Hauptseminar Multicore Programming: Transactional Memory

Christian Vaitl

07 Mai 2009

Inhaltsverzeichnis

1 Das Problem

Bei bisherigen Speicherkonzepten hat ein Prozessor oder ein Programm seine Speicherbereiche stets exklusiv für sich beansprucht. Andere parallel laufende Software konnten darauf nicht zugreifen.

2 Der traditionelle Ansatz

Der traditionelle Ansatz zur Synchronisation nebenläufiger Programme ist die Verwendung von Locks.
Möchte ein Prozess exklusiven Zugriff auf eine Ressource, muss er eine Sperre bei einem Verwaltungsprozess (z.B. einem Locking-Manager) anfordern.

2.1 Arten von Locks

Um die angeforderte Ressource nicht komplett zu sperren gibt es zwei grundlegende Arten von Locks:

- **Read-Lock**: Besitzt eine Ressource einen Read-Lock so möchte der Prozess, der dieses Lock gesetzt hat, von der Ressource nur lesen. Somit können auch andere Prozesse auf diese Ressource lesend zugreifen, diese aber nicht verändern.
- **Write-Lock**: Eine Ressource mit Write-Lock verhindert, dass die Ressource von anderen Prozessen gelesen oder geschrieben wird, da der Prozess, der den Lock gesetzt hat, die Ressource verändern möchte.

2.2 Probleme

- **Deadlocks**: Eine Menge von Prozessen befindet sich in einem Deadlock, wenn jeder dieser Prozesse auf ein Ereignis wartet, das nur ein anderer Prozess aus dieser Menge verursachen kann.
- **Livelocks** bezeichnet eine Art des Deadlocks von zwei oder mehr Prozessen, die im Unterschied zum Deadlock nicht in einem Zustand verharren, sondern ständig zwischen mehreren Zuständen wechseln, aus denen sie nicht mehr entkommen können. Jeder einzelne Prozess verharrt also nicht für immer im wait-Zustand, sondern ist weiterhin aktiv, kann aber auch nicht seine Aufgabe abarbeiten.

  Anschaulich kann man sich dazu zwei Personen vorstellen, die sich in einem Gang entgegenkommen und bis in die Unendlichkeit versuchen, einander in der gleichen Richtung auszuweichen, und sich dabei trotzdem immer gegenseitig blockieren.
- **Problem-Potenziale beim Prozessabbruch**: Werden die Sperren freigegeben? In welchem Zustand befinden sich die gemeinsam genutzten Speicherbereiche?

- **Prioritätsumkehr**: Prozess A mit hoher Priorität kann nicht ausgeführt werden, weil ein Prozess C niedriger Priorität eine benötigte Ressource blockiert. Wird C durch einen anderen Prozess B mittlerer Priorität verdrängt, so kann C die Resource nicht freigaben und B wird vor A ausgeführt. (Lösung: A verebt seine Priorität an C, bis C die Resource freigibt.)
- **Warten trotz höherer Priorität**: Schlecht insbesondere bei Echtzeit-Systemen.
- **Lock Convoy**: Tritt auf, wenn threads mit gleicher priorität wiederholt versuchen das selbe Lock zu bekommen. Anders als beim Deadlock oder Livelock, machen die Threads Fortschritte, aber jedes mal wenn ein Thread versucht das Lock zu bekommen, und nicht erfolgreich ist, gibt der Sheduler auf, und forciert einen Context-Switch. Durch den Overhead der Context-Switches wird die gesamt Performance beeinträchtigt.
- **Zu viele Sperren** (z.B. bei Anfängern, die 'auf Nummer sicher' gehen wollen) führen zu niedriger Geschwindigkeit aufgrund von wartenden Prozessen. Um dies zu vermeiden ist jedoch sehr viel Detailverständnis der Materie von nöten, und es können sehr leicht Fehler passieren (z.B. ein Lock vergessen).

3 Die Alternative: Transactional Memory (TM)

3.1 Definitionen

3.1.1 Was ist eine Transaktion?

Eine Transaktion ist eine Sequenz von Aktionen, die für den außenstehenden Beobachter atomar erscheint. Sie muss drei grundlegende Eigenschaften erfüllen:

- **Atomarität** verlangt, dass die Aktionen einer Transaktion entweder komplett und erfolgreich ausgeführt werden, oder aber bei einem Abbruch keine Spuren davon im System übrig bleiben.
- **Kontinuität** sorgt dafür, dass das System immer in einem konsistenten Zustand ist. Eine Transaktion hat keine Information darüber was für Transaktionen bisher ausgeführt wurden, wenn eine Transaktion startet geht sie davon aus, dass das System in einem konsistentem Zustand ist und in diesem Zustand muss das System auch wieder sein wenn es von der Transaktion verlassen wird.
- **Isolation** erfordert, dass jede Transaktion ein korrektes Resultat produziert, ganz egal wie viele andere Transaktionen parallel dazu ausgeführt werden.

3.1.2 Veschachtelte Transaktionen

Eine verschachtelte Transaktion ist eine Transaktion, die vollständig von einer anderen Transaktion umschlossen wird. Die innere Transaktion sieht die Veränderungen, die von

der äußeren gemacht werden. Für das Verhalten der beiden Transaktionen gibt es mehrere Varianten.

- Falls die Transaktionen **flach** sind, führt der Abbruch der inneren Transaktion auch zu einem Abbruch der äußeren Transaktion. Die Änderungen der inneren Transaktion sind nicht gültig falls die äußere Transaktion nicht erfolgreich abgeschlossen wird.

- Sind die Transaktionen **geschlossen**, führt ein Abbruch der inneren Transaktion nicht automatisch zu einem Abbruch der äußeren Transaktion. Schließt die innere Transaktion ihre Aktionen erfolgreich ab, sind die gemachten Änderungen nur für die äußere Transaktion sichtbar. Für das ganze System ist die Änderung erst sichtbar, wenn die äußerste Transaktion erfolgreich terminiert.

- Im Gegensatz dazu wird bei **offenen** Transaktionen die Änderung der inneren Transaktion sofort nach deren Termination im ganzen System sichtbar. Diese Änderungen bleiben bestehen auch wenn die äußere Transaktion später abgebrochen wird.

3.1.3 Synchonisationsmechanismen

- **Lock-free**: Das Gesamtsystem kann sich nicht aufhängen. Es macht Fortschritte während einer endlichen Anzahl von Schritten der beteiligten Prozesse.(Irgend ein Faden macht immer Fortschritte)

- **Wait-free**: Jeder Faden macht Fortschritte, sogar wenn andere Fäden ihre Ausführung verzögern oder abbrechen.(stärkere Eigenschaft als lock-free)

3.1.4 Compare and Swap (CAS)

CAS ist eine Atomare Operation, die von der Hardware angeboten werden muss. Es ist die Grundlage für die meisten Sperrmachanismen (zumindest der 'lock-free').
Um korrekt zu funktionieren benötigt das CAS drei Argumente:

- **v**: Einen Vergleichswert, der den alten Wert einer Operation repräsentiert.

- **n**: Den neuen Wert der Operation.

- **M**: Adresse der Hauptspeicherstelle aus der v am Anfang der Operation gelesen wurde.

Nach der Operation wird der Wert an M mit v verglichen.

- Stimmen sie überein, wird der neue Wert n auf M geschrieben.

- Stimmen sie nicht überein, wird n verworfen und die Operation noch einmal mit dem neuen v ausgeführt.

Erfolg oder Nichterfolg wird zurückgemeldet, meist durch Rückgabe des vorgefundenen Wertes.

Beispiel: Auszahlung am Geldautomaten, unter der Bedingung dass der Saldo des Kontos nicht negativwerden darf.

- **Traditionelles Vorgehen** : Während des Auszahlvorganges wird eine Sperreauf das Konto angefordert. Andere Geldautomaten müssen warten, wenn sie gleichzeitig auf dasselbe Konto zugreifen wollen.
- **Lock-free-Variante**:
  1. lese den alten Wert, prüfe die Bonität (und breche evtl. ab).
  2. ziehe Auszahlungsbetrag ab.
  3. setze neuen Betrag mittels CAS, falls das fehlschlägt, starte wieder bei (1).

  Dieser Algorithmus ist 'lock-free' aber nicht 'wait-free', denn durch andere Zugriffe auf dieses Konto könnte sich der Saldo stets wieder geändert haben, die CAS-Operation fehlschlagen und so den Auszahlungsvorgang hinauszögern

3.2 Transactional Memory im Allgemeinen

3.2.1 Definition

Mit Transactional Memory bezeichnet man ein neues Hauptspeicherkonzept, das bei Mehrprozessorsystemen zum Einsatz kommen soll. Ziel ist es die Schwierigkeiten der Synchronisierung und Koordination, die bei parallelen Berechnungen entstehen, vom Programmierer in den Compiler und Hardware zu verlagern. Bisher befindet sich Transactional Memory noch im Forschungsstadium. Mit seiner Serienreife wird erst in einigen Jahren gerechnet. Transactional Memory kann entweder komplett in Software (STM), als Hardware (HTM) oder mit Hardware-Unterstützung (Hybrid-HTM) implementiert werden.

3.2.2 Grundprinzip

Im Vergleich zur Benutzung von Locks, die in den meisten modernen Multiprozessor Systemen verwendet werden, ist TM ein sehr *optimistisches* Verfahren. Ein Thread macht die Änderungen am gemeinsamen Speicher, ohne sich groß darum zu kümmern, was andere Threads tun, und zeichnet jeden read- bzw. write-Vorgang in einem Log auf. Nach Beendigung der ganzen Transaktion wird überprüft, ob andere Threads nebenläufig Veränderungen am benutzten gemeinsamen Speicher durchgeführt haben(**Konflikt**). Ist das nicht der Fall, so werden die Veränderung, die der Thread am Speicher getätigt hat, überprüft, und falls erfolgreich, dauerhaft auf den Speicher geschrieben. Das wird **commit** genannt.

Eine Transaktion kann aber auch jederzeit abgebrochen werden, was dazu führt, dass alle vorherigen Veränderungen rückgängig gemacht werden. Falls eine Transaktion wegen Konflikten nicht commited werden kann, so wird üblicherweise die Transaktion abgebrochen (**abort**), und neu gestartet. So lange bis sie erfolgreich ist.

3.2.3 Abbruch einer Transaktion

Bei einem Abbruch müssen alle Effekte einer Transaktion rückgängig gemacht werden. Dazu müssen die schreibenden Zugriffe vorher aber entweder

- Eager Versioned sein:
  - Direkte Aktualisierung der Speicherinhalte
  - Auffangen der überschriebenen Werte in Undo-Log
  - Rückspielen des Logs bei Konflikt
  - Schnelle Commits
  - Langsame Aborts
  - Keine Fehlertoleranz
  - Schwache Atomizität
- Lazy Versioned sein:
  - Pufferung der Schreibzugriffe
  - Aktualisierung der Speicherinhalte aus Puffer bei Commit
  - Verwerfen des Puffers bei Konflikt
  - Langsame Commits
  - Schnelle Aborts
  - Fehlertolerant
  - Starke Atomizität

Ausserdem dürfen während einer Transaktion dürfen keine (direkten) I/O-Operationen ausgeführt werden, da diese nicht rückgängig gemacht werden könnten.

Das sogenannte 'Output-Commit'-Problem, also das Problem, dass man zwar im Falle eines Abbruchs oder Fehlers das System wieder herstellen kann, aber nicht die getätigten 'schlechten' Outputs, kann durch Verzögern der Ausgaben behoben werden.

Das sogenannte 'Input-Commit'-Problem, kann im Allgemeinen dadurch behoben werden, dass man die Eingaben speichert und nach der Wiederherstellung des Systems neu 'abspielt'.

3.2.4 Konfliktüberprüfung

Um Konflikte zu entdecken und zu behandeln, hat ist eine Transaktion normalerweise mit zwei Datensätzen verbunden.

- Read-Set: Hier wird die Speicheradresse jedes 'read'-Befehls der Transaktion gespeichert.
- Write-Set: Hier wird die Speicheradresse, und der Wert jedes 'write'-Befehls der Transaktion gespeichert.

Es giebt zwei grundsätzliche Arten der Konfliktüberprüfung.

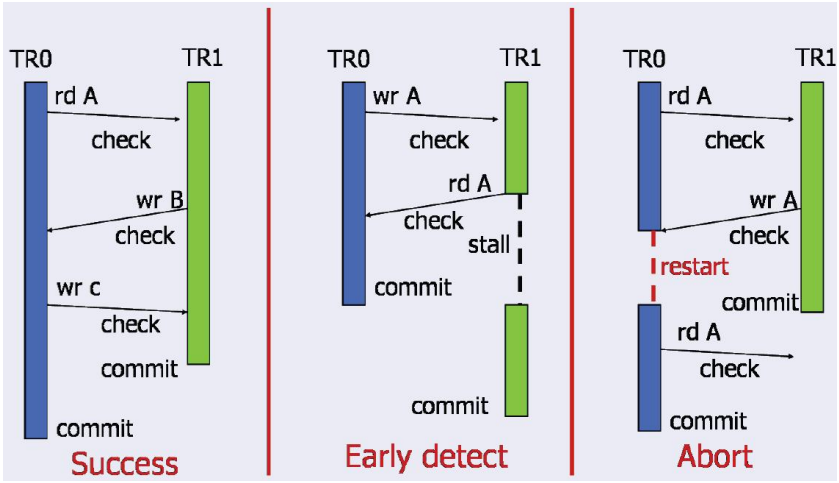
- Eager Checking:
  - Überprüfung während Lade- und Speicheroperationen
  - Pessimistisches Modell
  - Verwendung des Contention Managers zur Entscheidung
  - Read und Write Sets müssen auf dem ganzen System sichtbar sein
- Lazy Checking:
  - Überprüfung auf Konflikt erst mit Commit
  - Optimistisches Modell
  - Mischen unterschiedlicher Mechanismen für Lade- und Speicheroperationen möglich

Vor und Nachteile der beiden Varianten:

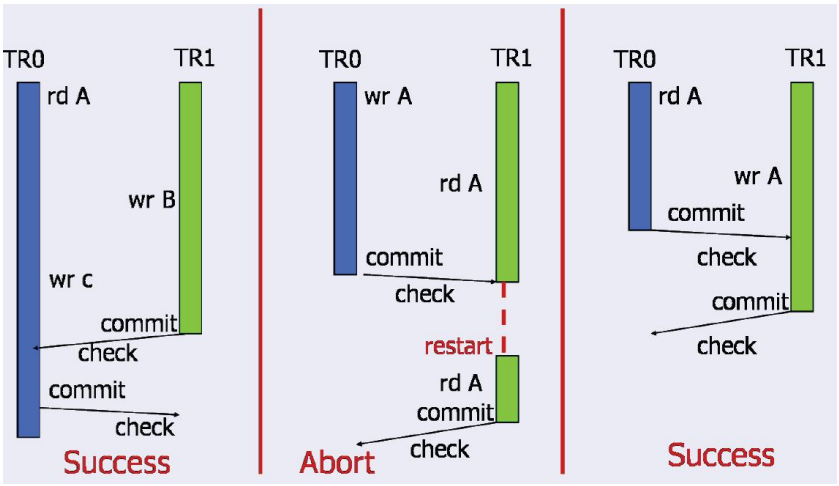
- Frühzeitige Besitznahme (**eager detection**):
  + Kann Konflikte frühzeitig erkennen.
  - Kann Transaktionen unnötiger Weise abbrechen.
- Späte Besitznahme(**lazy detection**):
  -Lässt zum Scheitern verurteilte Transaktionen lange laufen.
  +Überieht Konflikte, die sich später gar nicht auswirken.

Beispiel für die Unterschiede der beiden Varianten:

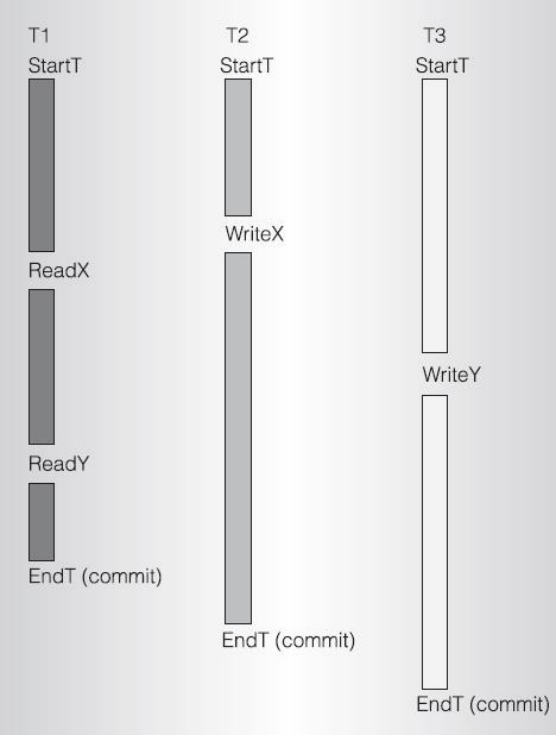
1. Eager Checking(pesimistisch)



2. Lazy Checking(optimistisch) Auflösen des Konfliktes, sobald er erkannt wurde:



Normalerweise würde eine der beiden Transaktionen abgebrochen werden müssen. Die Entscheidung welche der beiden Transaktionen abgebrochen wird, ist aber komplex.



Zu sehen ist die Ausführungs-Zeilinie dreier Transaktionen. Transaktion 1 hat einen Konflikt mit Transaktion 2 (ReadX und WriteX), und dann später auch noch mit Transaktion 3 (ReadY und WriteY). Sollte nun bei dem ersten Konflikt die zweite Transaktion abgebrochen werden, so muss bei dem zweiten Konflikt eine weitere Transaktion abgebrochen werden. Wird hingegen Transaktion 1 beim ersten Konflikt abgebrochen so können die Transaktionen 2 und 3 beide commiten.

3.2.5 Granularität

Bei der Implementierung eines Transactional Memory Systems muss festgelegt werden, mit welcher Granularität gearbeitet wird. Bei früheren Vorschlägen für hard- und softwarebasierte Systemen wurde mit Worten gearbeitet, d.h. jede Transaktion bestand aus dem Zugriff auf einem oder mehreren Worten. Probleme dabei waren der hohe Verwaltungsaufwand und der hohe Speicherbedarf. In modernen hardwarebasierten Systeme arbeitet man mit Blöcken fester Größe, insbesondere Seiten oder Cache-Zeilen. (Andere Blockgrößen können unterstützt werden, je nach Fähigkeit der Hardware, die alle Schreib-und Lesezugriffe überwachen muss) Moderne STM-Systeme arbeiten dagegen überwiegend objektorientiert, um den Aufwand für die Zugriffskontrolle zu minimieren.

Arten der Granularität:

- Objektgranularität
  - Geringer Aufwand

  - Entspricht Programmierer-Denkweise
  - Große Objekte bedingen False-Sharing und damit unnötige Aborts
  - Nur SW-TM bzw. Hybridmodell
- Wortgranularität
  - Minimierung von False-Sharing
  - Steigender Aufwand
- Cachezeilen-Granularität
  - Kompromiß zwischen Objekt- und Wortgranularität
  - Tauglich für HTM und STM
- Mix and Match
  - Ausrichtung an TM-Modell und Gegebenheiten

3.2.6 Vor- und Nachteile von Transactional Memory

Vorteile

- Kohärenz-Kontrolle einfacher
- Kommunikation zwischen Prozessen seltener
- Weniger Synchronisationspunkte nötig
- Illusion von Einprozessorsystemen

Nachteile

- Interprozessor-Bus muss hohe Bandbreite haben
- Falls Commit zu groß für den Cache, muss der Bus gesperrt werden
- Algorithmus in Hardware naher Sprache kann schneller sein

3.2.7 Optimierungsmöglichkeiten für TM

- Double Buffering
- Hardware-Controlled Transactions
- Localisation of Memory References

Double Buffering

- Zusätzliche Write-Buffers und read- bzw. write-Bits in der Cacheline
- CPU kann die nächste Transaktion starten bevor die aktuelle committet

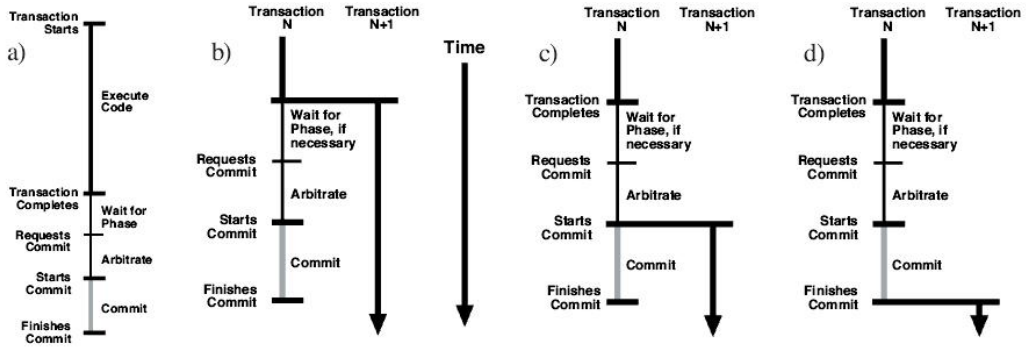


Abbildung 1: Double Buffering

Hardware-Controlled Transactions

- Teilt das Programm selbstständig in Transactions
- Fasst kleinere Transactions zu einer großen zusammen
- Gelegentlich Barrier setzen, um tote Threads zum Leben erwecken

Folge: Höherer Speedup

Localisation of Memory References

- Load & Store, die nur lokal benötigt werden, nicht broadcasten(z.B. Stack Referenzen)
- Realisierbar, indem man entweder Seiten als local-only markiert oder mit Befehlen wie local-load und local-store arbeitet

Folge: Bandbreite wird gespart

3.3 Software Transactional Memory(STM)

3.3.1 Funktionsweise STM

Innerhalb einer Transaktion muss Buch geführt werden über alle Lese- und Schreibzugriffe auf gemeinsam genutzte Speicherbereiche, um gegebenenfalls bei einem Abbruch wieder auf einen konsistenten Zustand zu gelangen. In typischen STM-Systemen werden gemeinsam genutzte Speicherbereiche deklariert (durch **open()**), wobei der Programmierer dem STM mitteilt, ob er nur lesend oder auch schreibend auf diese Speicherbereiche zugreifen will. Von Speicherbereichen, auf die geschrieben werden soll, wird eine Kopie angelegt und nur die Kopie modifiziert, und falls die Transaktion erfolgreich abgeschlossen wird, wird mit einer atomaren Operation der alte Speicherbereich durch den neuen ersetzt.

In einem typischen STM werden Schreibzugriffe publiziert (d.h. für alle anderen Prozesse sichtbar). Man sagt, dass der jeweilige Prozess versucht, den entsprechenden Block in Besitz zu nehmen (**acquire()**).

An dieser Stelle werden Konflikte sichtbar:

- Wenn eine andere Transaktion den fraglichen Bereich in Besitz hält, kann eine der beiden Transaktionen abgebrochen werden.
- Andere Transaktionen können abgebrochen werden, wenn sie die Daten gelesen haben, für die hier grade der Besitz angefordert wird.
- Die Transaktion kann abgebrochen werden, wenn Daten ihrer vorhergehenden Lesezugriffe inzwischen überschrieben wurden.

Die Bekanntgabe der Schreiber ist grundsätzlich unerlässlich für die Konflikterkennung und findet immer vor dem abschließenden, tatsächlichen Festschreiben der Änderungen (commit) statt. acquire()ist normalerweise keine eigenständige Operation, sondern Bestandteil von open() oder commit()(je nach STM-Implementierung).

3.3.2 STM-Beispiel-Implementierung

Im folgenden wird beispielhaft eine STM Implementierung mit frühzeitiger Konflikterkennung gezeigt. Die Transaktion ist durch die TMThread-Klasse realisiert, die von Thread erbt und zusätzliche Methoden für Transaktion bereitstellt: beginTransaction, open, commit, abort, checkStatus, etc. Das Transaktions-Objekt TMObject ist als Hüll-Klasse realisiert, die Java 'Object' enthält und implementiert die clone-Operation. **open** legt hierbei

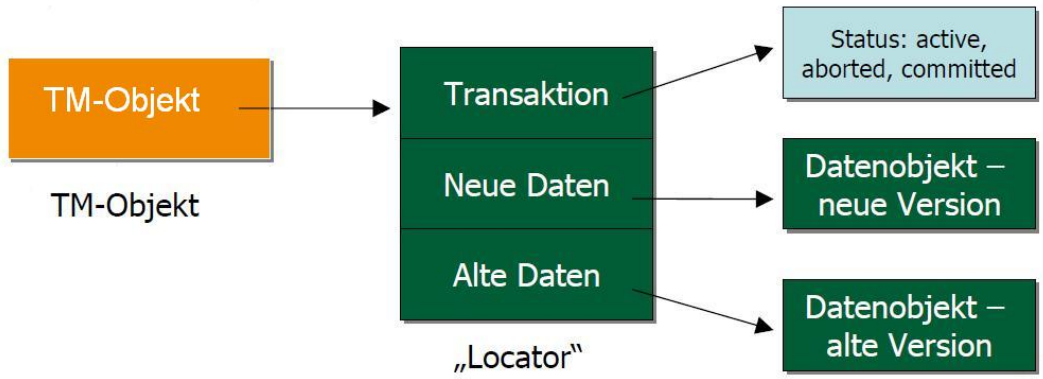
Beispiel atomarer Zähler:

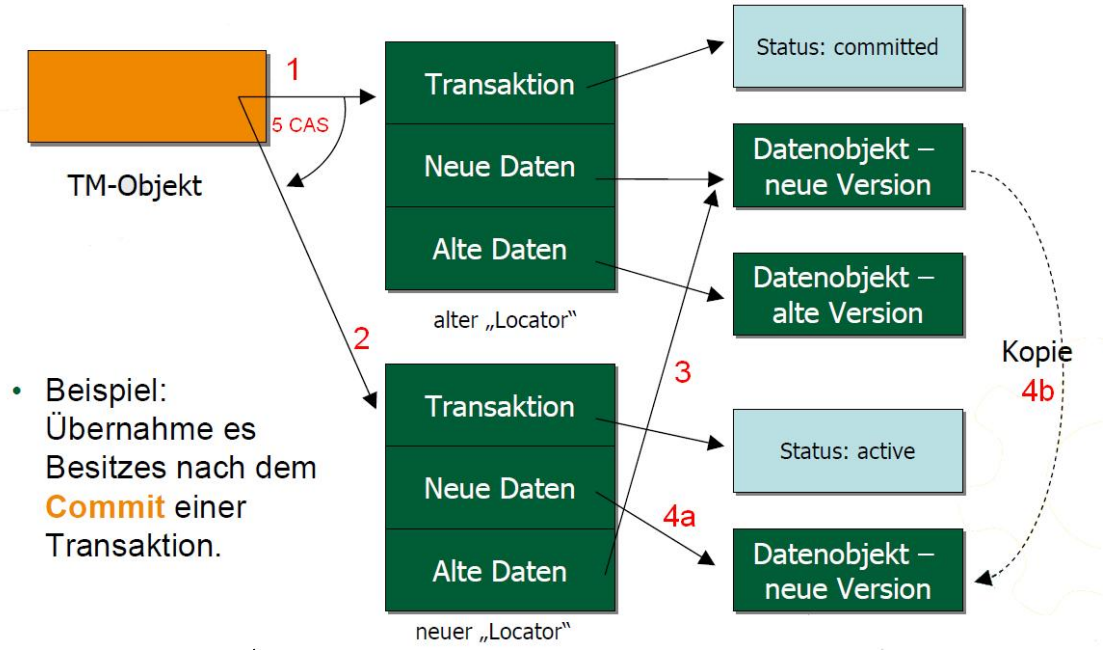
```
//Initialisierung
Counter   counter = new Counter(0);
TMObject tmObject = new TMObject(counter);

//Verwendung innerhalb einer Transaktion
//Faden ruft beginTransaction auf
Counter counter=(Counter)tmObject.open(WRITE);
counter.inc();
...
```

die Datenstrukturen für Transaktion einschließlich einer Kopie an(vgl. unten). Es wird von der Transaktion wie gewöhnlich erst nur die Kopie des Objekts verändert, wobei in der Implementierung gesichert sein muss, dass keine andere Transaktion zugriff auf die lokale Kopie des Objekts hat. Die Transaktion kann von sich aus abbrechen(**abort**), ist dies nicht der Fall, wird mit CAS eine konsitente neue Version erzeugt(oder auch nicht) (**commit**). Gemeinsam genutzte Objekte werden über eine zusätzliche Indirektion referenziert. Der so genannte 'Locator' enthält

- eine Referenz auf die Transaktion, die das Objekt 'besitzt' bzw. zuletzt besaß,
- einen Zeiger ('alt') auf eine Kopie der Daten vor Beginn der Transaktion,

- einen Zeiger ('neu') auf die aktuellen Daten. Diese werden von der Transaktion genutzt und ggf. verändert.

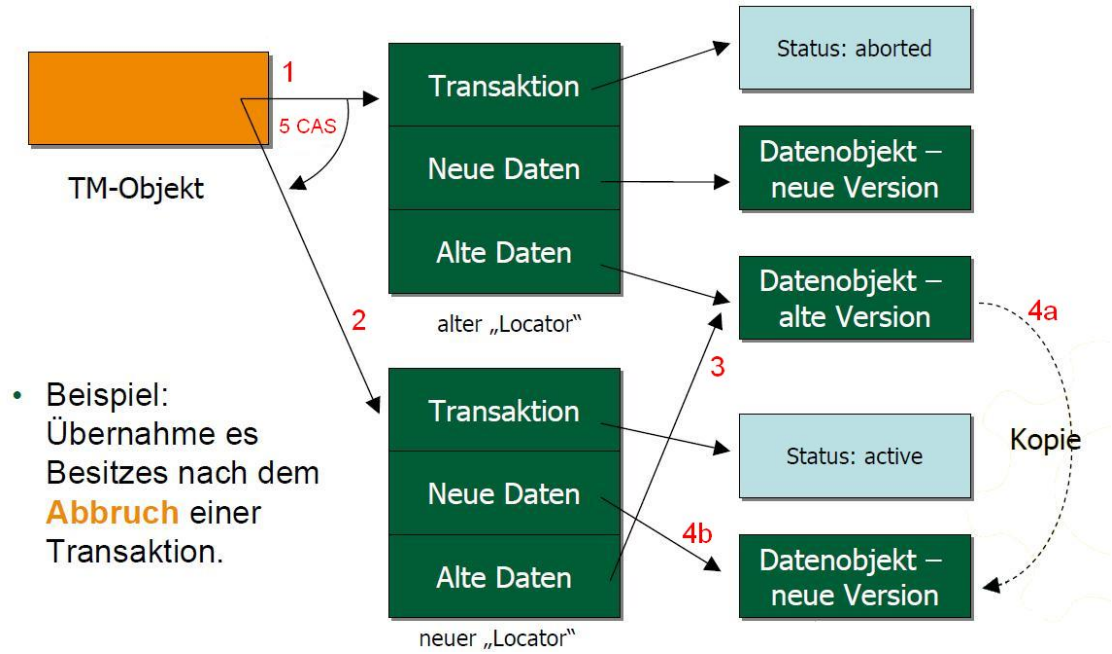




- Beispiel: Übernahme es Besitzes nach dem **Commit** einer Transaktion.

Besitznahme an einem Objekt durch eine neue Transaktion nach einem Commit:

1. Ein Zeiger auf den alten Locator wird gespeichert.
2. Es wird ein neuer Locator angelegt, der auf die neue Transaktion verweist. Deren Zustand wurde mit 'active' initialisiert.
3. Der Zeiger 'alt' des neuen Locators zeigt auf die aktuellen ('neu' nach commit) Daten der abgeschlossenen Transaktion.
4. Der Zeiger 'neu' im neuen Lokator zeigt auf eine frisch angelegte Kopie dieser Daten.
5. Mittels einer CAS-Operation wird die global sichtbare Referenz für das gemeinsam genutzte Objekt auf den neuen Locator umgesetzt, wenn die Referenz sich seit (1.) nicht geändert hat (durch eine andere Transaktion).



Besitznahme an einem Objekt durch eine neue Transaktion nach einem Abbruch:

1. Ein Zeiger auf den alten Locator wird gespeichert.
2. Es wird ein neuer Locator angelegt, der auf die neue Transaktion verweist. Deren Zustand wurde mit 'active' initialisiert.
3. Der Zeiger 'alt' des Locators zeigt auf die 'alten', ungeänderten Daten der abgebrochenen Transaktion.
4. Der Zeiger 'neu' zeigt auf eine frisch angelegte Kopie dieser alten Daten.

5. Mittels einer CAS-Operation wird die global sichtbare Referenz für das gemeinsam genutzte Objekt auf den neuen Locator umgesetzt, wenn die Referenz sich seit (1.) nicht geändert hat (durch eine andere Transaktion).

Bei nur lesenden Transaktionen kann am Ende einer solchen Transaktion wieder mit **compare-and-swap (tmObjectRef, alter Verweis, alter Verweis)** validiert werden, ob die gelesenen Daten noch gültig sind (und nicht von einer anderen Transaktion verändert wurden). 'Nur Lesen' wird durch **tmObject.open(READ)** signalisiert.

Leser können durch Inspektion des Locatorsfeststellen, welche Transaktion das Objekt in Besitz (genommen) hat, und welchen Zustand diese Transaktion hat. Leser können auf Objekte zugreifen, die zu einer abgeschlossenen oder abgebrochenen Transaktion gehören. Leser müssen frühzeitig abbrechen, wenn das Objekt, auf das sie zugreifen wollen, zu einer aktiven Transaktion gehört. Leser müssen abbrechen, sobald sie detektieren, dass das Objekt, auf das sie zugreifen, von einer Transaktion (zum Schreiben) in Besitz genommen wurde.

3.3.3 Probleme der STM

Auch wenn eine lesende Transaktion schließlich abgebrochen wird, wenn sich die Daten, die sie gelesen hat, zwischendurch geändert haben, können Probleme entstehen, wenn sie auf diesen Daten weiterarbeitet.

- Deadlocks durch Endlosschleifen (s. Beispiel unten)
- Null-Referenzen
- Diese Probleme müssen in Abhängigkeit von OS, Sprache und STM gelöst werden.
- Das Beispiel unten stellt kein Problem dar, wenn die Implementierung mit Kopien der zu ändernden Speicherbereiche arbeitet.

```
atomic {
  if (x != y)
    while (true) {}
}
```

Transaktion A

```
atomic {
  x++;
  y++
}
```

Transaktion B

3.3.4 Vor- bzw Nachteile des STM

Vorteile der STM

- Bereits auf bestehenden Systemen einsetzbar

Nachteile der STM

- Schlechte Skalierbarkeit

3.4 Hardware Transactional Memory (HTM)

3.4.1 Funktionsweise HTM

Die ersten HTM Designs waren minimalistische Versuche, die auf Veränderungen im Cache Consistency Protocol und einer Komplettierung der Instrucktion Set Architecture(ISA) mit einer kleinen Menge von neuen Befehlen basierte. Ausserdem werden die neuen Daten werden in einem erweiteren, oder abgegrenzten Cache (oder Buffer) zwischengespeichert, bis die Transactions entweder commited oder abbricht. Diese zwei Modifikationen sind ausreichend für einfache HTMs.
Um Transaktionen auf dem ISA Level zu unstützen, werden Befehle für das Starten (STR) und das Beenden (ETR) von Transaktionen benötigt. Ausserdem sind spezielle Versionen von load(TLD) und write(TST) für read bzw. write Aktionen auf gemeinsamen Speicher notwendig. Das Hinzufügen von Befehlen für den Abbruch (ABR) bzw. die Überprüfung(VLD) von Transaktionen lässt einige Optimierungen zu. So kann eine Überprüfung einer langen Transaktion die am Anfang einen Konflikt ergibt zum sparen von Energie beitragen.
Da HTMs die neuen Daten der Transaktionen im Harware Buffer oder im Data Cache speichern, werden Konflikte nicht wie bei STM, der Konflikte anhand der Objekte der Programmierprache erkannt, sondern erst in den Cache Lines erkannt. Eine Cache-Line ist die kleinste Verwaltungseinheit innerhalb des Caches von Prozessoren. Es handelt sich dabei um eine Kopie eines Speicherbereichs mit mehreren aufeinander folgenden Adressen. Die Zugriffe vom Cache-Speicher zur CPU oder zum Hauptspeicher erfolgen somit in einem einzigen, blockweisen Transfer. Die Cache-Line ist in der Regel 8 bis 128 Byte groß.

Cachelines: Jede Zeile muss zusätzlich folgende Informationen enthalten:

- Read Bit: Cacheline wurde während einer Transaction gelesen
- Modified Bit: Falls Cachline (teilweise) geändert wurde

Zusätzlich möglich:

- Renamed Bit: Zeigt an, welcher Teil einer Cacheline geändert wurde

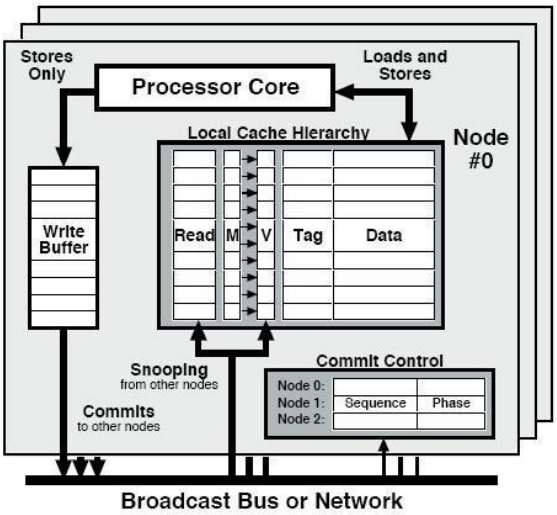
3.4.2 HTM-Beispiel TCC

- Transaktion auf Cachegröße beschränkt
- Transaktionsstart -> Lesen von Befehlen und Daten in Caches
- Cache Overflow -> Anforderung einer Commit-Erlaubnis

- Weiterbearbeitung nach Commiterlaubnis
- Lazy Versioning
- Festhalten von Cachezeilenzugriffe mittels Read/Modified-Bits
- Optimistische Konflikterkennung
- Je Transaktion Erstellung einer Liste betroffener Adressen (Write State)
- Kommunikation dieser Adressen bei Commit (Invalidierung und Aktualisierung)

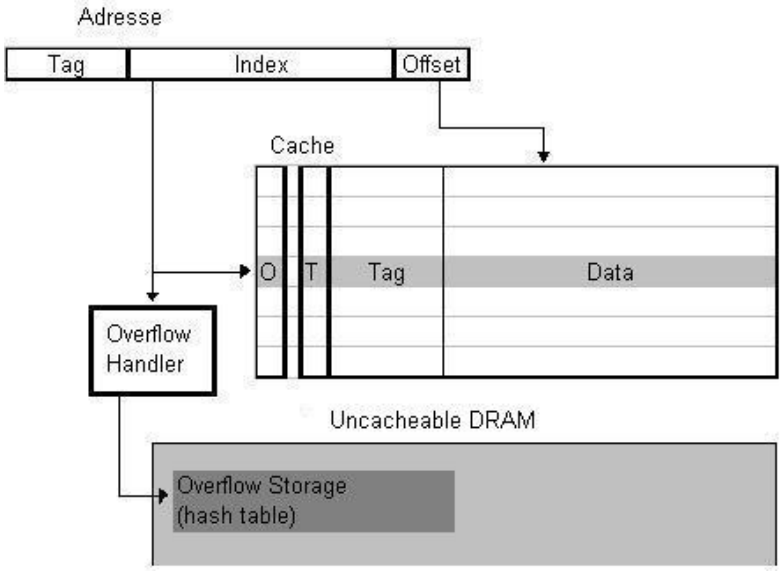
Hardware Eigenschaften:

- Register-Checkpointing (Umbenennung)
- Schreibpuffer für Write-Set (4-8kB)
- Read-/Modified-Bits
- Commit-Kontrolle



3.4.3 HTM-Beispiel LTM

- Transaktionen von der Größe des verfügbaren physikalischen Speichers
- Verwendung einer zusätzlichen Hash-Tabelle (overflow storage)
- Lazy Versioning
- pessimistische Konflikterkennung
- Konsistenzwahrungaufwendiger als bei TCC



3.4.4 HTM-Beispiel ATLAS

Der Hardware TM funktioniert vollständig ohne Locks und Semaphoren, und es wird kein MESI-Protokoll verwendet.
Transaktionen sind für alle Anwendungen transparent.

- Erstes CMP-System mit Shared Memory und TM-Support
- 8 PowerPC 405 + 1 für OS
- Auf BEE2-Multi-FPGA Board gemappt
- Läuft mit 100MHz
- Verwendet kein MESI-Protokoll

Transaktionsbeginn

- R/M-Bits rücksetzen
- Register sichern
- Commit-Token-Arbiter benachrichtigen

Transaktionsende

- Anfordern des Commit-Tokens
- Rückschreiben des Write-States in den HS
- Konflikt: Ausnahmebehandlung

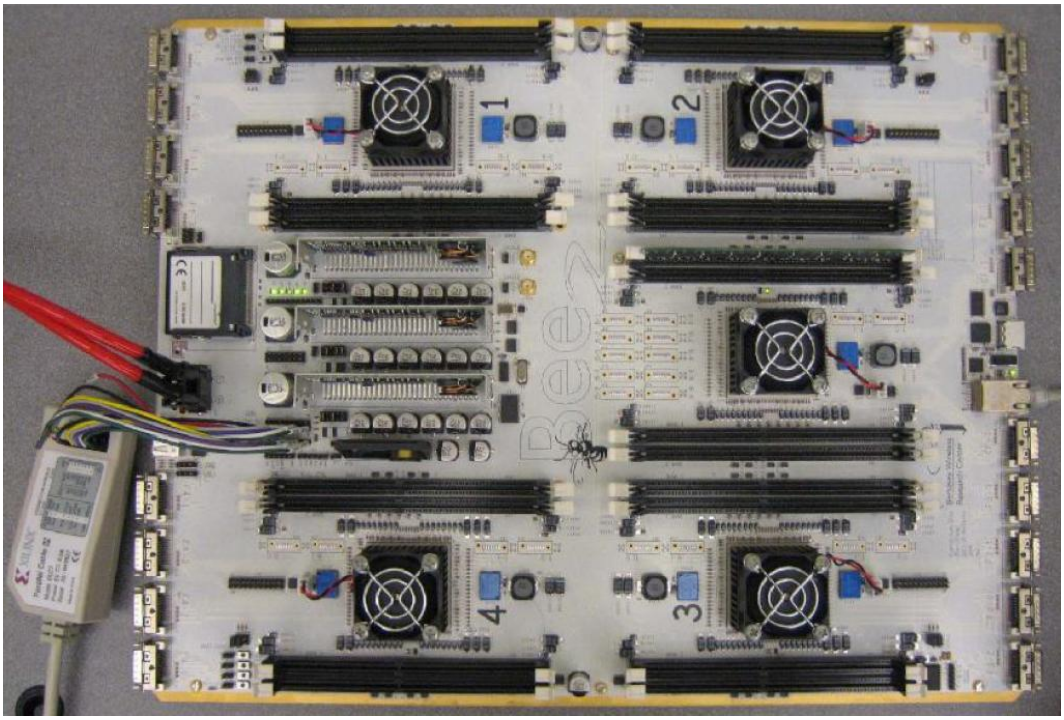


Abbildung 2: Aufbau des ATLAS

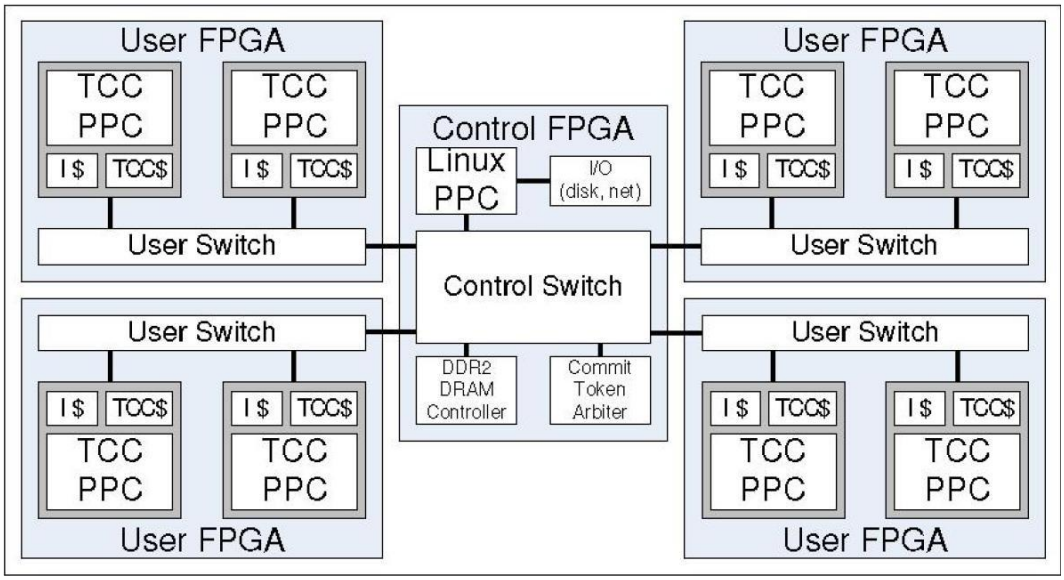


Abbildung 3: Schaltbild des ATLAS

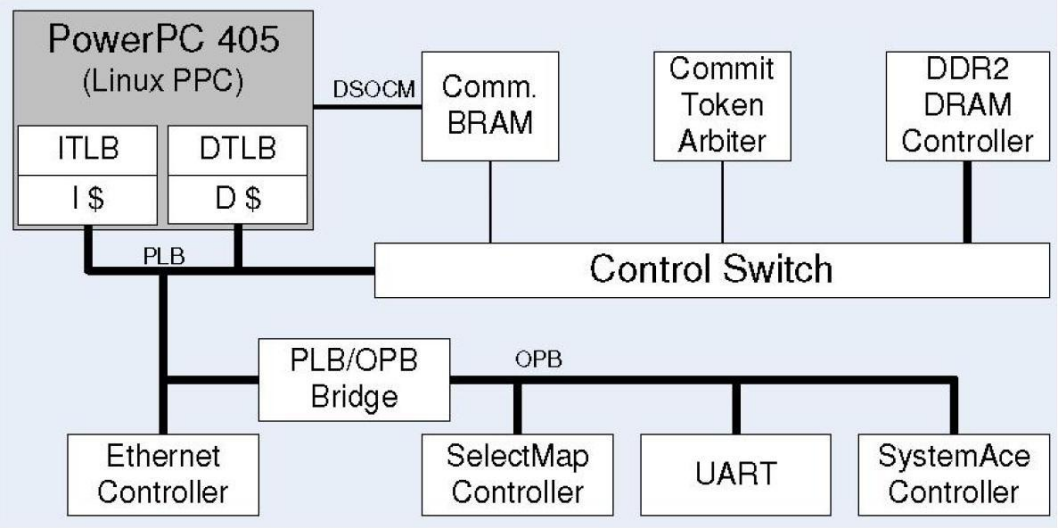


Abbildung 4: Control FPGA

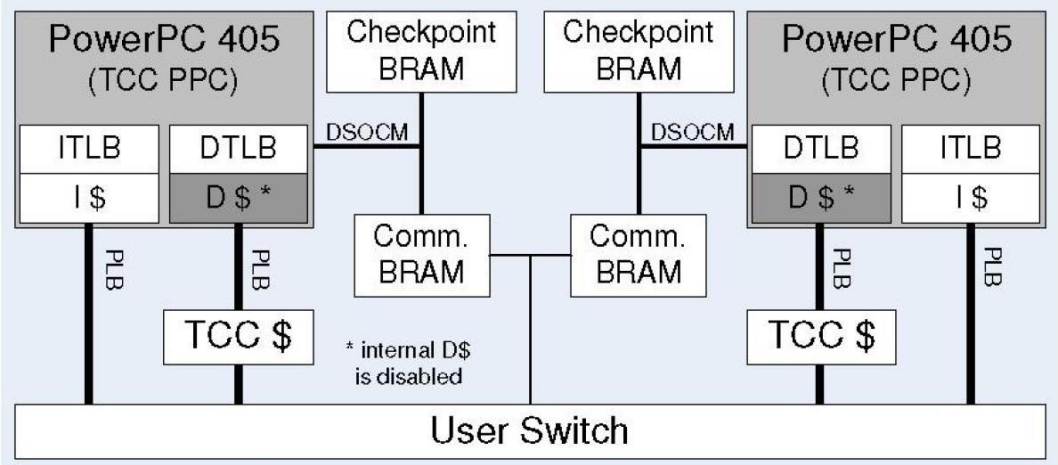


Abbildung 5: User FPGA

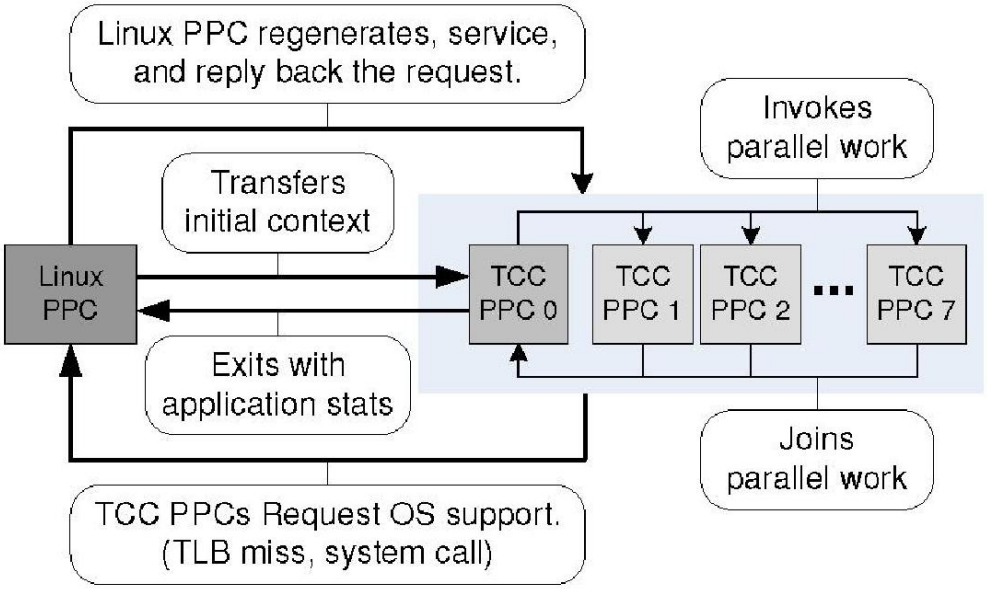


Abbildung 6: System Software

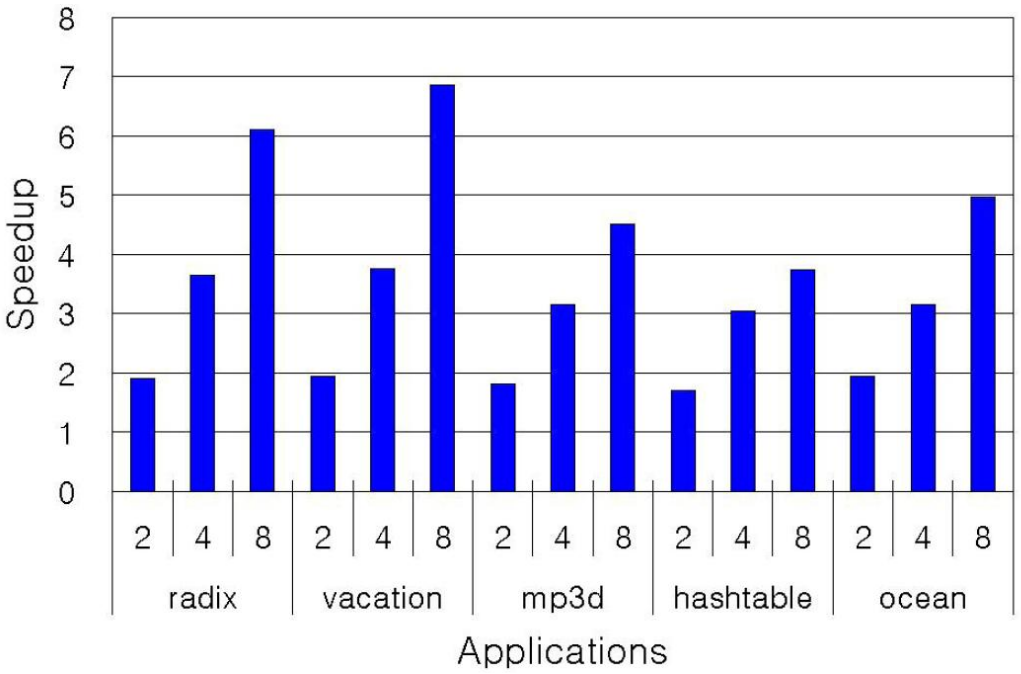


Abbildung 7: Benchmark ATLAS

3.4.5 Vor- bzw. Nachteile des HTM

Vorteile HTM

- Skaliert sehr gut
- Atomizität transparent für alle Anwendungen

Nachteile HTM

- Erfordert neue Hardware
- Nicht einfach zu Programmieren
- Nicht so flexibel wie STM

3.5 Hybrid-Transactional Memory

3.5.1 Funktionsweise

Die Transaktion wird zunächst via HTM getätigt. Erfolg ein Abbruch, dann wird die Transaktion per STM abgehandelt(**fall-back**). Ein Abbild der atomaren Blöcke wird sowohl für HTM- als auch STM-Ausführung erstellt (multi-versioning). Hybrid-TM hat nahezu HTM Performance für kleine Transaktionen, aber sobald ein fall-back eintritt einen großen Performance Verlust.

3.5.2 Konflikt Erkennung

- STM/STM-Konflikte durch STM
- HTM/HTM-Konflikte durch HTM
- HTM/STM-Konflikte durch zusätzlichen Code in HTM-Codepfad

3.5.3 Interoperabilität HTM/STM

- Zusammenspiel zwischen HTM und STM
- Überprüfung des Transaction Records
- Sicherstellung, daß Zugriffsadresse nicht durch STM-Transaktion blockiert ist
- Zwei Basisfunktionen
  - HTMReadBarrier(*addr*)
    - Lesen, falls *addr* nicht durch STM-Transaktion blockiert
    - Abort bei Konflikt
  - HTMWriteBarrier(*addr*)
    - Schreiben, falls addr nicht durch STM-Transaktion blockiert
    - Abort bei Konflikt

3.5.4 Hybrid-TM-Beispiel HyTM

- Wird zur Zeit von Sun erforscht

Kombiniert Vorteile von HTM mit STM:

- Falls HTM verfügbar wird HTM benutzt, falls nicht (oder für HTM zu groß), benutze STM (Fallback)
- Wortbasiertes STM
- in kommendes Sun-CPU (Codename: ROCK) integriert

Wie funktionert HyTM? Lösung: Orec-Tabelle und Transaction Discriptor

Orec-Tabelle enthält folgende Informationen

- tdid: Transaktion-ID
- ver: Versionsnummer der Transaktion
- mode: Zeigt aktuellen Zustand des Orecs an
- rdcnt: Anzahl der lesenden Transactions

	tdid	ver	mode	rdcnt
0:	0	27	W	-
1:	0	27	W	-
2:				
3:	7	53	R	2
4:	1	35	W	-
5:				
6:	5	27	R	1
7:				

Abbildung 8: Orec-Tabelle

Transaction Discriptor enthält folgende Informationen

- Transaction Header
- Read Set
- Write Set

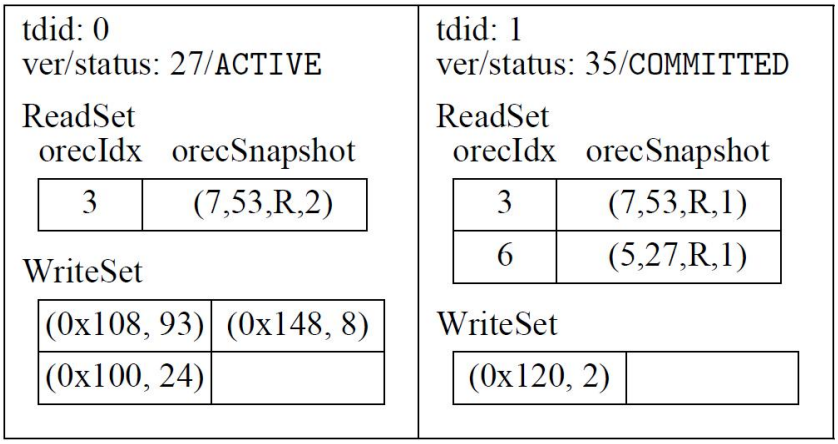


Abbildung 9: Transaction Discriptor