Simulation einer vielfädigen Prozessorarchitektur

Winfried Grünewald, Theo Ungerer

Institut für Rechnerentwurf und Fehlertoleranz
Universität Karlsruhe
76128 Karlsruhe

{gruenewald, ungerer}@informatik.uni-karlsruhe.de

Zusammenfassung

Mit den vorgestellten Simulationen werden Techniken untersucht, die es erlauben, Wartezeiten, die bei Speicherzugriffen oder bei Synchronisationen entstehen, durch einen schnellen Wechsel des Kontrollfadens (Kontextwechsel) zu überbrücken. Simuliert wird die auf dem PARS-Workshop 1994 vorgestellte vielfädige (multithreaded) Prozessorarchitektur Rhamma [1], bei der die Lade-/Speicheroperationen und die Ausführungsoperationen innerhalb eines Kontrollfadens voneinander entkoppelt werden. Um die durch Datenabhängigkeiten entstehenden Wartezeiten innerhalb eines Kontrollfadens überbrücken zu können, werden die Lade-/Speicheroperationen und die Ausführungsoperationen verschiedener Kontrollfäden vom Prozessor parallel ausgeführt. Für die Lade-/Speichereinheit werden verschiedene Implementierungsmöglichkeiten vorgestellt.

Einleitung

Beim Zugriff auf den Hauptspeicher oder einen entfernten Speicher in einem Multiprozessorsystem sowie bei der Synchronisation von Threads entstehen Wartezeiten. Durch Cache-Speicher können die Wartezeiten beim Speicherzugriff im Mittel stark verringert werden. Verbessert wird diese Methode durch das Vorabladen von Daten per Software, beispielsweise durch Einfügen von Cache-Touch-Befehlen (software prefetching), oder per Hardware, wie beispielsweise das Laden des nächsten Cache-Blocks bei den PowerPC-Prozessoren (hardware prefetching). Der Ansatz der vielfädigen (multithreaded) Prozessoren kann sowohl die Wartezeiten überdecken, die beim Speicherzugriff entstehen, als auch die Zeiten, die bei der Synchronisation mehrerer Kontrollfäden auftreten. Vorausetzung ist dafür, daß genügend parallel ausführbare Kontrollfäden als Last vorhanden sind, und der Kontextwechselaufwand sehr klein ist.

Das wesentliche Merkmal für vielfädige Prozessoren ist ein schneller Kontextwechsel, der durch das Vorhandensein mehrerer Registersätze auf dem Prozessor-Chip wesentlich unterstützt wird. Dazu gibt es derzeit drei Ansätzen:

- Die *Cycle-by-Cycle-Interleaving-Technik*: Mit jedem Prozessortakt wird ein Befehl eines anderen Kontrollfadens in die Prozessor-Pipeline eingefüttert. Dieses bei den Rechnern HEP [2], Horizon [3] und Tera [4] eingesetzte Verfahren besitzt den Nachteil einer geringen Leistung, falls nur wenige Threads als Last zur Verfügung stehen. Denn im Regelfall wird ein Befehl desselben Thread erst in die Prozessor-Pipeline eingespeist, nachdem der Vorgängerbefehl die Pipeline vollständig durchlaufen hat.

- Die *Block-Multithreading-Technik* [5]: Die Befehle eines Thread werden solange aufeinanderfolgend ausgeführt, bis ein Ereignis eintritt, das zu Wartezeiten führt. Dann wird ein Kontextwechsel durchgeführt. Dieses Ereignis ist beim Sparcle-Prozessor [6] ein Cache-Fehlzugriff oder eine fehlgeschlagene Synchronisation. Der Nachteil dieser Technik ist, daß solche Ereignisse erst spät in der Pipeline erkannt werden und nachfolgende bereits in der Pipeline vorhandene Befehle nicht verwendet werden können. Daraus resultiert ein Kontextwechselaufwand in der Größenordnung von 14 Takten (Sparcle-Prozessor).

- Die *Interleaving- oder Simultaneous-Multithreading-Technik* [7,8]: Die Ausführungseinheiten eines superskalaren Prozessors werden simultan aus mehreren Befehlspuffern bestückt. Jeder Befehlspuffer ist einem anderen Thread zugeordnet. Damit kann theoretisch wohl der höchste Leistungsgewinn erzielt werden, jedoch sind etliche Probleme der simultanen Befehlszuordnung noch ungelöst.

Ziel des Forschungsprojekts ist es, einen Universalprozessor zu entwickeln, der Wartezeiten durch einen schnellen Kontextwechsel überbrückt. Dabei wird eine Block-Multithreading-Technik untersucht, bei welcher die Kontextwechsel im Befehlsstrom codiert sind. Im Gegensatz zu ähnlichen Projekten, wie dem Sparcle-Prozessor, wird durch die Entkopplung von Ausführungs- und Lade-/Speichereinheit bereits in der Decodierphase der Pipeline erkannt, ob ein Kontextwechsel vorgenommen werden muß. Dadurch kann der Aufwand für einen Kontextwechsel auf wenige Prozessortakte reduziert werden. Allerdings werden Kontextwechsel häufiger als beim Sparcle-Prozessor durchgeführt.

Im folgenden Abschnitt wird die Prozessorarchitektur Rhamma skizziert und danach der Simulator beschrieben. Es werden die Ergebnisse von Simulationen vorgestellt, welche die Effizienz der Programmausführung des vielfädigen Prozessors mit derjenigen eines konventionellen Prozessors vergleichen. Insbesondere wird untersucht, welcher Aufwand für den Kontextwechsel zur Überbrückung relativ kleiner Wartezeiten tolerierbar ist. Da sich dann die Lade-/Speichereinheit als Engpaß erweist, werden verschiedene Implementierungen der Lade-/Speichereinheit simuliert und die Ergebnisses vorgestellt. Anschließend werden Hardware-Maßnahmen vorgestellt, die den zeitlichen Aufwand für den Kontextwechsel vermeiden.

Die Prozessorarchitektur

Die vielfädige Architektur Rhamma (siehe Abb. 1) ist charakterisiert durch die Entkopplung von Ausführungseinheit (execution unit), Lade-/Speichereinheit (load/store unit) und Synchronisationseinheit (sync unit). Die drei nebenläufig zueinander arbeitenden internen Einheiten sind durch FIFO-Puffer für die Thread-Kennungen (thread tags) bzw. Synchronisationsanforderungen (sync requests) verbunden. Um einen schnellen Kontextwechsel ermöglichen zu können, sind auf dem Prozessor mehrere Registersätze vorhanden (register frames), in denen jeweils alle spezifischen Daten eines Kontrollfadens gespeichert sind.

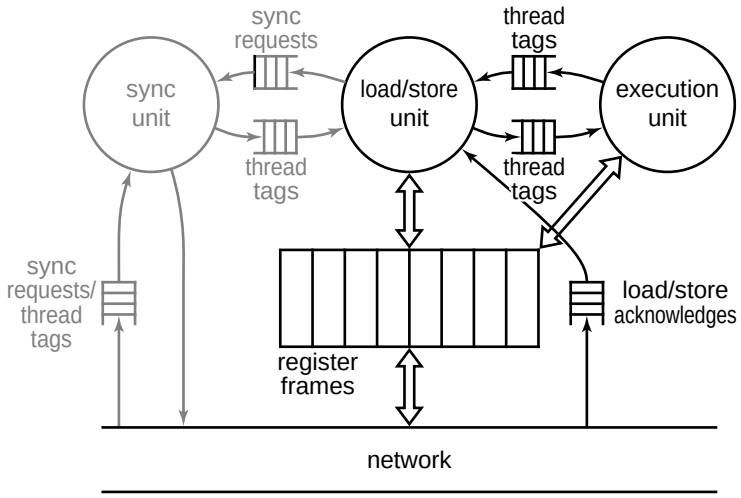

Abb. 1: Datenpfade der vielfädigen Architektur

Die Ausführungseinheit bearbeitet einen Thread so lange, bis sie auf eine Synchronisations-, Lade- oder Speicheroperation trifft. Dann übergibt sie die Thread-Kennung (thread tag) an die Lade-/Speichereinheit,

entnimmt dem FIFO-Puffer eine weitere Thread-Kennung und wechselt den Thread durch Umschalten auf den durch die Thread-Kennung angegebenen Registersatz. Die Lade-/Speichereinheit lädt parallel zur Arbeit der Ausführungseinheit Daten eines anderen Thread in dessen Registerrahmen. Je nach zugrunde gelegtem Konsistenzmodell kann die Abgeschlossenheit einer Lade- oder Speicheroperation auf dem entfernten Speicher durch den Rücklauf von Lade-/Speicherbestätigungen (load/store acknowledges) geprüft werden. Die Synchronisationseinheit führt die Synchronisationsoperationen durch, die in Synchronisationsanforderungen (sync requests) von der lokalen oder, im Falle eines Multiprozessorsystems, auch von einer entfernten Lade-/Speichereinheit angefordert sind. Im folgendem wird die Synchronisationeinheit nicht betrachtet. Eine genauere Beschreibung der Prozessorarchitektur findet sich in [1].

Der Simulator

Für die vielfädige Prozessorarchitektur wurde ein Ereignis-orientierter Simulator [9] entwickelt, der sowohl das Verhalten eines Einprozessorsystems wie auch das eines speichergekoppelten Multiprozessorsystems auf Register-Transfer-Ebene nachbildet. Als Ausführungseinheit wurde ein konventioneller Prozessor (eine Variante des DLX-Prozessors der Universität Stanford [10]) eingesetzt und dessen Befehlssatz um Synchronisations- und Thread-Management-Befehle [11] erweitert.

Folgende Vereinfachungen wurden vorgenommen:

- Der Zugriff auf den Befehlsspeicher wird von der Simulation nicht berücksichtigt. Datenspeicherzugriffe werden mit einer festen Speicherzugriffszeit versehen. Eine Speicherhierarchie aus Cache- und Hauptspeicher wird nicht nachgebildet.
- Pro Befehlsausführung wird ein Simulationstakt gerechnet.
- Für den Zugriff auf einen FIFO-Speicher und die Mindestverweildauer in den FIFO-Speichern wird ein Simulationstakt angenommen.

Als Simulationsparameter sind berücksichtigt:

- die Kontextwechselzeit, diese berechnet sich aus der Anzahl der Takte, die zum Auffüllen der Befehls-Pipeline mit einem neuen Kontrollfaden nötig sind,
- die Speicherzugriffszeit, welche die Zeitdauer angibt, die von einer Lade- (oder Speicheranforderung) durch die Lade-/Speichereinheit bis zur Bereitstellung des Speicherwortes im Register (bzw. der Speicherbestätigung) vergeht, und
- die Zykluszeit, welche die minimale Zeitdauer angibt, die zwischen zwei aufeinanderfolgenden Lade- oder Speicheranforderungen vergehen muß.

Die Simulationen wurden mit einer Klasse von synthetischen Lastprogrammen durchgeführt, deren Elemente sich durch die folgenden Parameter unterscheiden:

- der Anzahl aller Befehle,
- der Anzahl der unabhängigen Kontrollfäden,
- dem Verhältnis der Folgen aus Lade-/Speicherbefehlen zu Folgen aus Ausführungsbefehlen und
- durch die Anzahl der innerhalb eines Befehlsstromes unabhängigen Folgebefehle.

Für die Diagramme im nächsten Abschnitt wurde ein Lastprogramm ausgewählt, das aus ca. 100 000 Befehlen besteht, welches drei Kontrollfäden und ein Verhältnis der Lade-/Speicherbefehle zu Ausführungsbefehlen von eins zu drei aufweist. Die Anzahl der datenunabhängigen Folgebefehle sei zwei.

Variiert wird in den folgenden Simulationen die Kontextwechselzeit, die Speicherzugriffszeit und die Zykluszeit. In den Diagrammen ist jeweils die Anzahl der Simulationstakte, die für das Ausführen des Simulationsprogramms benötigt werden, als Maß für die Leistung der Prozessorarchitektur aufgetragen.

Grundlegende Simulationen

Das erste Experiment wurde mit einer sehr einfachen Implementierung der Lade-/Speichereinheit vorgenommen: Ein Speicherzugriff wird angefordert und dessen vollständige Ausführung abgewartet, bevor der nächste Befehl (Lade-/Speicherbefehl irgendeines Kontrollfadens oder Ausführungsbefehl desselben Kontrollfadens) ausgeführt wird. In Diagramm 1 werden verschiedene Kontextwechselzeiten für den vielfädigen Prozessor angenommen und die Ausführungszeiten mit derjenigen des konventionellen Prozessors verglichen.

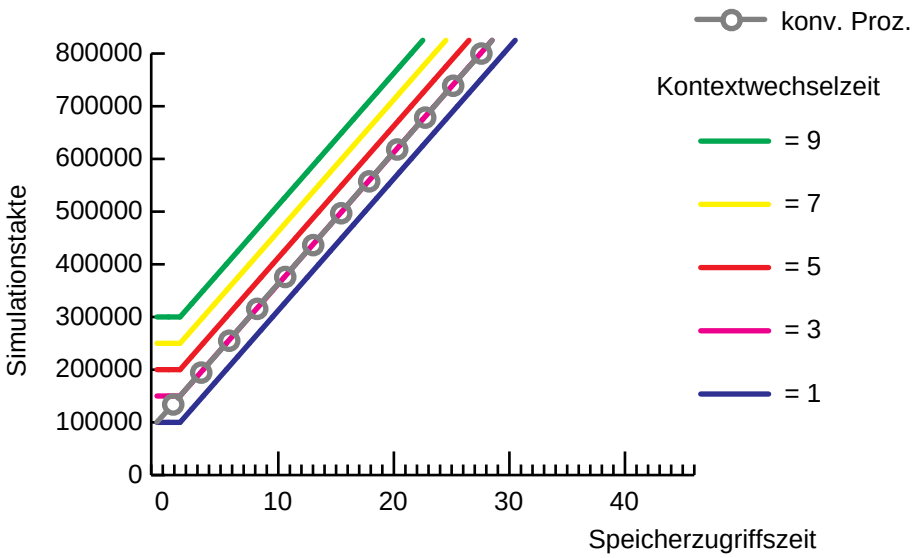

Diagramm 1: Lade-/Speichereinheit wartend

Sind genügend Kontrollfäden vorhanden, d.h. Anzahl der Threads ≥ 3, so ist der vielfädige Prozessor in der Lage, einen geringen Teil der Speicherzugriffszeit zu überbrücken (waagerechte Abschnitte der Kurve). Jedoch verläuft die Kurve des vielfädigen Prozessors rasch parallel zur Kurve des konventionellen Prozessors. Mit der einfachen Lade-/Speichereinheit ist es dem vielfädigen Prozessor für große Speicherzugriffszeiten unmöglich, Wartezeiten durch Kontextwechsel zu überbrücken. Es zeigt sich sogar, daß ab einer Kontextwechselzeit von drei der vielfädige Prozessor nicht mehr schneller als der Referenzprozessor ist.

Durch Erhöhung der Anzahl der Threads kann die Leistung des Prozessors nicht erhöht werden, da die Lade-/Speichereinheit bei einem Verhältnis von einem Lade-/Speicherbefehl zu drei Ausführungsbefehlen schon ausgelastet ist.

Simulationen zur Optimierung der Lade-/Speichereinheit

Um die Frage beantworten zu können, ob der Engpaß Lade-/Speichereinheit vermieden werden kann, wurden vier Implementierungsvarianten für die Lade-/Speichereinheit simuliert:

(i) *Wartend* (entsprechend der Simulationen in Diagramm 1): Ein Lade-/Speicherzugriff wird abgesetzt und auf die Abgeschlossenheit der Ausführung gewartet, bevor der nächste Befehl ausgeführt wird.

(ii) *Umschaltend/Wartend:* Nach dem Absetzen eines Lade-/Speicherzugriffs wird innerhalb der Lade-/Speichereinheit ein Kontextwechsel vorgenommen (Cycle-by-Cycle-Interleaving-Technik in der Lade-/Speichereinheit). Der Folgebefehl wird erst zur Ausführung freigegeben, wenn der Speicherzugriff beendet ist.

(iii) *Überlappend/ohne Umschalten:* Nach dem Absetzen eines Lade-/Speicherzugriffs werden im Zeitschatten des Speicherzugriffs Folgebefehle (Lade-/Speicher- oder Ausführungsbefehle) ausgeführt, die unabhängig von diesem Speicherzugriff sind.

(iv) *Umschaltend/Überlappend* (Kombination aus (ii) und (iii)): Ein Lade-/Speicherzugriff wird abgesetzt und danach im Schatten des Speicherzugriffs weitere Befehle abgearbeitet. Falls ein Folgebefehl datenabhängig ist und noch weitere Takte auf das Eintreffen des Bestätigungssignals gewartet werden muß, wird innerhalb der Lade-/Speichereinheit und analog auch in der Ausführungseinheit ein Kontextwechsel durchgeführt.

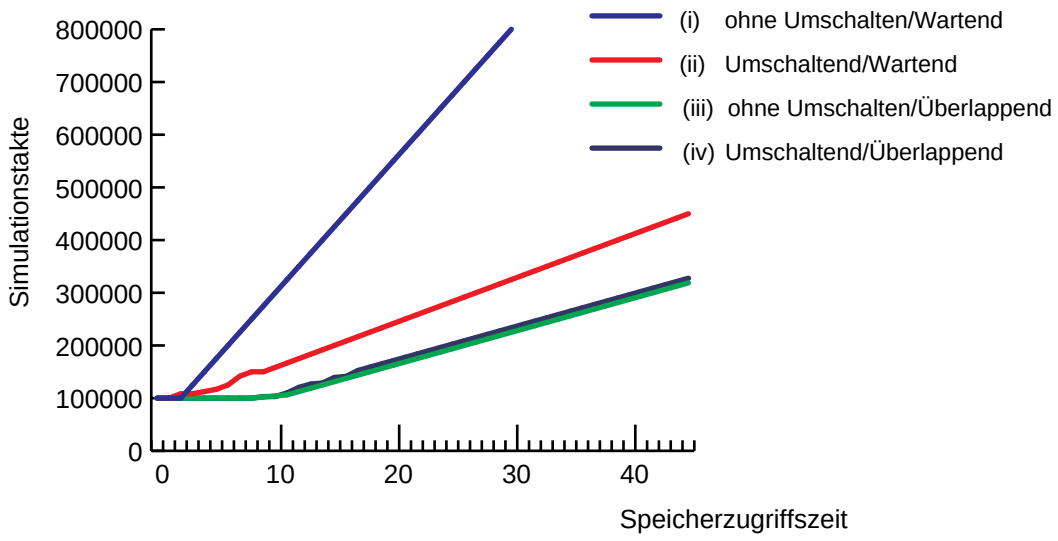

Diagramm 2: Verschiedene Implementierungen der Lade-/Speichereinheit

Diagramm 2 gibt die Ergebnisse der Simulationen der Varianten der Lade-/Speichereinheit bei einer Kontextwechselzeit von einem Takt wieder. In den verschiedenen Experimenten zeigt sich, daß die umschaltende Implementierung (iii) und die Kombinationslösung (iv) in etwa gleichwertig sind. Die Versionen (ii) und (iii) unterscheiden sich folgendermaßen:

- Version (ii) benötigt nicht wesentlich mehr Hardware als die Version (i). Dafür benötigt (ii) wesentlich mehr Kontrollfäden als die Versionen (i) und (iii). Die Anzahl der Threads sollte mindestens gleich der Latenzzeit in Prozessortakten sein. Der leichte Hügel am Anfang der Kurve (ii) ist auf den konkurrierenden Zugriff der Lade-/Speicher- und der Ausführungseinheit zurückzuführen.

- Mit Version (iii) können auch kleine Kontextwechselzeiten überbrückt werden. Die Versionen (iii) und (iv) nützen die Zeit, die die Thread-Kennungen im FIFO-Puffer verweilen zur Überbrückung der Speicherzugriffszeit. Deshalb kann die Anzahl der Befehle, die im Zeitschatten der Speicherbefehle ausgeführt werden, auch gering sein oder völlig fehlen, sofern durch genügend andere Kontrollfäden die FIFO-Verweildauer im Durchschnitt größer als die Speicherzugriffszeit ist.

Simulationen mit überlappend arbeitender Lade-/Speichereinheit

In Diagramm 3 sind die analogen Simulationen zu Diagramm 1 mit der überlappenden Implementierung der Lade-/Speichereinheit (Version (iii)) aufgetragen. Es zeigt den erzielbaren Gewinn, der beim vielfädigen Prozessor durch die Überdeckung der Speicherzugriffszeit durch Ausführungsbefehle eines anderen Thread erzielt werden kann. Man sieht deutlich eine Schere zwischen den Graphen des konventionellen Prozessors und des vielfädigen Prozessors. Die Öffnung der Schere hängt wesentlich von der Zykluszeit des Speichersystems ab. Für Speicherzugriffszeiten bis etwa dreißig Takte ist der Gewinn ist merklich größer, je kleiner die Kontextwechsel-zeit ist. Die Kontextwechselzeit muß dazu kleiner als die Hälfte der Speicherzugriffszeit sein.

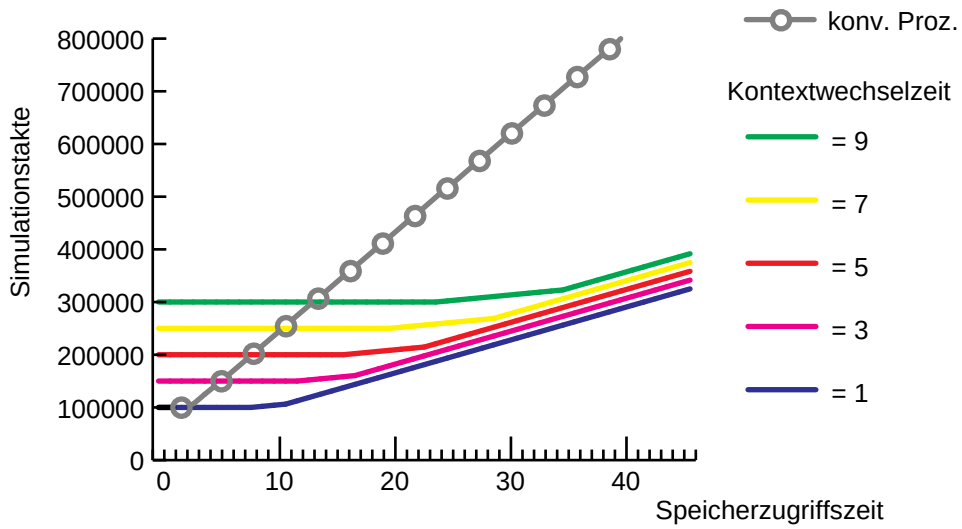

Diagramm 3: Lade-/Speichereinheit arbeitet überlappend

Der Knick in der Kurve des konventionellen Prozessors bei Speicherzugriffszeit drei und ein Vergleich mit Diagramm 1 zeigen, daß auch der konventionelle Prozessor durch eine überlappende Ausführung fähig wird, kleine Speicherzugriffszeiten zu überbrücken. Der vielfädige Prozessor profitiert jedoch stärker von dieser Hardware-Maßnahme.

Um große Wartezeiten vollständig überbrücken zu können, muß auch die Anzahl der geladenen Threads erhöht werden, da sonst nicht genügend Ausführungsbefehle zur Wartezeitüberbrückung zur Verfügung stehen. Das optimale Verhältnis der Anzahl geladener Kontrollfäden zu Speicherzugriffszeit entspricht für sehr kleine Kontextwechselzeiten in etwa dem Verhältnis Lade-/Speicherbefehle zu Ausführungsbefehlen.

Minimierung der Kontextwechselzeit

Demzufolge ist ein Hardware-Optimierungsziel, den Kontextwechselaufwand zu minimieren. In der oben beschriebenen Prozessorarchitektur errechnet sich auf der Grundlage einer fünfstufigen Prozessor-Pipeline (Befehl

laden, Decodieren, Register laden, Ausführen, Register rückschreiben) der Kontextwechselaufwand in der Ausführungseinheit (analog für die Lade-/Speichereinheit) folgendermaßen: Beim Decodieren wird erkannt, daß es sich um einen Lade-/Speicherbefehl handelt. Durch das Decodieren eines Befehls, der auf der Einheit nicht ausgeführt werden kann, wird ein Takt verschenkt. Eine neue Thread-Kennung wird aus dem FIFO-Speicher geholt. Für das Bereitstellen der neuen Thread-Kennung wird ein weiterer Takt angesetzt. Ein explizites Umschalten der Registersätze ist nicht nötig, da die Referenz in der Thread-Kennung steht und durch die Befehls-Pipeline hindurchgereicht werden kann. In einem weiteren Takt muß der Befehlszähler geladen werden. Um den ersten Befehl des neuen Thread bis zur Decodierstufe der Pipeline zu laden, vergehen zusätzlich zwei Takte. Insgesamt ergibt sich eine Verzögerung von fünf Takten.

Codiert man den Kontextwechsel in den Vorgängerbefehl, das ist der letzte Befehl der noch auf dieser Einheit ausgeführt wird, kann das unnötige Decodieren des Befehls eingespart werden. Unter der Annahme von genügend Last, d.h. daß immer eine Thread-Kennung zum Laden zur Verfügung steht, kann auf diese Thread-Kennung im voraus zugegriffen und der Befehlszähler vorzeitig geladen werden. Verdoppelt man die Befehlsladestufe, kann der Befehl vorabgeladen werden. Somit bleibt noch ein Prozessortakt für den Kontextwechsel übrig. Falls auch die Decodierstufe verdoppelt wird, kann der Kontextwechselaufwand bei genügend Last sogar auf Null reduziert werden.

Man beachte jedoch, daß gegenüber dem konventionellen Prozessor die Lade- und die Decodierstufen durch deren getrenntes Auftreten in der Ausführungs- und der Lade-/Speichereinheit im vielfädigen Prozessor bereits verdoppelt wurden. Eine weitere Verdopplung dieser Stufen führt zu einem erheblichen Hardware-Mehraufwand, insbesondere, da diese Stufen im Gegensatz zu deren Vervielfachung bei superskalaren Prozessoren an verschiedenen Threads arbeiten müssen. Das bedeutet, daß für die zwei bis vier Befehlsladestufen auch entsprechend viele Zugänge zum Programmspeicher oder eine genügend große Speicherbandbreite mit Arbitrierung vorgesehen sein müssen.

Obige Experimente zeigen neben der prinzipiellen Fähigkeit des vielfädigen Prozessors Wartezeiten zu überbrücken, daß die Lade-/Speichereinheit der eigentliche Engpaß der Architektur ist. Dieser Engpaß kann gemildert werden, sofern die Zykluszeit sehr klein ist.

Zusammenfassung und Ausblick

Der vielfädige Prozessor zeigte seine Fähigkeit, auch kurze Wartezeiten zu überbrücken, sofern genügend Threads als Last zur Verfügung stehen. *Es zeigte sich, daß nicht die Speicherzugriffszeit und Zykluszeit, wie für konventionelle Prozessoren die Leistungssteigerung des vielfädigen Prozessors begrenzen, sondern ausschließlich die Zykluszeit.* Weitere Arbeiten sollen klären, inwieweit die Zykluszeit mit Cache-Techniken verringert werden kann.

Eine synthetisierbare VHDL–Implementierung soll klären, welche Hardware-Optimierungen für den Prozessor möglich sind. Es sollen Hardware-Grenzen gefunden und deren Kosten mit der Leistungsfähigkeit des Prozessors verglichen werden.

Literatur

[1] K. Bittnar, W. Grünewald, T. Ungerer: Entwurf einer vielfädigen Prozessorarchitektur zum Einsatz in Distributed-Shared-Memory-Systemen. PARS-Workshop, Potsdam, 19.-20. September 1994, Seite 85 - 94.

[2] B. J. Smith: The Architecture of HEP. In: J. S. Kowalik (Hrsg.): Parallel MIMD Computation: The HEP Supercomputer and Its Applications. The MIT Press, Cambridge, Mai. 1985.

[3] M. R. Thistle, B. J. Smith: A Processor Architecture for Horizon. Supercomputing 88, Orlando1988, Seite 35 - 41.

[4] R. Alverson et al.: The Tera Computer System. 4th International Conference on Supercomputing, Amsterdam, 11.-15. Juni 1990, Seite 1- 6.

[5] A. Agarwal: Performance Tradeoffs in Multithreaded Processors. IEEE Transactions on Parallel and Distributed Systems, Band 3, Heft 5, September 1992, Seite 525 - 539.

[6] A. Agarwal et al.: The MIT Alewife Machine: Architecture and Performance. The 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, 22.-24. Juni 1995, Seite 2 - 13.

[7] J. Laudon, A. Gupta, M. Hohowitz: Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, 4.-7. Oktober 1994, Seite 308-318.

[8] D. E. Tullsen, S. J. Eggers, H. M. Levy: Simultaneous Multithreading: Maximizing On-Chip Parallelism. The 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, 22.-24. Juni 1995, Seite 392 - 403.

[9] R. S. French, M. S. Lam, J. R. Levitt, K. Olukotun: A General Method for Compiling Event-Driven Simulations. The 32nd ACM/IEEE Design Automation Conference, Juni 1995

[10] J. L. Hennessy, D. A. Patterson: Computer Architecture a Quantitative Approach, San Mateo1990.

[11] B. Grünewald: Ein Modula-2 Compiler für eine erweiterte DLX-Architektur, Diplomarbeit, Mathematisch-Naturwissenschaftliche Fakultät, Universität Augsburg, 1995