Entwurf einer vielfädigen Prozessorarchitektur zum Einsatz in Distributed-Shared-Memory-Systemen

Karl Bittnar, Winfried Grünewald, Theo Ungerer

Institut für Rechnerentwurf und Fehlertoleranz
Universität Karlsruhe
76128 Karlsruhe

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

Die vielfädige Prozessorarchitektur überbrückt Wartezeiten, die bei entfernten Speicherzugriffen oder bei Synchronisationen entstehen, durch einen schnellen Wechsel des Kontrollfadens (Threadwechsel). Voraussetzung dafür sind eine Anzahl von Registersätzen auf dem Prozessor. Synchronisationen werden von einer Synchronisationseinheit ausgeführt. Die Lade-/Speicheroperationen und die Ausführungsoperationen eines Kontrollfadens sind voneinander entkoppelt und werden von getrennten Einheiten ausgeführt.

1. Einleitung

Zur Zeit werden fast ausschließlich Standardmikroprozessoren als Knoten für Multiprozessorsysteme eingesetzt. Solche Standardprozessoren wurden jedoch für Mikrocomputer oder für Workstations mit einem Prozessor oder mit einer geringen Anzahl von Prozessoren entwickelt und optimiert. Sie sind deshalb nur bedingt zum Einsatz in größeren Multiprozessorsystemen geeignet. Spezielle Architekturtechniken erlauben es jedoch, Prozessoren zu entwickeln, die für den Einsatz in Multiprozessorsystemen besser geeignet sind.

Ziel unseres Forschungsprojektes ist es, Architekturtechniken für Prozessoren zu entwickeln, die als Knoten in Distributed-Shared-Memory-Systemen (DSM) effizienter als Standardmikroprozessoren einsetzbar sind. DSM-Systeme [1, 2] besitzen physikalisch verteilte Speicher, jedoch einen gemeinsamen Adreßraum. Der gemeinsame Adreßraum erlaubt eine einfachere Programmierung und eine leichtere Nutzung feinkörniger Parallelität als bei nachrichtengekoppelten Systemen. Andererseits ermöglichen die physikalisch verteilten Speicher eine Skalierung bis zu Tausenden von Prozessoren.

Bei der Prozessorentwicklung für DSM-Systeme müssen die Wartezeiten bei Zugriffen auf entfernte Daten überbrückt und eine effiziente Synchronisation der Kontrollfäden ermöglicht werden. Die Wartezeiten bei Speicherzugriffen sind in der Regel begrenzt und können durch Threadwechsel überbrückt werden, sofern diese sehr schnell durchgeführt werden. Die beliebig langen Wartezeiten, die bei der Synchronisation der Kontrollfäden auftreten, können ebenfalls durch Kontollfadenwechsel aufgefüllt werden. Weiterhin benötigt man schnelle, d.h. Hardware-implementierte, Synchronisationsprimitive, die ohne aktives Warten arbeiten. Die Forderung nach einem sehr schnellen Threadwechsel bedeutet in letzter Konsequenz, daß dieser möglichst in einem Prozessortakt abgeschlossen sein sollte.

Vergleichbare Ansätze, die ebenfalls durch Threadwechsel versuchen, das Synchronisationsproblem zu lösen und die Wartezeiten bei Zugriffen auf entfernte Daten zu überbrücken, sind die APRIL-Architektur [3], der daraus entwickelte Sparcle-Prozessor [4], die *T-Architektur [5], der daraus entwickelte Motorola 88110MP-Prozessor [6], der Multithreaded-Prozessor des Media Research Laboratory der Matsushita Electric Industrial Co. [7] und diverse Datenflußarchitekturen [8].

2. Die Prozessorarchitektur

Wir betten die Prozessorarchitektur in ein DSM-System ein. Die Prozessoren teilen sich einen gemeinsamen Adreßraum und greifen mit unterschiedlichen Antwortzeiten auf verschiedene Speicherbereiche zu. Die Hauptidee der Prozessorarchitektur ist, alle Operationen, die zu aktivem Warten führen können, von der Ausführungseinheit (execution unit) des Prozessors fernzuhalten. Deshalb werden Lade-, Speicher- und Synchronisationsbefehle von anderen Einheiten des Prozessors bearbeitet. Die Wartezeiten bei Speicherzugriffen hängen vom Netzwerk und vom Speicher ab, sind aber abschätzbar. Dagegen können bei Synchronisationsoperationen keine Aussagen über die Dauer der Wartezeit getroffen werden, weil diese vom Anwendungsprogramm bestimmt sind. Aus diesem Grund haben wir eine Einheit für die Speicheroperationen (store/load unit) und eine Einheit für die Synchronisationsoperationen (sync unit) vorgesehen.

Da jede Einheit des Prozessors einen anderen Thread bearbeitet, können die einzelnen Einheiten asynchron zueinander arbeiten. Sie sind durch FIFO-Puffer miteinander verbunden und greifen auf eine Anzahl von gemeinsamen Registersätzen (register frames) zu. Der Architekturentwurf benutzt keinerlei Annahmen über das Netzwerk, mit dem der Prozessor mit dem Speicher und mit anderen Prozessoren verbunden ist. Aus Effizienzgründen ist der Einsatz von Cachespeichern denkbar. Das Blockschaltbild der vielfädigen Prozessorarchitektur in Abb. 1 gibt die Datenpfade des Prozessors wieder.

Jedem Thread wird eineindeutig eine Threadkennung (thread tag) und ein Aktivierungsrahmen zugeordnet, in dem Thread-lokale Daten abgelegt werden. Dazu gehören unter anderem der Befehlszähler, die eigene Threadkennung, die Prozeßkennung und verschiedene andere Zustandsinformationen. Die logischen Aktivierungsrahmen werden physikalisch auf die Registerrahmen verteilt. Sind mehr Aktivierungsrahmen als Registerrahmen vorhanden, werden Aktivierungsrahmen von wartenden Kontrollfäden in den Speicher ausgelagert.

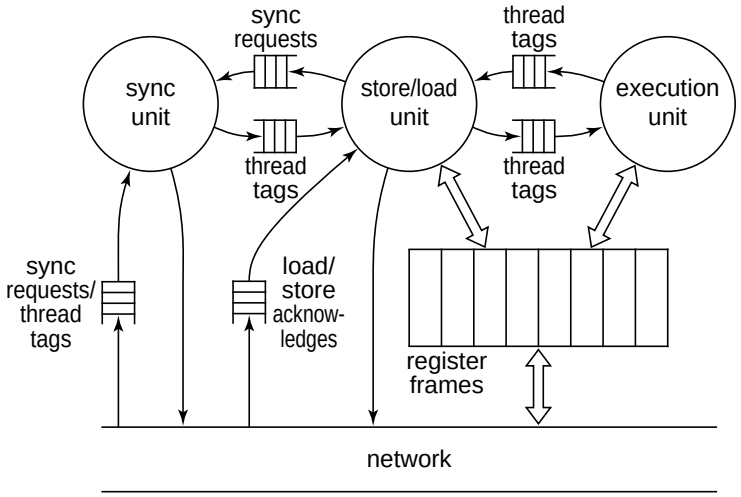

Abb. 1: Blockschaltbild der vielfädigen Prozessorarchitektur

Die Einheiten arbeiten an verschiedenen Kontrollfäden und somit auf verschiedenen Aktivierungsrahmen, was wiederum verschiedene Registerrahmen erzwingt. Der Registerrahmen bestimmt vollständig den Bearbeitungszustand des Thread. Somit kann ein schneller Kontextwechsel durch einfaches Umschalten auf einen anderen Registerrahmen erreicht werden. Ein Kontextwechsel findet statt, wenn eine Einheit auf einen Befehl trifft, den sie nicht bearbeiten kann und den sie deshalb an eine andere Einheit weitergeben muß, oder wenn ein Befehl explizit einen Kontextwechsel verlangt. Ist die Bearbeitung für einen Thread abgeschlosssen, so wird eine neue Threadkennung aus einem der FIFO-Puffer entnommen und bearbeitet.

Speicherkonsistenzmodelle [9, 10] fordern, daß Speicherzugriffe zu festgelegten Synchronisationszeitpunkten abgeschlossen sind. Nur die Speicher-/Ladeeinheit besitzt das nötige Wissen, da sie für jeden Speicherzugriff eine Bestätigung erhält. Deshalb ist die Speicher-/Ladeeinheit vor der Synchronisationseinheit plaziert. Die sequentielle Ausführungsreihenfolge durch die Synchronisationseinheit und die Technik, in einem DSM-System jede Synchronisationsvariable eindeutig an eine Synchronisationseinheit zu binden, ermöglichen es, sequentielle Konsistenz [11] für die Synchronisationsoperationen zu garantieren. Die Realisierung verschiedener Konsistenzmodelle unterscheidet sich in der Ausprägung der Speicher-/Ladeeinheit.

3. Detailbeschreibung der Einheiten

Für diese Beschreibung nehmen wir an, daß der Code auf jedem Prozessor bereits geladen ist. Das Blockschaltbild aus Abb. 1 definiert die Verbindungsstruktur zwischen den diversen Einheiten. Es ist daher noch das Ein-/Ausgabeverhalten jeder einzelnen Einheit zu beschreiben. Dazu wird eine Modula-2 ähnliche Spezifikationssprache verwendet.

3.1 Rahmen, Befehlsformat und Speicherschnittstelle

Der Satz von Aktivierungsrahmen wird als zweidimensionales globales Feld `R` modelliert. Der erste Index gibt die Zuordnung von Thread(-kennung) `F` zu Aktivierungsrahmen wieder. Der zweite Index adressiert ein Register innerhalb des Rahmens.

```
SHARED  R     : ARRAY THREADTAG, REGISTER OF WORD;
```

Die Befehle für die Ausführungs- und die Speicher-/Ladeeinheit sollen in dekodierter Form vorliegen. Das Tupel `(op, r1, r2, r3, imm)` stellt einen Befehl dar, wobei `op` die Operation, `r1`, `r2` und `r3` die beteiligten Register und `imm` eine Konstante bezeichnet.

Das folgende Programm beschreibt die Speicherschnittstelle. Es werden keine Annahmen über die Reihenfolge bei der Bearbeitung von Speicheroperationen bzw. über die Topologie des Netzes getroffen. Bei den Speicheroperationen werden zwei Arten unterschieden. Erhält der Speicher eine Ladeanforderung `loadRequ` für die Adresse `adr` von einer Speicher-/Ladeeinheit, dann wird der Wert `storage[adr]` zusammen mit einer Ladebestätigung `loadAckn` an dieselbe Speicher-/Ladeeinheit zurückgeschickt. Dagegen wird bei einer Speicheranforderung `storeRequ` der Wert in die Speicherstelle `storage[adr]` eingetragen und mit `storeAckn` bestätigt.

```
LOCAL storage : ARRAY WORD OF WORD;

UNIT memory;
BEGIN
  LOOP
    take (F,code,adr,data,reg) from one store/load
    CASE code OF
    | loadRequ:     hand (F,loadAckn,adr,storage[adr],reg) over to that store/load
    | storeRequ:    storage[adr] := data;
                    hand (F,storeAckn) over to that store/load
    END;
  END;
END memory;
```

Abb. 2: Die Speicherschnittstelle

3.2 Die Ausführungseinheit

Die Ausführungseinheit holt sich aus dem Puffer von der Speicher-/Ladeeinheit eine Threadkennung `F`. Anschließend wird der nächste Befehl des Thread geladen und dekodiert. Dabei steht die Adresse des Befehls im Register `PCReg` des Aktivierungsrahmens `F`. Trifft die Einheit auf eine arithmetische Operation (`add`, `sub`, `and`, ... , `srli`), wird die Berechnung ausgeführt und das Ergebnis in das Zielregister geschrieben. Die Befehle `mvr` und `mvl` dienen zum Austausch von Registerinhalten mit einem anderen Aktivierungsrahmen. Der Befehl `beqz` realisiert den bedingten Sprung.

```
SHARED  R       : ARRAY THREADTAG, REGISTER OF WORD;
LOCAL   Free    : SET OF THREADTAG;

UNIT execution;
BEGIN
   LOOP
      take F from store/load;
      LOOP
         pc := R[ F, PCReg ];
         (op, r1, r2, r3, imm) := load instruction at pc;
         pc := next pc;
         CASE op OF
         |  add  .. srl:     R[ F, r3 ] := R[ F, r1 ] op R[ F, r2 ];
         |  addi .. srli:    R[ F, r2 ] := R[ F, r1 ] op imm;
         |  mvr:             R[ R[ F, r3 ], r2 ] :=  R[ F, r1 ];
         |  mvl:             R[ F, r3 ] := R[ R[ F, r2 ], r1 ];
         |  beqz:            IF R[ F, r1 ] = Null THEN
                                pc := R[ F, r2 ] + imm;
                            END;
         |  start:           hand R[ F, r1 ] over to store/load;
         |  yield:           R[ F, PCReg ] :=  pc;
                            F := R[ F, r1 ];
                            pc := R[ F, PCReg ];
         |  stop:            R[ F, PCReg ] :=  pc;
                            EXIT inner loop;
         |  relfr:           add R[ F, r1 ] to Free;
         |  getfr:           pick one G out of Free;
                            R[ F, r1 ] := G;
                            R[ G, FrReg ] := G;
                            R[ G, PCReg ] := R[ F, r2 ] + imm;
                            R[ G, CtReg ] := 0;
         |  lw, sw,
            lock, unlock:   hand F over to store/load;
                            EXIT inner loop;
         END;
         R[ F, PCReg ] := pc;
      END
   END
END execution;
```

Abb. 3: Die Ausführungseinheit

Der Befehl `start` aktiviert den im Befehl angegebenen Kontrollfaden, indem dessen Kennung an die Speicher-/Ladeeinheit gegeben wird. Der Befehl `yield` ersetzt in der Ausführungseinheit den laufenden Thread durch den im Befehl spezifizierten Thread. Mit den Befehlen `getfr` und

`relfr` wird ein Aktivierungsrahmen erzeugt und initialisiert bzw. freigegeben. Nach der Abarbeitung des Befehls auf der Ausführungseinheit wird der nächste Befehl des Thread geladen.

Der Befehl `stop` suspendiert den aktuellen Kontrollfaden. Anschließend holt sich die Ausführungseinheit die nächste Threadkennung. Die Befehle `lw`, `sw`, `lock` und `unlock` werden nicht von der Ausführungseinheit bearbeitet. Stattdessen wird der aktuelle Thread genauso wie beim `stop`-Befehl beendet, aber zusätzlich wird die Threadkennung an die Speicher-/Ladeeinheit weitergegeben.

3.3 Die Speicher-/Ladeeinheit

```
SHARED R   : ARRAY THREADTAG, REGISTER OF WORD;

UNIT store/load;
BEGIN
   LOOP
      take (F, code, a, data, reg) from sync, execution or memory;
      CASE code OF
         |  loadAckn:  R[ F, reg ] := data;
                       DEC ( R[ F, CtReg ] );
         |  storeAckn: DEC ( R[ F, CtReg ] );
      END;
      LOOP
         pc := R[ F, PCReg ];
         (op, r1, r2, r3, imm) := load instruction at pc;
         pc := next pc;
         CASE op OF
         |  add..getfr:   IF R[ F, CtReg ] = Null THEN
                            hand F over to execution;
                          END;
                          EXIT inner loop;
         |  lw:           INC( R[ F, CtReg ] );
                          hand (F, loadRequ, R[F, r1]+imm, r2) over to memory;
         |  sw:           INC( R[ F, CtReg ] );
                          hand (F, storeRequ, R[F,r2]+imm, R[F,r1]) over to memory;
         |  lock:         IF R[ F, CtReg ] = Null THEN
                            hand (F, lockRequ, R[F, r1]+imm) over to sync;
                            R[ F, PCReg ] := pc;
                          END;
                          EXIT inner loop;
         |  unlock:       IF R[ F, CtReg ] = Null THEN
                            hand (F, unlockRequ, R[F, r1]+imm) over to sync;
                          ELSE
                            EXIT inner loop;
                          END;
         END;
         R[ F, PCReg ] := pc;
      END
   END
END store/load;
```

Abb. 4: Die Speicher-/Ladeeinheit

Für jede Speicheroperation wird eine Anforderungsnachricht (`loadRequ` oder `storeRequ`) an die Speicherschnittstelle geschickt, die vom Speicher durch eine Bestätigungsnachricht

(`loadAckn` bzw. `storeAckn`) beantwortet wird. Es werden solange Anforderungen abgeschickt, bis die Einheit auf einen Ausführungs- oder Synchronisationsbefehl trifft. Dann wird der Thread gewechselt und bei der nächsten Bestätigung wieder aufgenommen. Dazu wird die Threadkennung der Nachricht mitgegeben.

Durch die Bestätigungen der Speicheroperationen ist die Grundlage zur Unterstützung verschiedener Konsistenzmodelle durch die Prozessorarchitektur geschaffen. Es ist jedoch prinzipiell nicht möglich, ein Konsistenzmodell für ein System ausschließlich über den Prozessorentwurf zu realisieren. Es müssen immer zusätzliche Voraussetzungen an die Verbindungsstruktur erfüllt sein. Das von uns vorgesehene Konsistenzmodell fordert, daß vor der Abarbeitung eines Ausführungs- oder eines Synchronisationsbefehls alle Speicheroperationen des Kontrollfadens abgeschlossen, also die entsprechenden Bestätigungen eingetroffen sind. Das Register `CtReg` im Aktivierungsrahmen des Kontrollfadens zählt die ausstehenden Bestätigungen. Sind alle Anforderungen beantwortet (`CtReg = Null`), wird bei einem Ausführungsbefehl die Threadkennung an die Ausführungseinheit und bei einem Synchronisationsbefehl eine Synchronisationsanforderung an die Synchronisationseinheit weitergegeben. Der Kontext wird gewechselt außer beim `unlock`-Befehl, der bei unserer Implementierung die sofortige Abarbeitung nachfolgender Speicheroperationen zuläßt, um den Thread nicht zu blockieren.

3.4 Die Synchronisationseinheit

Wir unterstützen die Synchronisationsprimitive `lock` und `unlock` auf Schloßvariablen. Jede Schloßvariable existiert genau einmal im DSM-System und ist fest an eine Synchronisationseinheit gebunden. Deswegen werden Synchronisationsanforderungen (`lockRequ` oder `unlockRequ`) für Schloßvariablen von anderen Prozessoren an die zuständige Synchronisationseinheit verschickt.

```
LOCAL   mutex:     ARRAY WORD OF ( locked, unlocked );
        waiting:   ARRAY WORD OF SET OF THREADTAG;

UNIT sync;
BEGIN
   LOOP
      take (F, code, var) from other sync or store/load;
      CASE code OF
      | lockRequ:      IF var at this sync THEN
                          IF mutex[var] = locked THEN
                             add F to waiting[var];
                          ELSE
                             mutex[var] := locked;
                             IF (F, code, var) from store/load THEN
                                hand F over to store/load;
                             ELSE
                                hand (F, lockAckn, var) over to other sync;
                             END
                          END
                       ELSE
                          hand (F, code, var) over to other sync;
                       END;
```

```
      | unlockRequ:    IF var at this sync THEN
                         IF mutex[var] = locked THEN
                           IF waiting[var] = {} THEN
                             mutex[var] := unlocked;
                           ELSE
                             pick F out of waiting[var];
                             IF (F, code, var) from store/load THEN
                               hand F over to store/load;
                             ELSE
                               hand (F, lockAckn, var) over to other sync;
                             END
                           END
                         ELSE
                           hand (F, code, var) over to other sync;
                         END;
      | lockAckn:      hand F over to store/load;
      END
    END
  END sync;
```

Abb. 5: Die Synchronisationseinheit

Die Synchronisationseinheit prüft bei einer `lock`-Anforderung für eine eigene Schloßvariable var, ob die Variable gesperrt ist. In diesem Fall wird die Threadkennung `F` in die Menge der auf diese Schloßvariable wartenden Kontrollfäden aufgenommen. Andernfalls wird die Variable gesperrt und die Threadkennung durch ein `lockAckn` an den Auftraggeber zurückgeschickt. Anschließend wird die nächste Synchronisationsanforderung bearbeitet.

Bei einer `unlock`-Anforderung wählt die Synchronisationseinheit eine Threadkennung aus den wartenden Kontrollfäden aus und schickt ein `lockAckn` an den Prozessor zurück, der den Kontrollfaden bearbeitet. Existiert kein wartender Thread, so wird die Schloßvariable freigegeben.

4. Software- und Compilerauswirkungen

Um Leistungseinbußen, die auch bei sehr schnellen Kontextwechseln entstehen, möglichst gering zu halten, ist es erforderlich, die Anzahl der Kontextwechsel zu minimieren. Das läßt sich dadurch erreichen, daß durch geeignete Codeoptimierungen Befehle, die auf derselben Einheit ausgeführt werden können, zu möglichst großen Anweisungsblöcken gruppiert werden.

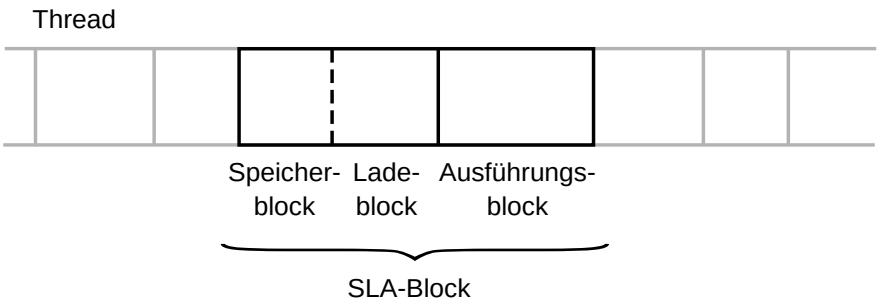

Abb. 6: Aufbau eines Kontrollfadens

Der Compiler untergliedert den Code jedes Kontrollfadens in eine Anzahl von Basisblöcken. Innerhalb eines solchen Blocks werden alle `load`-Anweisungen so weit als möglich nach vorne gezogen und alle `store`-Anweisungen nach hinten geschoben. Idealerweise zerfällt der Code dabei in eine Folge von Speicherblöcken, Ladeblöcken und Ausführungsblöcken. Ein Speicherblock, ein Ladeblock und ein Ausführungsblock werden als SLA-Block bezeichnet. Hierbei können einzelne Blöcke auch leer sein.

Dieses Verfahren wurde in den GNU C-Compiler integriert. Dabei ergab sich eine Häufung der SLA-Blocklängen bei 08/15-Programmen von fünf bis zehn Befehlen — was für unseren Ansatz durchaus akzeptabel ist. Bei numerischen Programmen lassen sich diese SLA-Blöcke beinahe beliebig verlängern.

Der Programmierer kann das Threadmanagement über vordefinierte Bibliotheksfunktionen beeinflussen. Außerdem wird beim Unterprogrammaufruf ein eigener Thread erzeugt. Der Compiler sorgt für die Erzeugung eines neuen Aktivierungsrahmens, schreibt die Parameter in diesen Rahmen und übergibt die Kontrolle an das Unterprogramm. Das Unterprogramm gibt am Ende seiner Ausführung die Kontrolle an den aufrufenden Thread zurück, welcher die Ergebnisse wieder aus dem Aktivierungsrahmen des Unterprogramms lesen kann. Danach wird der Aktivierungsrahmen wieder freigegeben.

5. Stand des Projektes

Unser Ziel ist es, die Leistungsfähigkeit des vielfädigen Prozessors mit der eines konventionellen Prozessors beim Einsatz innerhalb eines DSM-Systems zu vergleichen. Als konventionellen Prozessor benutzen wir die DLX-Prozessorarchitektur [12], die wir um Synchronisationsprimitive erweitert haben. Zur Simulation unserer vielfädigen Prozessorarchitektur verwenden wir ebenfalls den Befehlssatz des DLX-Prozessors, angereichert um Threadmanagement- und Synchronisations-Befehle.

Weiterhin wurden kleine Assemblerprogramme für die beiden Systeme geschrieben und auf dem Simulator ausgeführt. Dabei stellte sich heraus, daß von Hand geschriebene Assemblerprogramme zu kurz sind, um signifikante Testdaten zu erzeugen. Deswegen arbeiten wir an einer vollständigen Anpassung des GNU C-Compilers an die vorgestellte vielfädige Prozessorarchitektur.

6 . Literatur

[1] Lenoski, D., et al.: The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems* 4, 1, Januar 1993, Seite 41–61.

[2] Frank, S., et al.: The KSR1: Bridging the Gap Between Shared Memory and MPPs. *Compcon*, Spring 1993, Seite 285–294.

[3] Agarwal, A., et al.: APRIL: A Processor Architecture for Multiprocessing. *17th Annual International Symposium on Computer Architecture*, Seattle, 1990, Seite 104–114.

[4] Agarwal, A., et al.: Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. *IEEE Micro*, Juni 1993, Seite 48–61.

[5] Nikhil, R. S., Papadopoulos, G. M., Arvind: *T: A Multithreaded Massively Parallel Architecture. *19th International Symposium on Computer Architecture*, 1992, Seite 156–167.

[6] Beckerle, M.J.: Overview of the START (*T) Multithreaded Computer. *Compcon*, Spring 1993, Seite 148–156.

[7] Hirata, H., Kimura, S., Nagamine, S., Mochizuki, Y., Nishimura, A., Nakase, Y., Nishizawa, T.: An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. *19th Annual International Symposium on Computer Architecture*, 1992, Seite 136–145.

[8] Ungerer, T.: Datenflußrechner. Teubner-Verlag, Stuttgart 1993.

[9] Stumm, M., Zhou, S.: Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, Mai 1990, Seite 54–64.

[10] Nitzberg, B., Lo, V.: Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, August 1991, Seite 52–60.

[11] Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9, September 1979, Seite 690–691.

[12] J.Hennessy, J., Patterson, D.: Computer Architecture — A Quantitative Approach. San Mateo, 1990.