Coscheduling: Prozeßwechselentscheidung in User-Level Kommunikationsbibliotheken

Joachim M. Blum, Thomas M. Warschko, and Walter F. Tichy
Universität Karlsruhe, Fakultät für Informatik
Postfach 6980,76128 Karlsruhe,Germany
email:{blum,warschko,tichy}@ira.uka.de

27. Mai 1999

Zusammenfassung

Dieses Papier verdeutlicht Effizienzprobleme, die durch fehlende Prozeßinformation innerhalb multiprozeß-/ multiuserfähigen User-Level Kommunikationsbibliotheken auftreten. Trifft der Kern aufgrund der fehlenden Prozeßinformation eine falsche Entscheidung bei der Wahl des zu aktivierenden Prozesses, kommt es in vielen Fällen zu erheblichen Leistungseinbußen des Gesamtsystems.

Es werden Verfahren vorgestellt, die dieses Problem beheben und die Leistung des Gesamtsystems deutlich erhöhen. Ressourcenkonflikte können ohne diese Techniken in ca. 30000 μs aufgelöst werden. Mit diesen Techniken verringert sich die benötigte Zeit auf 5 μs. Insbesondere wenn User-Level Kommunikationsbibliotheken in multi-threaded Applikationen verwendet werden, sind diese Techniken dringend notwendig.

1 Einführung

Nachdem [BLA+94] den Kern des Betriebssystem aus dem Kommunikationspfad verbannt hatte und damit das Prinzip der User-Level Kommunikation begründete, folgten sehr viele Forschungsprojekte [PLC95, ACP95, WTH95, BDF+95] Mitte der neunziger Jahre diesen Weg. Sie eliminierten die komplette Funktionalität, die der Kern erbrachte und erreichten so Kommunikationsleistungen, die nahe an die Maximalleistung der Hardware heranreichten. Beispielsweise erreichte ein System auf Basis der ParaPC Karte [WTH95] 1994 schon Latenzzeiten von 5 μs. Der Nachfolger ParaStation reduzierte diesen Wert 1995 auf 3 μs.

Diese Latenzzeiten waren nur möglich, da der Betriebssystemkern vom Kommunikationspfad entfernt wurde. Das Problem das man sich damit einhandelte war der Verlust einer zentralen Instanz (dem Kern), die den Zugriff auf die singuläre Ressource Kommunikationshardware koordinierte. Die ersten Ansätze lösten diese Problem dadurch, daß sie die Anzahl der auf die Kommunikationshardware zugreifenden Prozesse auf einen reduzierten und somit das Koordinationsproblem umgingen.

Mit der Entwicklung der ParaStation Software wurde ab 1995 die Multiprozeßfähigkeit auch in der User-Level Kommunikation eingeführt [WBT96]. Diesem Beispiel folgten auch andere Projekte, wie zum Beispiel Active Messages [CMC97] oder FastMessages [CPL+97].

Nahezu alle Projekte, die sich mit Cluster-Computing beschäftigen, blenden die Hardware in den Benutzerbereich der Applikation ein und sprechen bei der Kommunikation die Funktionalität direkt aus dem Benutzerbereich an. Das Betriebssystem wird aus dem kritischen Pfad

der Kommunikation vollständig verdrängt. Für das Betriebssystem ist ein kommunizierender Prozeß nicht mehr von einem rechnenden zu unterscheiden, es erlangt keine Erkenntnis über eingehende oder ausgehende Nachrichten und kann die Kommunikation nicht mehr in seine Auswahlstrategien während der Wahl eines zu aktivierenden Prozesses einbeziehen. Dies bedeutet wiederum, daß Prozesse, die auf das Empfangen einer Nachricht warten, für das Betriebssystem *rechenwillig* sind und somit mit gleicher Wahrscheinlichkeit aktiviert werden, wie ein Prozeß der gerade richtige Berechnungen durchführt.

Der ausgewählte Prozeß wartet aktiv auf die Ankunft einer Nachricht und benutzt CPU Zyklen, die von anderen Prozessen sinnvoller verwendet werden könnten. Der fälschlicherweise aktivierte Prozeß behindert damit den sequentiellen und parallelen Workload auf jedem Knoten des Clusters. Solange man das gesamte Cluster nur als dedizierten Parallelrechner nutzt, bei dem auf jedem Knoten nur ein Prozeß abläuft, fällt diese Behinderung nicht ins Gewicht. Wenn das Cluster von verschiedenen parallelen und sequentiellen Applikationen zeitgleich genutzt wird, wird die CPU Zeit jedoch nur zu einem Bruchteil sinnvoll genutzt. Dies steht im Widerspruch zu den Zielen des ParaStation-Projekts Rechnerbündel in einem breiteren Einsatzspektrum zu betreiben.

Besonders gravierend fällt die Behinderung bei multi-threaded Applikationen aus. Wenn ein Thread aktiv auf den Empfang einer Nachricht wartet behindert er andere Threads innerhalb der selben Applikation bei ihren Berechnungen. Ein bedeutender Grund für die multi-threaded Programmierung ist jedoch gerade das Überdecken von blockierenden Aufrufen mit sinnvollen Aktivitäten. Dies wird im allgemeinen damit erreicht, daß blockierende Aufrufe vom System erkannt werden und danach jeweils zu anderen Threads umgeschaltet wird. Dieses Umschalten ist relativ günstig, da kein Adreßraumwechsel zu andern Applikationen vorgenommen werden muß. Wenn die blockierenden Aufrufe nun nicht mehr für das System erkennbar sind, geht ein bedeutender Vorteil der multi-threaded Programmierung verloren.

Konkret tritt dieses Problem bei Java RMI (*Remote Methode Invocation*) auf, wo im allgemeinen gleich mehrere Threads auf eine Rückantwort warten. Messungen von Java RMI auf der ParaStation User-Level Kommunikationsbibliothek zeigten ohne Coscheduling erwartungsgemäß keine gute Leistung. Dies war der Anlaß das Problem innerhalb des ParaStation Projekts anzugehen.

Im Parallelrechnerbetriebssystemen hat man schon früher erkannt, daß ohne Wissen über die Applikation Schedulingentscheidungen oft zu relativ schlechten Ergebnissen führen. Um dies zu verhindern gibt man der Applikation bzw. dem Laufzeitsystem die Möglichkeit das Betriebssystem bei der Schedulingentscheidung zu unterstützen. Diese Methode wird *Coscheduling* genannt. Ein ähnliches Verfahren existiert auch für FastMessages [SPWC97](Dynamic Coscheduling), wo wartende Prozesse aufgrund von ankommenden Nachrichten aktiviert werden. Allerdings ist FastMessages nicht für eine beliebige Anzahl von Prozessen geeignet.

Mit ähnlichen Verfahren wollen auch wir die User-Level Kommunikationsbibliotheken vor allem für multi-threaded Applikationen attraktiver machen.

2 ParaStations Coscheduling

Da die ParaStation Software von Anfang an als multiprozeßfähig entworfen worden ist, wurde schon früh erkannt, daß die Gesamtleistung mit zunehmender Anzahl der ParaStation Prozesse sehr stark abnimmt. Der Grund hierfür war das aktive Warten von Prozessen an leeren Empfangs-*Ports*, die andere Prozesse in ihrem Fortkommen behinderten. Mit der Einführung

einer freiwilligen Aufgabe des Prozessors nach einer Anzahl erfolgloser Empfangsversuche wurde das Problem reduziert.

Zusätzlich zu den Empfangsroutinen wird an gesperrten Semaphoren der Prozessor freiwillig aufgegeben. Da in den Semaphoren vermerkt ist, wer diese Semaphore besitzt, kann diese Information für die Wahl des Nachfolgeprozesses verwendet werden.

Erstaunlich ist, daß die Leistung die man durch das *Coscheduling* erhält sehr stark von der verwendeten Methode abhängt, die man zur Aufgabe des Prozessors verwendet. Wir haben unterschiedliche Methoden implementiert, die nun näher erläutert werden sollen:

- *none:* Dem Betriebssystem wird keine Unterstützung gegeben, wann und auf welche Prozesse umgeschaltet werden soll. Diese Strategie ist zur Evaluation sinnvoll, da sie die Leistung das Gesamtsystem ohne *Coscheduling*-Unterstützung repräsentiert.

- *sleep:* Bei dieser Strategie wird der aktive Thread durch einen `usleep()` Aufruf schlafen gelegt. Die Zeit (in μs), die er schläft, kann über einen Parameter eingestellt werden. Das Betriebssystem wählt einen anderen rechenwilligen Thread aus.

- *yield:* Bei multi-threaded Applikationen wird die Routine `pthread_yield()` verwendet. Hierdurch wird innerhalb der PThread-Bibliothek zu einem anderen Thread gewechselt.

- *thread_switch:* Digital Unix stellt den Mach Aufruf `thread_switch()` zur Verfügung. Hiermit wird zwischen verschiedenen Betriebssystem-Threads gewechselt. Bei einem Parameter ungleich Null wird der aktuelle Thread in der Priorität verringert und somit die Wahrscheinlichkeit, daß ein anderer Thread aktiviert wird erhöht. Wenn der Parameter gleich Null ist, wird einfach nur versucht zu einem anderen Thread gleicher oder höherer Priorität zu wechseln. Dieses Verfahren nennt man *Hand-Off Scheduling* [Bla90, Bla].

- *dynamic:* In den ParaStation-Gerätetreiber integrierte Methode, die es ermöglicht einen Prozeß gezielt zu aktivieren. Allerdings bezieht sich dieses Verfahren nur auf Prozesse und nicht auf Threads, die ja die Einheit der Aktivität sind.

Problematisch bei diesen Strategien ist, daß bisher ParaStation nur *Ressourcen* kennt, die prozeßweit allokiert werden. ParaStation macht keinen Unterschied zwischen den Threads, die innerhalb eines Prozesses ablaufen. Leider stellen Betriebssysteme in ihrer Grundfunktionalität keine **schnelle** Möglichkeit zur Identifizierung eines Threads zur Verfügung. Die PThread Bibliothek bietet hierfür zwar eine schnelle Möglichkeit an, jedoch ist diese nur in multi-threaded Applikationen verfügbar. Der sicherste Weg die Identität des aktuellen Threads herauszubekommen ist das Betriebssystem zu fragen, was jedoch den Zeitgewinn, den man durch das Prinzip der User-Level Kommunikation gewonnen hat, wieder verlieren lassen würde. Somit ist es nicht möglich die Thread-ID beim locken von Ressourcen zu übergeben.

3 Java Kommunikation

Java eignet sich sehr gut für die Multi-Threaded Programmierung. Es soll hier als Beispiel für eine solche Plattform verwendet werden. Java bietet verschiedene Möglichkeiten um mit entfernten Knoten zu kommunizieren. Mit den Java-Sockets können beliebige Programme mit einer *Send-Receive-* Semantik miteinander kommunizieren. Zudem existiert die Möglichkeit, daß man über die RMI Schnittstelle (*Remote Methode Invocation*) Methoden von Objekten, die auf entfernten Knoten residieren aufruft. Die Parameter des Aufrufs werden auf der

Senderseite automatisch eingepackt und über einen Nachrichtentransfer an den entfernten Knoten geschickt. Auf dem entfernten Knoten werden die Parameter wieder ausgepackt und die gewünschte Methode am Objekt aufgerufen. Nachdem die Methode abgearbeitet ist, wird das Ergebnis zum Aufrufer gesendet. Die RMI Bibliothek bedient sich hierbei auch den Diensten der Java-Sockets. Nach dem Senden der Nachricht wartet der Sender an einem blockierenden Empfangen bis die Antwort mit dem Rückgabewert an ihn zurückgesendet wird.

Sehr viele Java Applikationen machen regen Gebrauch der RMI Funktionalität. Bei Betriebssystem Kommunikation läßt sich die Latenz bis die Nachricht versendet ist, die Methode aufgerufen wird und die Rückantwort empfangen wird, durch weitere Threads gut überdecken. Somit sind im Regelfall eine große Anzahl von Threads am Warten auf die Rückantwort oder machen gerade Berechnungen.

Auf der ParaStation benutzt das RMI nun die ParaStation Sockets. Da die ParaStation Sockets vollständig im Benutzerbereich implementiert sind, wartet ein Thread, der auf die Rückantwort wartet aktiv bis die entsprechende Nachricht ankommt. Er verwendet hierfür CPU Zyklen, die von anderen Threads verwendet werden könnten.

Da ParaStation nur die Threads kennen kann, die gerade in der Bibliothek verweilen, ist es nicht möglich, gezielt zu Threads zu wechseln, die gerade *sinnvolle* Arbeit verrichten. Das unkoordinierte Wechseln zu beliebigen Threads verursacht, daß das Betriebssystem mit hoher Wahrscheinlichkeit einen wartenden Thread aktiviert. Dieses Problem wollen wir in einem neuen Ansatz beheben.

4 ParaStation-II Coscheduling

In der neuen ParaStation Bibliothek wollen wir Threads, die gerade auf eine Nachricht warten nicht mehr für eine längere Zeit aktiv halten. Hierzu werden sie nur noch eine kurze Zeit aktiv auf neue Nachrichten warten. Danach teilen sie der Kommunikationshardware mit, daß sie auf Nachrichten an einem gewissen Port warten und deaktivieren sich selbst. Wenn eine Nachricht für diesen markierten Port eintrifft, signalisiert die Kommunikationhardware dem Kern durch einen Interrupt das Eintreffen der Nachricht. Dieser sucht den wartenden Thread und aktiviert diesen wieder. Somit erhält man den Standardkommunikationsmechanismus für den Fall, daß ein Empfangsaufruf für eine längere Zeit nicht befriedigt werden kann und die wartenden Threads werden nicht mehr bei der Prozeßauswahl herangezogen. Für den Fall, daß die Programme gut synchronisiert sind, wird die Kommunikation immernoch pur über Benutzerbereich abgewickelt. Somit erhält man bei feingranularen Programmen sehr gute Latenzzeiten und grobgranularen Programmen, bei denen manche Programmteile auf andere warten müssen, bremsen nicht mehr die Gesamtleistung des Systems.

Diese Möglichkeit des selektiven Aufweckens eines wartenden Threads ist erst durch die Programmierbarkeit der Firmware auf den Kommunikationskarten enstanden. Bei der ParaStation–I Karte bestand diese Möglichkeit nicht. Im ParaStation–II Projekt bietet der freiprogrammierbare Kommunikationsprozessor auf der Myrinet Karte die Möglichkeit anhand der eintreffenden Nachrichten zu entscheiden, welche Nachrichten welche Aktionen auslösen sollen. Der in [BWT98] beschriebene *Message Handler* auf der Kommunikationshardware zeigt hierbei seine Stärken.

5 Ergebnisse

Die im Abschnitt 2 vorgestellten Verfahren werden anhand von zwei Tests validiert und bewertet.

Der erste Test stellt die Leistung des *Coschedulings* an einer gesperrten Ressource dar. Hierfür werden zwei Prozesse bzw. Threads so ineinander verzahnt, daß sie sich immer gegenseitig behindern, indem drei Semaphore nacheinander gesperrt werden. Jeder der beiden Prozesse/Threads besitzt zu jedem Zeitpunkt immer mindestens ein Semphore. Er versucht danach ein weiteres zu bekommen, das unter Umständen vom anderen Prozeß/Thread belegt wird. Somit erreicht man einen stetigen Wechsel zwischen den Prozessen/Threads. Die erreichten Ergebnisse werden in folgender Tabelle zusammengefaßt.

Semphore Lock Test auf Alphas 21164 500 MHz

Strategie	Prozeßwechsel (Zeit in μs)	Threadwechsel (Zeit in μs)
none	58701	99482
sleep (0)	58692	99511
sleep (1)	2928	2930
yield	58770	10
thread_switch	19	99482
dynamic	29	99502

Die Zahlen repräsentieren die Zeit (in μs), die benötigt wird um eine Schleife mit den drei Sperren zu durchlaufen. Wie man erkennen kann, weichen die verschiedenen Methoden sehr stark voneinander ab. Leider gibt es kein Verfahren, das sowohl für Interthread- als auch Interprozeßwechsel gute Leistung aufweisen kann. Lediglich die Methode *sleep()* mit dem Parameter 1 μs kann für beide Anwendungsfälle ähnliche Ergebnisse vorweisen. Die Leistung von *sleep()* weicht allerdings deutlich vom jeweiligen Optimum ab.

Der zweite Test (ChaosRMI) ist ein Test mit Java und RMI. ChaosRMI [Luk98] ist eine verteilte Fraktalberechung, das aus drei Bestandteilen besteht: Einem zentraler Server, einer Benutzerschnittstelle und eine beliebige Anzahl von Rechensklaven. Der Server startet die Rechensklaven und teilt die Berechnung des Bildes unter ihnen auf.

In diesen Tests variieren wir die Problemgröße und die Coscheduling Strategien bzw. die Anzahl der Rechensklaven. Die Problemgröße entspricht hierbei der Größe des zu berechnenden Fraktalbildes.

In Abbildung 1 wird der Geschwindigkeitsgewinn durch Benutzung der ParaStation Sockets anstelle der normalen Systemsockets dargestellt. In diesem Test wird als Strategie *sleep(1)* und *yield()* verwendet, die von den verwendeten Coscheduling Strategien am besten abschneiden. Wie man erkennen kann, erhält man durch einen einfachen Austausch der Sockets eine bis zu sechzehn-fache Beschleunigung.

In Abbildung 2 werden die unterschiedlichen Coscheduling Strategien miteinander verglichen. Wie angemerkt, bieten Yield() und Sleep(1) hier die besten Werte. Wie die Abbildung zeigt, kann man durch eine geeignete Wahl der Scheduling Strategien die Leistung eines Cluster bei der Benutzung von *User-Level Kommunikationsbibliotheken* sehr beeinflussen.

Die Leistung liegt üblicherweise weit über der Leistung eines Clusters mit System- Kommunikation, jedoch müssen die Applikationen im Multiprozeßbetrieb die CPU bei gesperrten Ressourcen und leeren Empfangspforten an andere rechenwillige Prozesse abgeben.

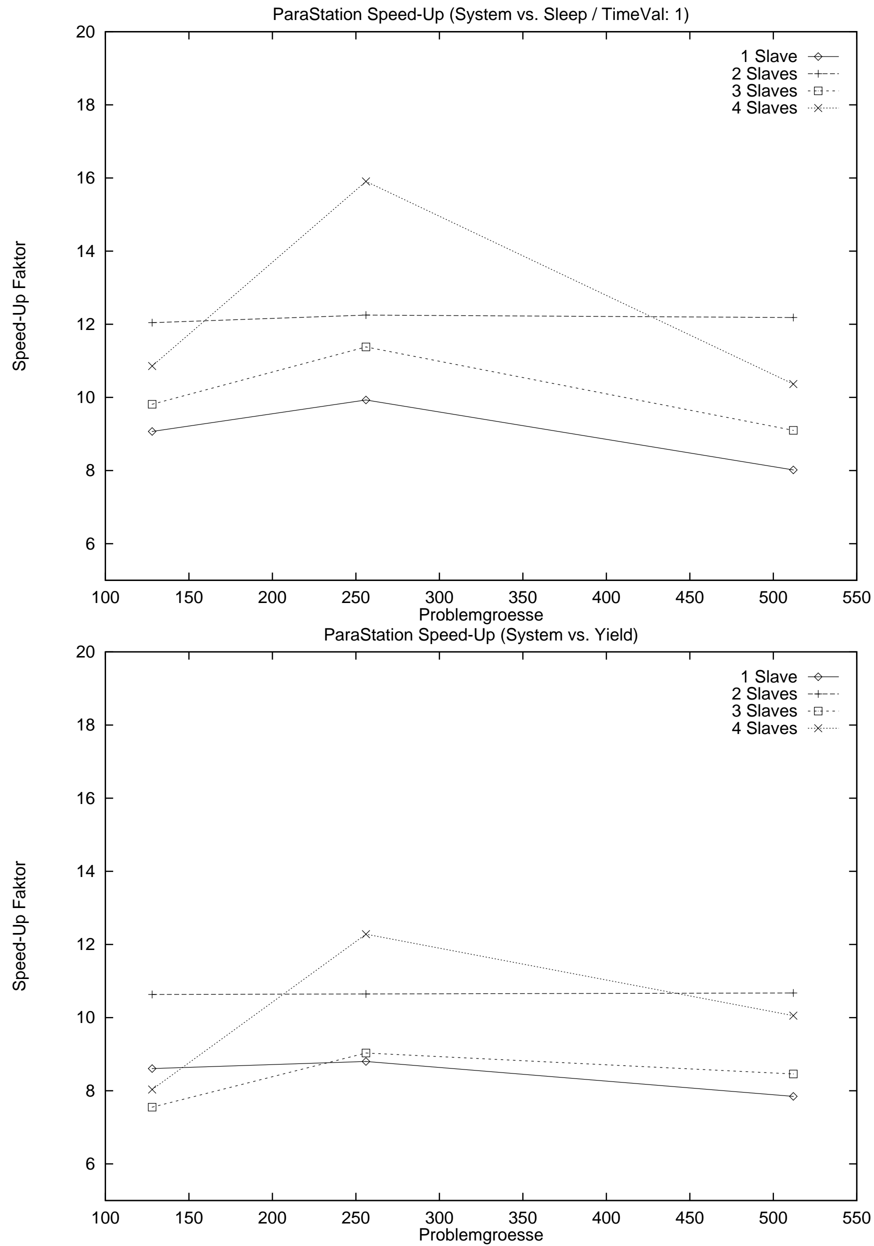

Abbildung 1: SpeedUp-Faktoren für "System vs. ParaStation-Sleep" und "System vs. ParaStation-Yield"

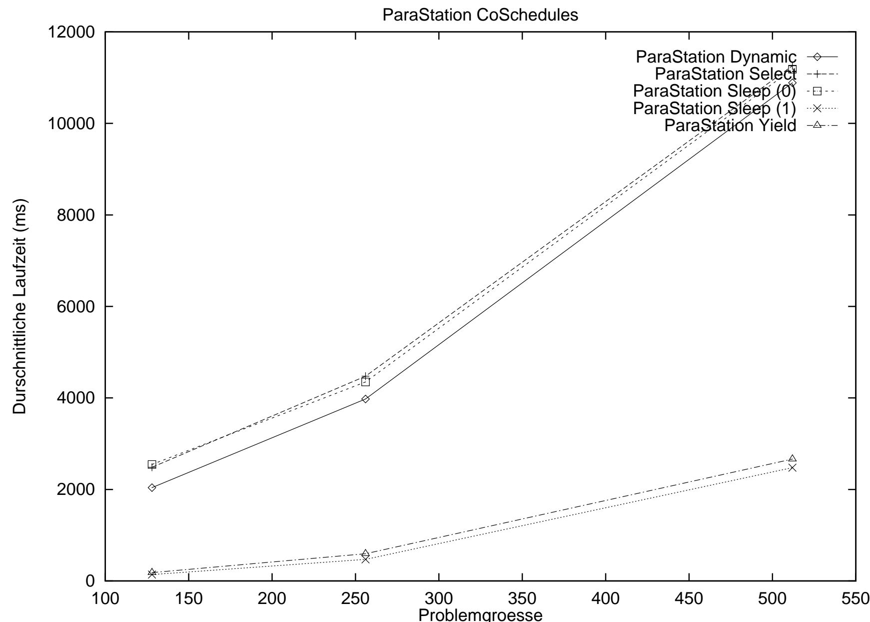

Abbildung 2: Übersicht über die Leistung der CoScheduling-Verfahren

6 Konklusion und weitere Arbeiten

Dieses Papier verdeutlicht, welche Probleme man durch das Prinzip der User-Level Kommunikation zu bewältigen hat, wenn ein Cluster nicht nur dediziert mit einer Applikation benutzt werden soll.

Es werden Verfahren vorgestellt, die die Gesamtleistung des Clusters deutlich erhöhen, wenn mehrere Thread oder Prozesse das Kommunikationssubsystem verwenden.

Als weitere Arbeiten werden wir das in Abschnitt 4 vorgestellte Verfahren implementieren, das die Kooperation zwischen Betriebssystem und der ParaStation Bibliothek weiter vertieft. Wir hoffen dadurch eine Methode des Coschedulings zu erreichen, die sowohl für multi-threaded als auch für multiprozeß Betrieb geeignet ist.

Literatur

[ACP95] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[BDF+95] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, and Malena R. Mesarina. Virtual-Memory-Mapped Network Interfaces. *IEEE Micro*, 15(1):21–28, February 1995.

[Bla] David L. Black. *Scheduling and Resource Management Techniques for Mutliprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University.

[Bla90] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. Technical Report CMU-CS-90-125, Carnegie Mellon University, April 1990.

[BLA+94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 21st International Symposium on Computer Architecture (IS-*

CA), Chicago, Illinois, pages 142–153, Los Alamitos, California, April 18-21, 1994. IEEE Computer Society Press.

[BWT98] Joachim M. Blum, Thomas M. Warschko, and Walter F. Tichy. Pulc: Parastation user-level communication. design and overview. In Jose Rolim, editor, *Parallel and Distributed Processing*, number 1388 in Lecture Notes in Computer Science, pages 498–509. Springer Verlag, March 1998.

[CMC97] B. Chung, A. Mainwaring, and D. Culler. Virtual Network Transport Protocols for Myrinet. In *Hot Interconnects'97*, Stanford, CA, April 1997.

[CPL+97] Chien, Pakin, Lauria, Buchanan, Hane, Giannini, and Prusakova. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing (PP97)*, 1997.

[Luk98] Daniel Lukic. Parastation-anbindung für java. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, 1998.

[PLC95] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, San Diego, California, December 3-8 1995.

[SPWC97] Sobalvarro, Pakin, Weihl, and Chien. Dynamic coscheduling for workstation clusters. In *submitted for publication*, 1997.

[WBT96] Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. The ParaStation Project: Using Workstations as Building Blocks for Parallel Computing. In *Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'96), New Horizons*, Sunnyvale, California, USA, August 9–11, 1996.

[WTH95] Thomas M. Warschko, Walter F. Tichy, and Christian G. Herter. Efficient Parallel Computing on Workstation Clusters. *PARS (GI) Mitteilungen*, 14:49–57, Dezember 1995.