ |
| | | Variable Arguments | Argument $n$ |
| | ↕ | Füller | |

Abbildung 15: *Struktur eines durch* `QT_VARGS()` *aufgebauten Stack Frames. Ein Argument ist hier jeweils ein 32-Bit Wert. Sollen 64-Bit Werte als Argumente übergeben werden, so bilden zwei aufeinanderfolgende 32-Bit Argumente ein 64-Bit Argument.*

Auch hier wird intern das Makro `QT_PA_RISC_READ_PLABEL()` auf die im Stapel abzulegende Rücksprungadresse angewandt. In diesem Fall ist die einzutragende Rücksprungadresse aber `qt_vstart`. Die Argumente für `userf()` werden im Fixed Arguments Bereich und in dem daran anschließenden optionalen Variable Arguments Bereich in aufsteigender Reihenfolge abgelegt. Nach der Restaurierung finden sich die Argumente `cleanup`, `userf`, `startup` und `t` in den Registern `%r15` bis `%r18` wieder. Die ersten vier Argumente wurden im Fixed Arguments Bereich nur temporär abgelegt und müssen vor dem Aufruf von `userf()` durch `qt_vstart` in die Argumentübergaberegister geladen werden. Abbildung 15 verdeutlicht nochmals den Aufbau des Stapels.

Die Länge des Stack Frame 0 ist dabei nicht fest, sondern hängt vielmehr von der Anzahl der Argumente für `userf()` ab.

### 5.2.8 `qt_vstart` (`md/hppa.s`) — Startfunktion für Threads mit variabler Argumentenzahl

```
113 qt_vstart
```

```
114         .PROC
115         .CALLINFO   CALLER, FRAME=16
116         .ENTRY
117
118         ldo         64(%sp),%sp
119
120         copy        %r15,%arg0
121         copy        %r16,%r22
122         .CALL
123         bl          $$dyncall,%mrp
124         copy        %mrp,%rp
125
126         ldo         -64(%sp),%sp
```

Die Codesequenz in den Zeilen 113 bis 126 dient zum Aufruf der `startup()` Funktion. Das Argument `t` wird dazu aus dem Register `%r15` in das Argumentübergaberegister `%arg0` kopiert. Das Register `%r16` enthält einen Funktionszeiger auf die benutzerdefinierte `startup()` Funktion.

Nach [HPC91] müssen Prozeduren unter PA-RISC im Bedarfsfall den Inhalt der Übergaberegister in den Fixed Arguments Bereich ihres Aufrufers sichern. Unter anderem geschieht dies um während des Backtracings die ersten vier Argumente ausgeben zu können. Durch die `startup()` Funktion würde in diesem Fall das Argument 0 überschrieben und damit für die Threadfunktion `userf()` verloren gehen. Zur Lösung dieses Problems gibt es mehrere Möglichkeiten.

Der hier verwendete Ansatz ist der am einfachsten zu realisierende und zugleich schnellste. Allerdings bleiben im Bezug auf die PA-RISC Konformität Wünsche offen. Die Idee ist, für den Aufruf der `startup()` Funktion einen seperaten Stack Frame anzulegen. Das Verschieben des Stapelzeigers wird durch den `ldo` (load offset) Befehl vorgenommen (Zeile 118 und 126). Nun ist die Speicherung der Übergaberegister durch die `startup()` Funktion unkritisch. Leider erlauben PA-RISC Konventionen während der Ausführung einer Prozedur nicht das Erzeugen eines weiteren Stack Frames. Diese Forderung garantiert, daß ein erfolgreiches Back/-tracing immer zu realisieren ist. Da aber die Stapelgröße von `qt_vstart` wegen der variablen Argumentenzahl zur Übersetzungszeit nicht festgelegt werden kann, ist ein erfolgreiches Backtracing hier sowieso nicht möglich. Aus diesem Grund wurde trotzdem dieser Weg gewählt.

Ein weiterer Ansatz, der aber zu deutlichen Effizienzeinbußen geführt hätte, basiert auf einer zwischengeschalteten Prozedur. Diese würde nach dem Aufbau eines eigenen Stack Frames die `startup()` Funktion aufrufen. Um eine Speicherung der Übergaberegister durch diese Zwischenprozedur auszuschließen, wäre sie in Assembler auszuführen gewesen.

Der dritte Vorschlag benutzt vier freie callee-saves Register als Pufferspeicher. Beim Start von `QT_VARGS()` würden die ersten vier Argumente in diese Register geladen und nach dem Aufruf von `startup()` wieder zurückgeschrieben. Dabei sind an zwei Stellen Optimierungen möglich. Zum einen könnte auf das Laden der Register verzichtet werden, falls die ersten vier Argumente bei der Initialisierung in dem Bereich der callee-saves Register im

Stack Frame 1 (siehe dazu Abbildung 15) untergebracht würden. Nach der Deblockierung befänden sich die Argumente automatisch und ohne weiteren Rechenaufwand in callee-saves Registern. Die zweite Stelle an der optimiert werden kann, ist das Laden der Argumentregister. Da sich die Argumente bereits in Integer Registern befinden, entfallen die vier Speicherzugriffe, die zum Laden der Integer Argumentübergaberegister nötig gewesen wären. Der Aufwand reduziert sich auf ein Kopieren zwischen Registern.

```
127         ldw         -36(%sp),%arg0
128         ldw         -40(%sp),%arg1
129         ldw         -44(%sp),%arg2
130         ldw         -48(%sp),%arg3
131         ldo         -32(%sp),%r22
132         fldws       -4(%r22),%farg0
133         fldds       -8(%r22),%farg1
134         fldws       -12(%r22),%farg2
135         fldds       -16(%r22),%farg3
136         copy        %r17,%r22
137         .CALL
138         bl          $$dyncall,%mrp
139         copy        %mrp,%rp
```

Nachdem durch die `startup()` Funktion eine thread-spezifische Ablaufumgebung geschaffen wurde, kann die eigentliche Threadfunktion aufgerufen werden. Da nicht bekannt ist von welchem Typ die Argumente sind, müssen diese sowohl in die Fließkomma-, als auch in die Integerübergaberegister geladen werden. Abbildung 16 zeigt die drei Arten auf die sich die Argumente interpretieren lassen. In der Befehlssequenz oben, werden dazu die Fließ-

| | arg word 3 | arg word 2 | arg word 1 | arg word 0 |
|---|---|---|---|---|
| 32-Bit Integer | `%arg3` | `%arg2` | `%arg1` | `%arg0` |
| 32-Bit Fließkomma | `%farg3` | `%farg2` | `%farg1` | `%farg0` |
| 64-Bit Fließkomma | `%farg3` | | `%farg1` | |

Abbildung 16: *Ablage unterschiedlicher Typen im Fixed Arguments Bereich*

kommaregister `%farg1` und `%farg3` durch den Befehl `fldds` (floating-point load doubleword short) mit 64-Bit Werten geladen, während die Register `%farg0` und `%farg2` durch `fldws` (floating-point load word short) nur mit 32-Bit Werten besetzt werden. Das Laden der Integerübergaberegister geschieht in trivialer Weise. Die aufgerufene Threadfunktion kennt die Typen ihrer Argumente und ist daher in der Lage, aus der Palette der angebotenen Varianten die Richtige zu wählen.

```
140         copy        %r15,%arg0
141         copy        %ret0,%arg1
```

```
142         copy        %r18,%r22
143         .CALL
144         bl          $$dyncall,%mrp
145         copy        %mrp,%rp
146
147         bl,n        qt_error,%r0
148
149         .EXIT
150         .PROCEND
```

Sobald die Threadfunktion `userf()` zurückgekehrt ist, wird die Funktion `cleanup()` aufgerufen. Ein Funktionszeiger auf `cleanup()` ist in dem callee-saves Register `%r18` abgelegt. Neben `t` ist der Rückgabewert der Funktion `userf()` ein Argument für `cleanup()`.

# 6 Fehlersuche in QuickThreads Applikationen

## 6.1 Spezielle Unterstützung durch den Debugger

Debugger, die eine spezielle Schnittstelle zu den QuickThreads enthalten, sind nicht erhältlich. Um über den Zustand einzelner QuickThreads Aussagen treffen zu können beziehungsweise deren Zustand zu verändern, müßte der Debugger die Verwaltungsstrukturen der Threads in der Bibliothek kennen. Bei der hohen Flexibilität des QuickThreads Pakets wäre dies nur sehr eingeschränkt möglich. Denkbar sind solche Debugging-Schnittstellen demnach erst für aufsetzende Thread Pakete.

Dabei ist, wie in [StSh92] beschrieben, ein dynamisches Binden des Debuggers mit einer auf das aufsetzende Thread Paket zugeschnittenen Debugging-Bibliothek anzustreben.

## 6.2 Rückverfolgung des Aktivitätsträgerpfades

Eine volle Unterstützung der Rückverfolgung des Aktivitätsträgerpfades (Backtracing) wird nicht immer benötigt und verlängert zudem noch die Zeiten für Kontextwechsel und Initialisierung. Falls das Backtracing trotzdem unterstützt werden soll, muß bei der Übersetzung der Bibliothek im Makefile die Option `-DQT_PA_RISC_FULL_BACKTRACING` gesetzt werden.

Das Backtracing orientiert sich, wie in Abschnitt 5.1.5 dieser Arbeit beschrieben, an den auf dem Stapel abgelegten Stack Frames. Da jeder QuickThread einen eigenen Stapel besitzt, hat er auch einen eigenen, von anderen Threads unabhängigen Aktivitätsträgerpfad. Die Rückverfolgung über eine Kontextwechselgrenze ist daher nicht möglich. Die `helper()` Funktion wird auf dem Stapel des zu deblockierenden Threads aufgerufen und gehört damit schon zum neuen Aktivitätsträgerpfad.

Da die variable Argumentenzahl der `qt_vstart` Routine ein Festlegen der Staplegröße zur Übersetzungszeit nicht erlaubt, scheitert hier auch das Backtracing.

Abbildung 17 zeigt eine mit dem Xdb Debugger aus der `helper()` Funktion heraus durchgeführte Rückverfolgung.

```
0 t_splat (old = 0x40006b40, oldp = 0x40006958, null = 00000000)
  [meas.c: 109]
1 qt_blocki + 0x00000070 (0x4000255a, 0x40007978, 0, 0x40004b00)
2 qt_block + 0x00000044 (0x4000255a, 0x40007978, 0, 0x40004b00)
3 test08_aux3 (np = 0x7b0335dc, mep = 0x40007978, nxtp = 0x40004918,
  null = 0x40007ac0) [meas.c: 489]
4 qt_start + 0x00000018 (0, 0, 0, 0)
```

Abbildung 17: *Beispiel für ein in der* `helper()` *Funktion angestoßenes Backtracing.*

Da `qt_start` durch keinen Unterprogrammaufruf gestartet wurde, endet hier die Rückverfolgung des Aktivitätsträgers.

## 6.3 Festlegung der Stapelgröße

Die kritische Größe im Umgang mit User-Level Threads im allgemeinen, ist deren Stapelgröße. Wählt man einen großen Stapel, so führt dies bei einer einigermaßen hohen Anzahl an Einzelthreads schnell zu einem untragbar hohen Gesamtspeicherverbrauch. Entscheidet man sich aufgrund dieser Problematik für kleine Stapel, so läuft man Gefahr über diese hinauszuschreiben. Schon allein der Aufruf der Standard C Funktion `printf()` benötigt, wie Abbildung 18 zeigt, rund vier Kilobyte[10]. Der Verbrauch an Stapelspeicher ist dabei von der Anzahl der Argumente unabhängig.

| aufgerufene C Funktion | HP C Compiler HP715 Workstation | Convex C Compiler Convex SPP |
|---|---|---|
| `printf()` | >= 3947 Byte | >= 4455 Byte |
| `malloc()` | >= 355 Byte | >= 327 Byte |

Abbildung 18: *Benötigter Stapelspeicher bei der Ausführung zweier Bibliotheksfunktionen.*

Fehler die dadurch entstehen, daß der Stapelzeiger über das Ende des Stapels hinauswächst, können kaum nachvollziehbare Fehler an vollständig anderer Stelle verursachen. Dies wird dadurch noch begünstigt, daß die Stapelbereiche der einzelnen Threads meist direkt hintereinander im Speicher liegen. Der Fehler eines Threads wird damit erst beim Deblockieren eines anderen Threads sichtbar oder wirkt sich eventuell nur auf das Ergebnis eines Threads aus, ohne das der Benutzer dies direkt wahrnimmt.

Die Wahl der Stapelgröße sollte also größte Aufmerksamkeit erfahren. Bei der Festsetzung dieses Wertes sind aber nicht nur Gesichtspunkte wie aufgerufene Bibliotheksfunktionen, sondern auch verschachtelte Funktionsaufrufe zu berücksichtigen.

Um dieser Problematik zu begegnen, kann der Versuch unternommen werden, blockierten QuickThreads ihren Stapel zu entziehen. Dafür kann dem momentan laufenden Thread ein in jedem Fall ausreichend großer Stapel zugeteilt werden. Ein sich blockierender Thread hätte nach obiger Vorgehensweise den von ihm benutzten Stapelbereich in ein privates Feld zu sichern. Der benutzte Stapelbereich ist dabei typischerweise sehr viel kleiner als der gesamte Stapel. Bei einer Deblockierung müßten die gesicherten Daten wieder auf den Stapel kopiert werden [Kep93]. Die dadurch anfallende Kopierarbeit ist aber kaum mit einem effizienten Laufzeitverhalten kombinierbar. Werden die Sicherungsbereiche beim Programmstart zu klein gewählt, kann es zudem nötig werden, weiteren Speicher anzufordern.

In [StSh92] wird eine zweistufige Prozeßverwaltung für das Betriebssystem SunOS vorgestellt. Die darin enthaltene User-Level Thread Bibliothek schützt die Applikation vor

[10]Zur Messung dieser Werte wurde ein neuer Stapelbereich generiert und dieser mit einem definierten Muster beschrieben. In einer Assembler Funktion, die als Argumente einen Zeiger auf den neuen Stapelbereich und einen Funktionszeiger entgegennimmt, wurde der Stapelzeiger auf den neuen Stapel gesetzt und die durch den Zeiger spezifizierte Funktion aufgerufen. Nach dem Aufruf der Funktion wurde mittels einer weiteren Assemblerfunktion der aktuelle Stapelzeiger ermittelt. Danach wurde die zu testende C Funktion gestartet. Aus dem, auf dem Stapel, veränderten Muster läßt sich auf die benötigte Stapellänge rückschließen.

einem Überlaufen eines Stapels, indem sie jedem im virtuellen Adreßraum „anonymous pages“ folgen läßt. Greift ein Thread über das Stapelende hinaus auf Adressen zu, so wird dies dem darunterliegenden Prozeß signalisiert. Diesem obliegt es dann geeignet zu reagieren. Für QuickThreads Applikationen ist dies allerdings nur bedingt geeignet, da durch solch eine Vorgehensweise die Portierbarkeit behindert würde. Die Bereitstellung von Mechanismen zur Erstellung der als anonym deklarierten Seiten, würde Eingriffe in den Kern unumgänglich machen.

Eine weitere Möglichkeit solche Fehler einzugrenzen oder diese gar auszuschließen, soll im folgenden vorgestellt werden. Dazu wäre eine Bibliothek denkbar, die zu jedem Zeitpunkt Zusicherungen bezüglich der Konsistenz der Stapelbereiche erlaubt. Diese Bibliothek könnte folgende Funktionalität besitzen.

```
qtsa_t *QTsa_init(long max_threads);
int QTsa_create(qtsa_t *sas_base, qt_t **sa_id, void *real_id,
                qt_t *sp_base);
int QTsa_update(qtsa_t *sas_base, qt_t **sa_id, qt_t *curr_sp);
int QTsa_check(qtsa_t *sas_base, qt_t **sa_id, void **victim);
int QTsa_abort(qtsa_t *sas_base, qt_t **sa_id);
```

Um weitere Systemaufrufe auf den Stapeln der QuickThreads zu vermeiden, wird der nötige Verwaltungsbereich starr durch `QTsa_init()` für `max_threads` allokiert. Dadurch entfällt auch eine sonst wahrscheinliche Verschränkung der Stapelbereiche und der Verwaltungsstrukturen. Würde ein Stapel überlaufen, wären die Verwaltungsstrukturen selbst in Gefahr.

Um einen Thread spezifizieren zu können, muß ein über die gesamte Threadlaufzeit konstanter Wert gefunden werden, der auch in der `helper()` Funktion bekannt ist. Dazu ist zweckmäßigerweise der Zeiger auf das Speicherwort, in dem der aktuelle Stapelzeiger des Threads gespeichert ist, gewählt. Der Identifikator ist also vom Typ `qt_t**`.

Jeder Thread erhält auf Anfrage einen Verwaltungsbereich zugewiesen, in den er eine Prüfsumme über seinen Stapel einträgt. `QTsa_create()` reserviert einen solchen Bereich und legt dort die Prüfsumme des initialisierten Stapels ab. Die Funkion `QTsa_update()` erlaubt es, die Prüfsumme eines sich blockierenden Threads zu aktualisieren. Der geeignetste Platz hierfür ist die `helper()` Funktion. Mittels `QTsa_check()` können die gespeicherten Prüfsummen dann jederzeit mit den aktuellen Prüfsummen der Stapel verglichen werden. Geschieht die Überprüfung außerhalb der `helper()` Funktion, so ist der laufende Thread von dieser auszuschließen, da sein Stapel gerade benutzt wird und daher seine gespeicherte Prüfsumme nicht aktuell wäre. Der Ausschluß wird durch Angabe seines Identifikators erreicht. In der `helper()` Funktion kann keine Überprüfung durchgeführt werden, da der Identifikator des neuen Threads nicht bekannt ist und weiter nicht ausgeschlossen werden kann, daß der Stapel des neuen Threads schon verändert wurde.

Die Verwaltung der oben dargestellten Sicherungsstrukturen ist bei Multiprozessoren durch geeignete Synchronisation zu serialisieren.

Dies alles führt allerdings wiederum zu einer gravierenden Laufzeitverlängerung.

# 7 Laufzeitmessungen

## 7.1 Messungen mit dem `time/raw` Werkzeug

Wie bereits erwähnt, verfügt das QuickThreads Paket über ein eigenes Programm zur Messung von Laufzeiten. Getestet werden können dabei sämtliche QuickThreads Basisoperationen sowie Operationen des darauf aufsetzenden Simple Threads Pakets. Bei allen diesen Messungen wird eine hohe Anzahl an Operationen auf einer sehr geringen Anzahl an Threadstrukturen ausgeführt[11]. Dies bedeutet, daß Speicherzugriffe nahezu immer durch den Cache befriedigt werden können. Diese Annahme ist in realen Applikationen kaum zuläßig.

Alle hier angegebenen Meßwerte sind „user"-Zeiten und wurden mit dem Kommando `/bin/time` gemessen. Getestet wurde auf einer HP715 Workstation sowie auf einem Prozessorknoten des Convex SPP. Beide Maschinen benutzen den HP7100 Prozessor. Allerdings unterscheiden sie sich in der Taktrate und bei der Größe des installierten Caches. Wegen der geringen Größe und kleinen Anzahl der Threadstrukturen (16×4 Byte) spielt die Cachegröße hier aber eine untergeordnete Rolle.

Abbildung 19 listet die Laufzeiten für die Initialisierung eines QuickThreads. Wie in Abschnitt 4.3 beschrieben, existieren zur besseren Anpaßung an die von der Applikation gestellten Anforderungen zwei Initialisierungsoperationen.

| Initialisierung | Laufzeit | | Taktzyklen |
|---|---|---|---|
| | HP715 (33 MHz) | SPP (100 MHz) | |
| `QT_SP()` und `QT_ARGS()` (feste Argumentenzahl) | | | |
| | 1,53$\mu$s | 0,5$\mu$s | ≈50 |
| `QT_SP()` und `QT_VARGS()` (variable Argumentenzahl) | | | |
| kein Argument | 4,4$\mu$s | 1,4$\mu$s | ≈143 |
| zwei Argumente | 5,8$\mu$s | 1,9$\mu$s | ≈191 |
| vier Argumente | 7,3$\mu$s | 2,4$\mu$s | ≈240 |
| acht Argumente | 10,5$\mu$s | 3,4$\mu$s | ≈343 |

Abbildung 19: *Zeiten und Taktzyklen für die Initialisierung eines QuickThreads.*

Die folgende Abbildung 20 geht auf die Dauer eines Kontextwechsels ein. Das QuickThreads Testprogramm verkettet zur Messung eine kleine Anzahl an QuickThreads starr, so daß jegliche Schedulingentscheidung entfällt. Diese starr verkettete Liste wird während des Tests mit einer sehr hohen Anzahl an Durchläufen belastet.

Auch für den Kontextwechsel sind im QuickThreads Paket zwei von der Komplexität unterschiedliche Operationen vorgesehen. `QT_BLOCKI()` schränkt den Prozessorkontext auf die callee-saves Integer- und callee-saves Space Register ein (vgl. 4.4). `QT_BLOCK()` dagegen sichert alle callee-saves Register nach Konvention.

[11] In der momentanen Implementierung sind dies typischerweise $10^6$ bis $10^8$ Operationen auf unter 10 Threadstrukturen.

| Kontextwechsel | Laufzeit | | Taktzyklen |
|---|---|---|---|
| | HP715 (33 MHz) | SPP (100 MHz) | |
| `QT_BLOCKI()` | 3,5$\mu$s | 1,1$\mu$s | $\approx$113 |
| `QT_BLOCK()` | 4,7$\mu$s | 1,5$\mu$s | $\approx$153 |

Abbildung 20: *Zeiten und Taktzyklen für einen Kontextwechsel zwischen zwei QuickThreads.*

Die gemessene Laufzeit für die Initialisierung, Deblockierung und das Wiederverwerfen eines QuickThreads ist aus Abbildung 21 zu entnehmen.

| Starten und Verwerfen | Laufzeit | | Taktzyklen |
|---|---|---|---|
| | HP715 (33 MHz) | SPP (100 MHz) | |
| `QT_ARGS()`, `QT_BLOCKI()` und `QT_ABORT()` | 8,2$\mu$s | 2,7$\mu$s | $\approx$270 |

Abbildung 21: *Zeiten und Taktzyklen für den Start und die anschließende Vernichtung eines QuickThreads.*

Abschließend soll noch auf dem Umstand aufmerksam gemacht werden, daß es Hewlett Packard mit dem HP7100 gelungen ist, den Durchsatz des Caches bei einer Verdreifachung der Prozessortaktfrequenz proportional wachsen zu lassen.

Abbildung 23 zeigt einen gesamten Meßdurchlauf für vier moderne RISC-Prozessoren im Vergleich. Die Zuordnung der jeweils durchgeführten Operation zum Zahlenwert auf der Abszisse ist Abbildung 22 zu entnehmen. Die Messungen für AXP wurden aus [Kep93] übernommen. Alle anderen Messungen erfolgten am CIP-Pool beziehungsweise am Regionalen Rechenzentrum Erlangen.

Es folgt eine Auflistung der vier Prozessoren sowie deren Konfiguration.

**HP7100** (Convex SPP), 100 MHz, 1 MB Datencache, 1 MB Instruktionscache

**TMS390Z50** (Sun SPARCstation 10 (Model 30)), 36 MHz, 4 kB Datencache, 4 kB Instruktionscache

**AXP** (DEC 3000), 133 MHz, 8 kB Instruktionscache, 8 kB First-Level Datencache, 512 kB Second-Level Datencache

**MIPS R6000** (CD4680FS), 60 MHz, 64 kB First-Level Instruktionscache, 16 kB First-Level Datencache, 512 kB Second-Level Cache

| QuickThreads Operationen | |
|---|---|
| Nr. | Beschreibung der durchgeführten Operation |
| 1 | `QT_SP()` und `QT_ARGS()` (feste Argumentenzahl) |
| 6 | `QT_ARGS()`, `QT_BLOCKI()` und `QT_ABORT()` |
| 7 | `QT_BLOCKI()` |
| 8 | `QT_BLOCK()` |
| 10 | Varagrs Initialisierung und Start mit keinem Argument |
| 11 | Varagrs Initialisierung und Start mit zwei Argumenten |
| 12 | Varagrs Initialisierung und Start mit vier Argumenten |
| 13 | Varagrs Initialisierung und Start mit acht Argumenten |
| 14 | `QT_SP()` und `QT_VARGS()` mit keinem Argument |
| 15 | `QT_SP()` und `QT_VARGS()` mit zwei Argumenten |
| 16 | `QT_SP()` und `QT_VARGS()` mit vier Argumenten |
| 17 | `QT_SP()` und `QT_VARGS()` mit acht Argumenten |

Abbildung 22: *Legende zur folgenden Abbildung.*

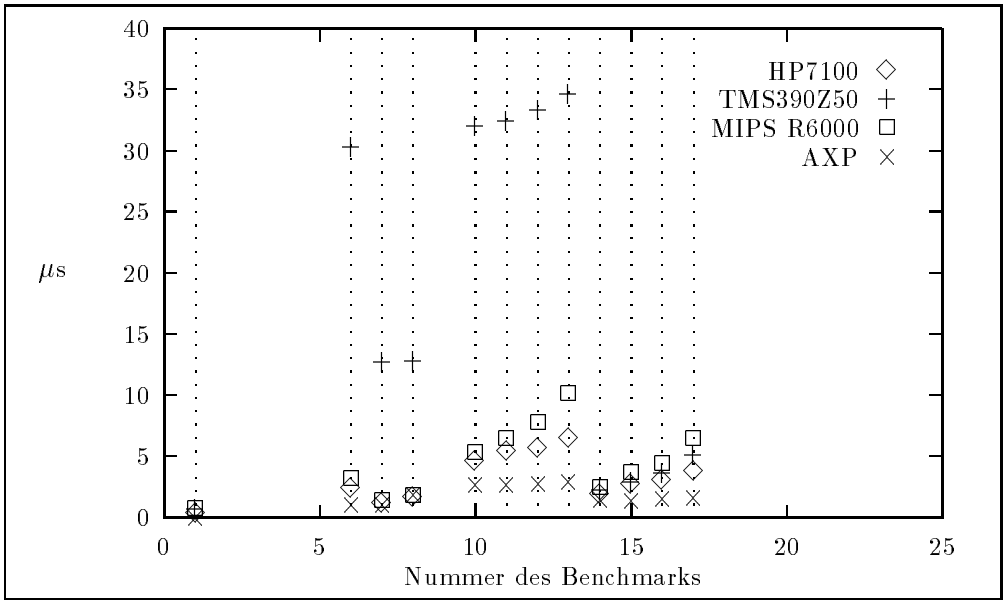



Abbildung 23: *Die Messungen 1, 6 bis 8 und 10 bis 17 von* `time/raw` *für vier Prozessoren im Vergleich.*

## 7.2 Messung von Migrationszeiten auf einer Global Shared Memory Maschine

Die Messungen wurden auf dem am Regionalen Rechenzentrum Erlangen installierten Convex SPP durchgeführt.

Ein Convex SPP ([CNX93]) besteht aus einem oder mehreren „Hypernodes“. Bei einem Hypernode handelt es sich um einen symmetrischen Multiprozessor mit bis zu acht Prozessoren. Wie Abbildung 24 verdeutlicht, sind die CPUs über einen Kreuzschienenverteiler mit dem Speicher gekoppelt. Zur Leistungssteigerung können mehrere dieser Hypernodes miteinander verbunden werden. So entsteht auf der Basis der Hypernodes eine NUMA[12]-Architektur.

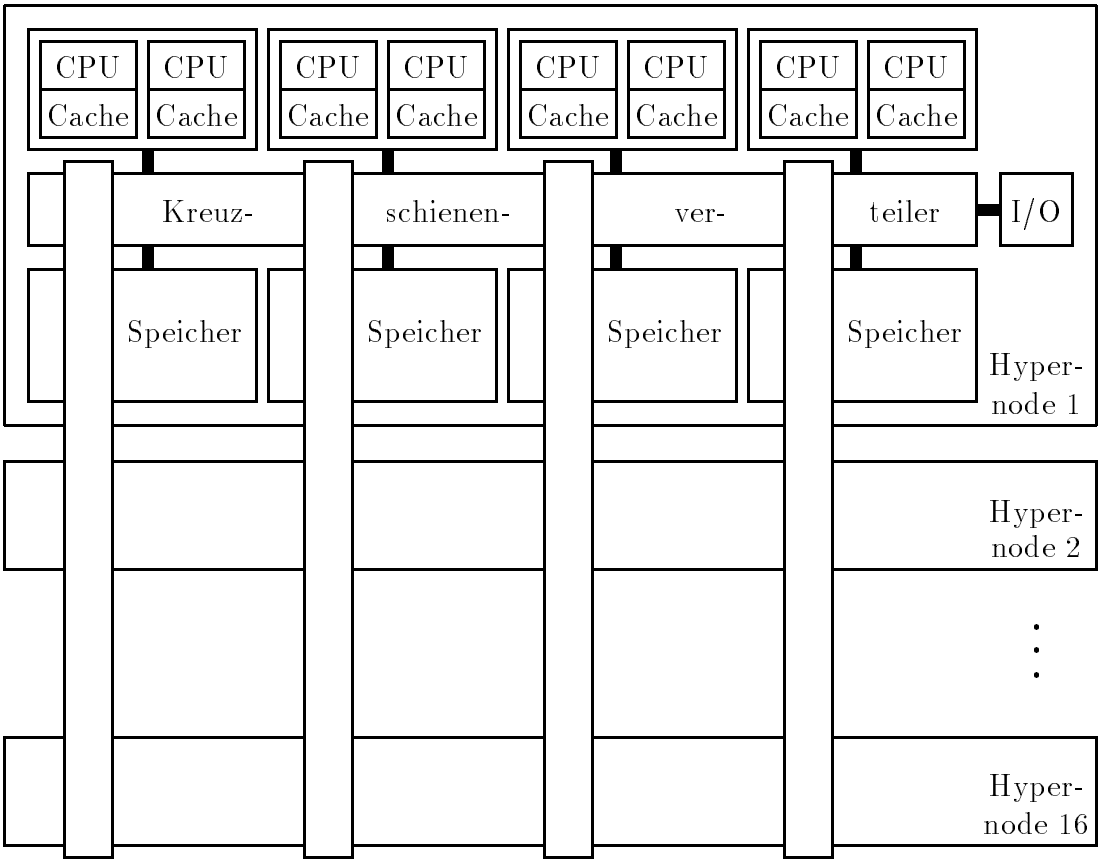



Abbildung 24: *Die Architektur des Convex SPP.*

Pro Hypernode stehen wie erwähnt acht HP PA-RISC Prozessoren zur Verfügung. Dies sind Prozessoren des Typs HP7100 mit einer Taktfrequenz von 100 Megahertz. Jeder Prozessor verfügt über jeweils ein Megabyte Daten- und Instruktionscache.

[12]NUMA steht für Non Uniform Memory Access

Das Betriebssystem SPP-UX basiert auf einem Mach 3.0 Kernel. Im System verteilter Speicher wird in einem gemeinsamen virtuellen Adreßraum verwaltet. Man bezeichnet dies als „global shared memory“.

Um Anhaltspunkte für die Implementierung einer dynamischen Lastverteilung auf Benutzerebene zu erhalten, wurden hier Messungen zur Threadmigration durchgeführt. Dazu wurde auf einem Prozessor des Systems eine große Anzahl an QuickThreads erzeugt und starr verkettet. Für den Stapel wurden dabei pro Thread 1024 Byte zur Verfügung gestellt. Wird ein Durchlauf dieser Kette angestoßen, so führt ein Thread nach dem anderen einen Kontextwechsel zu seinem Nachfolger durch. Die erste Bearbeitung dieser Kette wird dabei lokal auf dem erzeugenden Prozessor vorgenommen. Ein weiterer Lauf referenziert die Threadkette von einem anderen Prozessor im gleichen Hypernode und ein dritter von außerhalb des Hypernodes.

Die Messung der user-Zeiten wurde mit den `ttr` Funktionen durchgeführt. Diese Funktionen verwalten das thread-timer Register und liefern die Laufzeit eines einzelnen Kernel Threads. Die `times()` Funktion dagegen addiert die Laufzeiten aller Kernel Threads in der Task und ist daher für die hier benötigte Aufgabe unbrauchbar. Zur Bestimmung der real-Zeiten wurde die `gettimeofday()` Funktion verwendet.

Abbildung 25 zeigt die Meßwerte für das Initialisieren eines QuickThreads. Zur Ermittlung der Werte wurden 100.000 QuickThreads initialisiert.

Besonders ist hier auf die hohe im Kern verbrachte Zeit hinzuweisen. Unter Mach wird mittels `malloc()` angeforderter Speicher erst zu dem Zeitpunkt wirklich bereitgestellt, zu dem er erstmals referenziert wird. Für diese Bereitstellung benötigt das Betriebssystem des Convex SPP eine relativ große Zeitspanne, die sich wahrscheinlich aus der dabei nötigen Synchronisation ergibt.

| Initialisierung | Laufzeit | | | | Taktzyklen aus |
|---|---|---|---|---|---|
| | ttr-user | user | system | real | ttr-user errechnet |
| `QT_ARGS()` | 2,04$\mu$s | 2,20$\mu$s | 107,40$\mu$s | 109,60$\mu$s | $\approx$204 |

Abbildung 25: *Zeiten und Taktzyklen für die Initialisierung eines QuickThreads.*

Die Migration eines QuickThreads in einem Global Shared Memory System wird implizit durch dessen Deblockierung angestoßen. Ein Zugriff über den Stapelzeiger des QuickThreads löst dabei die Einlagerung der Seite, in der sich die referenzierte Adresse befindet, aus. Mit der neu eingelagerten Seite ist der gesamte Stapel oder ein Teil dessen in den physikalischen Speicher des anfordernden Hypernodes gewandert.

Um den großen Einfluß des Caches auf diese Messungen zu unterstreichen, wurden diese jeweils zweimal unmittelbar nacheinander mit einer unterschiedlichen Anzahl an verketteten QuickThreads durchgeführt. Abbildung 26 zeigt die Messungen für 10.000 Quick/-Threads, während Abbildung 27 die gleichen Messungen für 100.000 QuickThreads darstellt. In Abbildung 26 zeigt sich im jeweils zweiten Lauf die Einwirkung des Caches. Die Laufzeiten sinken erheblich. Bei einer hier nicht abgebildeten Messung mit 1.000 QuickThreads wurden nahezu die Ergebnisse erziehlt, die im Abschnitt 7.1 in Abbildung 20 niedergelegt sind. Die

Anzahl der Threads, die den Messungen in Abbildung 27 zugrunde liegen, ist so hoch, daß die Threadumschaltungen keinen Nutzen aus dem Cache ziehen können.

| Kontextwechsel/Migration | Laufzeit | | Taktzyklen aus |
|---|---|---|---|
| | ttr-user | real | ttr-user errechnet |
| Referenzierung von erzeugendem Prozessor | | | |
| erster Lauf | 11,4$\mu$s | 13,6$\mu$s | $\approx$1140 |
| zweiter Lauf | 6,7$\mu$s | 8,9$\mu$s | $\approx$670 |
| Referenzierung von einem Prozezessor des lokalen Hypernodes | | | |
| erster Lauf | 7,3$\mu$s | 9,5$\mu$s | $\approx$730 |
| zweiter Lauf | 6,7$\mu$s | 8,8$\mu$s | $\approx$670 |
| Referenzierung von einem Prozezessor außerhalb des lokalen Hypernodes | | | |
| erster Lauf | 16,7$\mu$s | 1578,4$\mu$s | $\approx$1670 |
| zweiter Lauf | 6,7$\mu$s | 9,5$\mu$s | $\approx$670 |

Abbildung 26: *Zeiten und Taktzyklen für einen Kontextwechsel inklusive Threadmigration. Grundlage für diese Messung war jeweils ein Kontextwechsel zwischen 10.000 verschiedenen QuickThreads.*

| Kontextwechsel/Migration | Laufzeit | | Taktzyklen aus |
|---|---|---|---|
| | ttr-user | real | ttr-user errechnet |
| Referenzierung von erzeugendem Prozessor | | | |
| erster Lauf | 14,34$\mu$s | 14,61$\mu$s | $\approx$1434 |
| zweiter Lauf | 11,08$\mu$s | 11,33$\mu$s | $\approx$1108 |
| Referenzierung von einem Prozezessor des lokalen Hypernodes | | | |
| erster Lauf | 11,22$\mu$s | 11,47$\mu$s | $\approx$1122 |
| zweiter Lauf | 11,02$\mu$s | 11,27$\mu$s | $\approx$1102 |
| Referenzierung von einem Prozezessor außerhalb des lokalen Hypernodes | | | |
| erster Lauf | 18,05$\mu$s | 1592,30$\mu$s | $\approx$1805 |
| zweiter Lauf | 14,77$\mu$s | 15,12$\mu$s | $\approx$1477 |

Abbildung 27: *Wie oben, allerdings mit 100.000 QuickThreads.*

Die Unterschiede in Abbildung 27 zwischen dem ersten und zweiten Lauf auf dem erzeugenden Prozessor ergeben sich aus der Tatsache, daß beim ersten Lauf in der Startfunktion `qt_start` verschiedene Registerkopierarbeiten durchgeführt werden. Diese Differenz zwischen den Laufzeiten geht in der Messung mit 10.000 QuickThreads (Abbildung 26) in der, durch die hohe Anzahl an cache-hits ausgelösten, erheblich höheren Differenz unter.

Wie nicht anders zu erwarten, weichen die Meßwerte für die Referenzierung von einem Prozessor innerhalb eines Hypernodes kaum von der Referenzierung des erzeugenden Pro-

zessors ab. Alle Prozessoren innerhalb eines Hypernodes greifen gleichberechtigt über einen Kreuzschienenverteiler auf den Speicher zu.

Der erste Zugriff von einem anderen Hypernode auf die Threadkette benötigt in etwa das einhundertfache der real-Zeit, die ein lokaler Zugriff innerhalb des Hypernodes verbrauchen würde. Da nach dem ersten Duchlauf alle benötigten Seiten über das Netzwerk zum referenzierenden Hypernode gewandert sind, fallen die Laufzeiten wieder auf die üblichen Werte ab.

# 8 Möglichkeiten zur weiteren Leistungssteigerung

Verschiedene Prozessorarchitekturen verfügen über einen `prefetch` Befehl. Diese Operation erlaubt es, eine Speicherseite anzufordern, ohne auf sie warten zu müssen. Man benutzt diesen Befehl um Seiten von denen bekannt ist, daß sie in naher Zukunft verwendet werden, einzulagern. Um zu erreichen, daß der Stapel des zu deblockierenden QuickThreads rechtzeitig für den ersten Zugriff im Hauptspeicher beziehungsweise Cache des Prozessors vorliegt, kann ein solcher `prefetch` Befehl eingesetzt werden. Der Stapelzeiger des neuen Stapels wird `qt_blocki()` als Argument übergeben, so daß der `prefetch` Befehl schon vor dem Sichern des Kontextes, des zu blockierenden QuickThreads, eingesetzt werden kann.

Bei der Implementierung eines auf die QuickThreads aufsetzenden Thread Pakets kann unter PA-RISC mit einem Trick die Zeit für einen Kontextwechsel weiter eingeschränkt werden. Der Zeiger auf die `helper()` Funktion wird `qt_blocki()` in Form eines Procedure Labels übergeben. Dieses Procedure Label muß dann bei jedem Aufruf von `qt_blocki()` in `$$dyncall`neu ausgewertet werden. Verlegt man die Auswertungen des Procedure Labels nach außen, so ist diese nur ein einziges Mal pro `helper()` Funktion durchzuführen. Infolgedessen könnte eine vereinfachte `$$dyncall` Funktion eingeführt werden, die auf das Testen des L-Bits verzichtet.

Um die Anzahl der von `QT_BLOCK()` zu sichernden callee-saves Register zu minimieren, wären Veränderungen der Aufruf Konventionen anzustreben. Die Aufteilung der Registermenge sollte dann zugunsten der caller-saves Register vorgenommen werden [Kep93]. Ein Aufrufer von `QT_BLOCK()` hätte dann die Aufgabe den Hauptanteil der Register zu sichern. Dieser würde aber die sogenannten „live Register“ kennen und müßte daher auch nur diese sichern.

# 9 Zusammenfassung

Wie in Kapitel 2 dieser Arbeit begründet, sind herkömmliche Prozeßkonzepte für massiv parallele Applikationen in keinem Fall tragbar. User-Level Thread Bibliotheken erlauben dagegen den Einsatz von Threads, die auf ein bestimmtes Problem zugeschnitten werden können. Desweiteren zeichnet sie ein weitgehender Verzicht auf Interaktionen zwischen Bibliothek und Betriebsystemkern aus. Die hohe Effizienz der User-Level Threads beruht im wesentlichen auf diesen beiden Punkten. Ein vollkommener Verzicht auf Schnittstellen zwischen Kern und User-Level Thread Bibliothek führt allerdings zu diversen Unzulänglichkeiten. Daher müssen bestimmte Informationen dem Kern übergeben beziehungsweise der Bibliothek zugänglich gemacht werden.

Für den Kontextwechsel zwischen User-Level Threads haben sich der Preswitch und das Konzept des Stateless Schedulers als besonders tragfähig erwiesen. Andere in Kapitel 3 vorgestellte Modelle fallen wegen unzureichender Effizienz oder Flexibilität aus.

Das in Kapitel 4 beschriebene QuickThreads Paket stellt Grundoperationen zur Erstellung eines User-Level Thread Pakets bereit. Hierbei handelt es sich um den maschinennahen Teil, der im wesentlichen in Assembler programmiert ist. Diese Operationen zeichnen sich durch exzellente Laufzeitergebnisse aus. Allerdings verschlechtern sich die angegebenen Zeiten noch durch das Aufsetzen des, dem Benutzer zugänglichen Thread Pakets. Die angebotene Flexibilität wird dabei nicht nur zur Optimierung des Laufzeitverhaltens genutzt. Sie dient auch als Grundlage für das Aufsetzen einer breiten Palette an User-Level Thread Paketen.

Ein Teil dieser Arbeit war die in Kapitel 5 vorgenommene Portierung des QuickThreads Pakets auf die PA-RISC Architektur von Hewlett Packard. Alle Merkmale dieser Architektur erwiesen sich als gut dokumentiert. Nicht zuletzt deswegen gestaltete sich die Portierung unkompliziert. Besonderer Wert wurde dabei auf die Einhaltung sämtlicher Konventionen gelegt. Das von Hewlett Packard definierte Backtracing wird aktiv unterstützt. An der Portierung wurden nach Abschluß der Arbeiten Laufzeitmessungen für typische Threadverwaltungsfunktionen durchgeführt. Die Ergebnisse liegen im Spitzenfeld moderner RISC-Architekturen.

Bei der Fehlersuche in QuickThreads Applikationen stellte sich die Größe des Stapels bald als äußerst kritisch heraus. Eine Lösung dieses Problems auf der Ebene der QuickThreads würde aber entweder die Portierbarkeit oder die Effizienz beeinträchtigen. In Kapitel 6.3 werden einige dieser Lösungsansätze aufgezeigt.

Sollen User-Level Threads einer breiten Gruppe von Benutzern zugänglich gemacht werden, muß die Fehlersuche durch das System stärker unterstützt werden. Die effiziente Überwachung der Stapel der Threads steht hier im Mittelpunkt. Weiter zu fordern ist ein Debugger, der die Möglichkeit eines dynamischen Bindens mit einer Debugging Bibliothek unterstützt.

# Literatur


[And92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, Henry M. Levy. *Scheduler activations: Effective kernel support for the user-level management of parallelism.* Transaction on Computer Systems. ACM, 10(1):53-79, Februar 1992.

[CNX93] CONVEX Computer Corporation. *Exemplar Architecture.* Richardson, Texas. First Edition, November 1993. Document Number: 081-023430-000

[Hof91] Fridolin Hofmann, *Betriebssyteme: Grundkonzepte und Modellvorstellungen.* 2. überarbeitete Auflage. Teubner, Stuttgart, 1991.

[HPA92] Hewlett-Packard Company. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* Cupertino, California. Second Edition, September 1992. HP Part Number: 09740-90039.

[HPB91] Hewlett-Packard Company. *Assembly Language Reference Manual.* Cupertino, California. Fourth Edition, January 1991. HP Part Number: 92432-90001.

[HPC91] Hewlett-Packard Company. *PA-RISC Procedure Calling Conventions Reference Manual.* Cupertino, California. Second Edition, January 1991. HP Part Number: 09740-90015.

[Kep93] David Keppel. *Tools and Techniques for Building Fast Portable Threads Packages.* University of Washington. Technical Report UWCSE 93-05-06.

[Kle92] Jürgen Kleinöder. *Ausgewählte Kapitel der praktischen Betriebsprogrammierung I.* Vorlesungsskript Universität Erlangen/Nürnberg — IMMD IV. WS 1992/93.

[KlRi93] Jürgen Kleinöder, Thoms Riechmann. *Hierarchische Scheduler in der PM-Systemarchitektur.* Universität Erlangen/Nürnberg — IMMD IV.

[Kop93] Christoph Koppe. *Convex SPP — Automatische Parallelisierung mit C und Programmierung mit Threads.* Vorlesungsskript Universität Erlangen/Nürnberg — IMMD IV. SS 1993.

[MuSG93] Bodhisattwa Mukherjee, Karsten Schwann, Prabha Gopinath. *A Survey of Multiprocessor Operation System Kernels.* College of Computing, Georgia Institute of Technology. Atlanta, Georgia, 1993. GIT-CC-92/05.

[PKBS91] M. L. Powell, S. R. Kleimann, S. Barton, D. Shah, D. Stein, M. Weeks (Sun Microsystems Inc.). *SunOS Multi-thread Architecture.* Winter'91 USENIX. Dallas, Texas, 1991.

[Snu93] Georg Schnurer. *Prozessor Poker.* c't Magazin für Computertechnik. Heise, Hannover, Dezember 1993.

[StSh92] Dan Stein, Devang Shah. *Implementing Lightweight Threads.* Summer'92 USENIX. San Antonio, Texas, 1992.

[TuGu89] A. Tucker, A. Gupta. *Process control and scheduling issues for multiprogrammed shared memory multiprocessors.* Proceedings of the 12th ACM Symposium on Operating Systems Principles, Dezember 1989.