Christian-Albrechts-Universität zu Kiel

Masterarbeit

Transaktionaler Speicher

Karl Balzer

31. August 2016

Institut für Informatik
Arbeitsgruppe Programmiersprachen und
Übersetzerkonstruktion

Prof. Dr. Frank Huch

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Zusammenfassung

Transactional Memory (TM) stellt eine Alternative zu herkömmlichen Modellen beim Zugriff auf geteilten Speicher da. Im wissenschaftlichen Umfeld gibt es viele Vorschläge zur Umsetzung von TM und seit einigen Jahren auch eine begrenzte Unterstützung durch Prozessoren von Intel und IBM. Der GCC bringt eine Compilerunterstützung und ein hybrides TM-System mit. Da dieses System aber exklusiv als Hardware- oder SoftwareTM arbeitet und so in seiner Parallelität begrenzt ist, wurde in dieser Arbeit Parallbrid entwickelt. Die Hoffnung bei einem TM-System, das mehr Parallelität ermöglicht, war, dass es transaktionale Aufgabenszenarien schneller bewältigt. Als Vorbild für Parallbrid wurde Invyswell genommen, da es als schnelles, gut skalierendes System gilt, und um einen Fehler korrigiert.
Parallbrid erfüllt den Wunsch nach mehr Parallelität bei Transaktionen und somit weniger brachliegenden Prozessorkernen. Wie die mit der STAMP-Benchmarksuite ausgeführten Tests aber zeigen, führt dieser Mehrgewinn an Parallelität nicht zu einer schnelleren Ausführung der Benchmarks. Dies ist zum Einen dem Overhead geschuldet, der notwendig ist, um die unterschiedlichen Transaktionstypen korrekt parallel auszuführen. Zum Anderen bedeutet mehr Parallelität auch mehr Konflikte, wodurch mehr Transaktionen abbrechen und Rechenzeit vergeuden.

Inhaltsverzeichnis

1 Einleitung

Transactional Memory ist ein Programmiermodell für die Verwendung von einem geteilten Speicher. Es hat sich in den letzten drei Jahrzenten von einem theoretischen zu einem, in der Praxis einsetzbaren, Modell entwickelt. Für diesen Einsatz in der Praxis wurden viele unterschiedliche Ansätze für Software, Hardware und Hybrid Transactional Memory vorgeschlagen. Seit ein paar Jahren ist auch in kommerziellen Prozessoren eine Unterstützung für Hardware Transactional Memory verfügbar.
Der Vorteil von Transactional Memory ist, dass ein Programmierer sich bei der Verwendung von geteiltem Speicher keine oder nur wenige Gedanken über die Zugriffskontrollen machen muss. Das TM-System nimmt ihm diese Aufgabe ab. Damit die Benutzung von Transactional Memory aber nicht genau so komplex ist, wie das Zugriffsproblem, das es löst, muss eine Compilerintegration für TM vorhanden sein. Der GCC, wie viele Compiler, stellen so eine Compilerintegration und ein TM-System bereit.
Das im Rahmen dieser Arbeit entwickelte Parallbrid ist ein System, das Hybrid Transactional Memory anbietet. Es ist aus der Frage entstanden, ob es möglich ist, die Compilerintegration des GCCs zu nutzen und ein System zu erschaffen, das durch Ausnutzung von mehr Parallelität schneller läuft als GCC-TM. Als Vorbild für Parallbrid wurde Invyswell [CGS+14] genommen, da es einen hohen Grad an Parallelität bietet. Invyswell hat in seiner vorgeschlagenen Form einen Fehler, der im Rahmen dieser Arbeit korrigiert wurde, aber leider die Parallelität wieder etwas einschränkt.
Diese Arbeit beschäftigt sich als Erstes mit der Erklärung, was Transactional Memory (Kapitel 2) ist, wie es funktioniert und was die drei Varianten Software, Hardware und Hybrid Transactional Memory sind. Bei den drei Varianten liegt der Schwerpunkt auf den hybriden Systemen, in deren Kontext auch Invyswell und GCC-TM vorgestellt werden. Die Anforderungen, die Parallbrid erfüllen muss, die Funktionsweise des Systems und eine Erklärung, wo und wie Parallelität gewonnen wird, sind im Kapitel 3 beschrieben. Anschließend an diese Darstellung erfolgt eine Beschreibung der Implementierung (Kapitel 4) als geteilte Bibliothek, die von Programmen genutzt werden kann, die mit dem GCC compeliert wurden. Die Verwendung von Parallbrid als TM-System ist in Kapitel 5 erklärt. Da die STAMP-Benchmarksuite typischerweise zur Berwertung von TM-Systemen benutzt wird, beschäftigt sich Kapitel 6 mit dem Vergleich von Parallbrid und GCC-TM unter Benutzung von STAMP. Abschließend wird ein Fazit (Kapitel 7) gezogen und ein Ausblick (Kapitel 8) gegeben.

2 Transactional Memory

Transactional Memory ist ein Programmiermodell für die Verwendung von geteiltem Speicher. Es stellt eine Alternative zu herkömmlichen Ansätzen für gegenseitigen Ausschluss, wie Locks, da.
Wird in einer Software mit mehreren Threads ein geteilter Speicher benutzt, so müssen Zugriffskontrollen für diese Speicherbereiche benutzt werden. Ohne Zugriffskontrollen droht die Gefahr, dass ein inkonsistenter Datenzustand gelesen wird oder Änderungen am Speicher verloren gehen. In Abbildung 2.1 sind zwei Threads dargestellt, die beide eine globale Variable i inkrementieren. Das Ergebnis von i ist ohne eine Zugriffskon-

```
int i = 0 // global variable
```

Thread1:	**Thread2:**
`i++;`	`i++;`

Abbildung 2.1: Zwei Threads, die bei paralleler Ausführung zu einem undefinierten Zustand von i führen.

trolle nicht eindeutig definiert, da es je nach Schedulerausführung 1 oder 2 sein kann. Der Grund hierfür ist, dass das Inkrementieren nicht atomar stattfindet, sondern aus einer Lese- und einer Schreibeoperation besteht. Für einfache Datentypen bieten viele Programmiersprachen zwar atomare Operationen an, für komplexe Datentypen oder mehrere Anweisungen, die atomar ausgeführt werden sollen, gibt es diese aber nicht. Die Lösung für dieses Problem ist der gegenseitige Ausschluss beim Zugriff auf die geteilten Variablen. Klassischerweise kommen hierbei Locks, Mutexes oder Semaphoren zum Einsatz. Die Programmierung von komplexen Softwaresystemen, die viele geteilte Speicherbereiche benutzen, kann mit diesen Zugriffskontrollen aber sehr anspruchsvoll und fehleranfällig sein. Bei der Verwendung von nur einem Lock für alle geteilten Speicherbereiche ist die Korrektheit des Programms zwar leicht zu zeigen, es ist aber kein paralleler Zugriff auf unterschiedliche, geteilte Speicherbereiche möglich. Wählt man stattdessen eine feine Granularität bei der Zugriffskontrolle, dann wird die Korrektheit schwer zu gewährleisten und es droht die Gefahr von Dead- und Livelocks. Transactional Memory bietet hier eine Alternative, da sich das Transaktionssystem um die korrekte Ausführung kümmert.

Bei der Verwendung von Transactional Memory kommen Transaktionen zum Einsatz. Jeder Programmabschnitt, der als atomare Einheit ausgeführt werden soll, wird vom Programmierer als Transaktion definiert. Erreicht die Ausführung des Programmablauf eine

Transaktion, so wird diese mit Unterstützung des Transaktionssystems ausgeführt. Das Transaktionssystem versucht, die Transaktion erfolgreich auszuführen, und wiederholt den Vorgang, falls dies nicht möglich war. Eine erfolgreiche Ausführung ist dann gegeben, wenn eine Programmausführung eine Transaktion abschließt und die für Transaktionen definierten ACI-Eigenschaften (Atomicity, Consistency und Isolation)[1] eingehalten wurden (Kapitel 2.1). Daneben werden im Folgenden die Funktionsweise von Transactional Memory (Kapitel 2.2) sowie insbesondere das Nesting (Kapitel 2.2.1) und sogenannte unwiderrufliche Aktionen (Kapitel 2.2.2) thematisiert. Abschließend werden Software (Kapitel 2.3.1), Hardware (Kapitel 2.3.2) und Hybrid Transactional Memeroy (Kapitel 2.3.3) vorgestellt.

2.1 Transaktions Eigenschaften und Garantien

Die Atomicity-Eigenschaft gewährleistet, dass eine Transaktion wie eine atomare Operation ausgeführt wird. Andere Transaktionen, die den Speicher beoachten, nehmen Veränderungen einer Transaktion T am Speicher nur als Ganzes war. Sie sehen entweder keine oder alle Veränderungen, die T gemacht hat.
Die Consistency-Eigenschaft stellt sicher, dass Transaktionen den Speicher nicht in einen inkonsistenten Zustand überführen. Wenn der Speicher vor der Ausführung einer Transaktion in einem konsistent Zustand war, so muss er das auch nach der Ausführung sein. Transaktionen, die abgebrochen und wiederholt werden müssen, dürfen keinen inkonsistenten Zustand hinterlassen.
Mit der Isolation-Eigenschaft ist gemeint, dass eine Transaktion T das System wahrnimmt, als wäre sie die Einzige im System. T wird somit nicht von parallel laufenden Transaktionen beeinflusst und hat auch keinen Einfluss auf diese. Der Zustand des Speichers nach einer parallelen Ausführung von mehreren Transaktionen muss genauso sein, als wären die Transaktionen serialisiert ausgeführt worden.
Neben den ACI-Eigenschaften sollte ein Transaktionssystem auch Fortschritt und *opacity* garantieren.
Bei der Fortschrittsgarantie unterscheidet man zwischen einfachem Fortschritt (*progress*) und fortschreitendem Fortschritt (*forward progress*). Ersterer garantiert nur, dass es Transaktionen gibt, die erfolgreich abgeschlossen werden, aber nicht, dass alle Transaktionen erfolgreich abgeschlossen werden. Es kann also Transaktionen geben, die nie beendet werden, zum Beispiel aufgrund von Starvation. *Forward progress* ist eine strengere Fortschrittsgarantie. Sie sichert zu, dass alle Transaktionen irgendwann erfolgreich ausgeführt werden. Die zweite Garantie ist bei Transaktionssystemen wünschenswert, da ohne sie nicht sicher gestellt werden kann, dass sich ein Programm unter Benutzung von Transactional Memory erwartungskonform verhält.
Opacity ist die Eigenschaft, dass zum Scheitern verurteilte Transaktionen abgebrochen werden und nicht als Zombie-Transaktionen weiter im System laufen. Transaktionen, die einen inkonsistenten Datenbestand gelesen haben, können nicht mehr erfolgreich

[1] Die bei Datenbanktransaktionen erforderliche Eigenschaft *Durability* spielt bei Transactional Memory keine Rolle.

abgeschlossen werden. Laufen diese Transaktionen trotzdem noch weiter, ohne sofort abgebrochen zu werden, spricht man von Zombie-Transaktionen. Diese Transaktionen haben zwar keinen Einfluss auf das System und andere Transaktionen, sie können aber zu einem Problem für sich selbst werden. Der von ihnen gelesene inkonsistente Datenbestand könnte nämlich zu Endlosschleifen und Null-Pointer-Fehlern führen. Für die korrekte Ausführung eines Programmablaufs sollte ein Transaktionssystem daher die Garantie geben, *opacity* einzuhalten.

2.2 Funktionsweise

In einem multi-threaded Programm, in dem mehrere Transaktionen parallel ausgeführt werden sollen, findet die Ausführung der nicht-transaktionalen Programmabschnitte wie gewöhnlich statt. Trifft die Ausführung auf Transaktionen, so werden diese vom Transaktionssystem, unter den oben genannten Eigenschaften, ausgeführt. Dies kann zu einem Konflikt führen, wenn zwei oder mehr Transaktionen auf den gleichen Speicherbereich zugreifen wollen. Sind beide Zugriffe Leseoperationen besteht kein Konflikt und die Transaktionen können erfolgreich weiter ausgeführt werden. Ist aber einer der Zugriffe eine Schreiboperation, so besteht ein Konflikt. Das Transaktionssystem muss diesen Konflikt lösen, da nicht mehr beide Transaktionen erfolgreich abgeschlossen werden können. Die Auflösung des Konflikts sieht im Allgemeinen so aus, dass das Transaktionssystem entscheidet, welche der beiden Transaktionen zurück gerollt und neu gestartet wird und welche ihre Ausführung fortsetzen darf. Kommt eine Transaktion am Ende ihrer transaktionalen Ausführung an, ohne dass es Konflikte gab, dann führt sie ihren Commit aus und ist erfolgreich abgeschlossen. Mit einem erfolgreichen Commit sind die Änderungen dieser Transaktion auch für alle anderen Transaktionen im System sichtbar.

Damit Transactional Memory funktioniert, muss ein Transaktionssystem in irgendeiner Form eine Buchführung über die Speicherzugriffe der Transaktionen betreiben. Hierfür kommen ein Read- und ein Writeset pro Transaktion zum Einsatz. Im Readset wird vermerkt, welche Teile des Speichers gelesen und im Writeset, auf welche zugegriffen wurde. Diese Buchführung dient der Konflikterkennung zwischen Transaktionen und der Versionsverwaltung der Daten im Speicher.
Bei der Konflikterkennung unterscheidet man zwischen *eager* und *lazy* Systemen. In einem Transaktionssystem mit *eager* Konflikterkennung werden Konflikte sofort beim Auftreten erkannt. Wenn eine Transaktion T_1 lesend auf einen Speicherbereich zugreift, der zuvor von Transaktion T_2 beschrieben wurde, sich also in T_2's Writeset befindet, dann wird sofort eine der beiden Transaktionen zurück gerollt. Bei einem umgedrehten Zugriffsszenario kommt es ebenso zu einem Konflikt, der sofort mit dem Abbruch einer der beiden Transaktionen aufgelöst wird. In einem System mit *lazy* Konflikterkennung bleibt so ein Konflikt unerkannt, bis eine Transaktion bei ihrem Commit angekommen ist. Erst in der Commit-Phase wird der Konflikt erkannt und dann auf die gleiche Art und Weise wie beim vorherigen System aufgelöst. Transaktionssysteme mit einer reinen *lazy* Konflikterkennung ermöglichen daher, dass Zombie-Transaktionen bis zu ihrer

Commit-Phase weiter ausgeführt werden. Sie verletzen dann die *opacity*-Eigenschaft
Auch bei der Versionsverwaltung wird zwischen *eager* und *lazy* unterschieden. In einem *eager* System nimmt eine Transaktion Änderungen am Speicher sofort vor und merkt sich den vorherigen Zustand. Kommt es zu einem Konflikt, muss beim Zurückrollen der alte Speicherzustand wieder hergestellt werden. Erreicht eine Transaktion in so einem System die Commit-Phase und darf den Commit durchführen, dann ist sie fertig, da alle Änderungen schon im Speicher stehen. Im Gegensatz dazu nimmt ein System mit *lazy* Versionsverwaltung Änderungen am Speicher nicht sofort, sondern erst in der Commit-Phase vor. Immer, wenn eine Transaktion eine Schreiboperation ausführt, merkt sie sich was sie tun soll und alle diese Schreiboperationen werden in der Commit-Phase gebündelt ausgeführt.

Systeme mit einer *eager* Versionsverwaltung haben schnelle Commits und langsame Rollbacks, da beim Commit nichts mehr zu tun ist und nur beim Rollback der Speicherzustand wiederhergestellt werden muss. Bei *lazy* Systemen ist es genau umgekehrt, da sie bei einem Rollback nur ihre spekulativen Daten verwerfen und beim Commit die Änderungen am Speicher vollziehen müssen. Ein weiterer Unterschied existiert bei den möglichen Konflikten. Bei einem System mit *eager* Versionsverwaltung ist ein Konflikt zwischen zwei Schreiboperationen gegeben. Nehme man an, zwei Transaktionen T_1 und T_2 würden nacheinander einen Wert im Speicher verändern, ohne dabei mit ihrer transaktionalen Ausführung fertig zu werden. T_1 hätte dann den originalen Wert zwischen gespeichert und T_2 den Wert, den T_1 hinterlassen hat. Ist T_2 nun erfolgreich und macht seinen Commit, dann ist der Speicher noch konsistent. Bricht aber danach T_1 ab, dann wird beim Rollback wieder der originale Wert in den Speicher geschrieben und es existiert ein inkonsistenter Speicherzustand. Der Konflikt zweier Schreiboperationen kann bei Transaktionssystemen mit einer *lazy* Versionsverwaltung nur auftreten, wenn diese einen parallelen Commit erlauben. Ist die Ausführung zwar parallel, der Commit aber sequentiell, dann existiert die Problematik nicht. Ein Lese-/Schreibkonflikt und ein Schreib-/Lesekonflikt ist aber in allen Systemen gegeben.

2.2.1 Nesting

Ein großer Vorteil von Transactional Memory gegenüber Locks ist, dass Funktionskompositionen einfach sind. Während man beim Einsatz von Locks auf Deadlocks achten muss, wenn man Funktionen innerhalb eines geschützten Bereichs aufruft, können Transaktionen beliebig verschachtelt werden. Transaktionssysteme kennen hier zwei Strategien, um mit Verschachtelung umzugehen. Entweder es werden alle inneren Transaktionen flach geklopft (*flattening*), so dass sich eine große Transaktion ergibt, oder es wird *checkpointing* betrieben. Bei Letzterem wird bei jedem inneren Transaktionsstart ein Checkpoint angelegt. Dieser Checkpoint wird verworfen, wenn die Transaktion am entsprechenden inneren Transaktionsende ankommt. Benutzt wird der Checkpoint, um zu dem Transaktionsstart zurückzukehren, falls die Transaktion zurückgerollt werden muss. Beide Varianten haben ihre Vor- und Nachteile. Das Flachklopfen hat den Vorteil, dass es einfach umzusetzen und dadurch schnell ist. Es bringt aber einen Neustart der gesamten Trans-

aktion inklusive aller äußeren, mit sich, wenn die Transaktion zurückrollen muss. Das Anlegen von Checkpoints ist hier eleganter, da bei einem Konflikt die Transaktion nur bis zu einem Checkpoint zurückrollt, an dem der Konflikt nicht bestand. Der Nachteil daran ist aber, dass man einen möglicherweise unnötigen Overhead bei jeder inneren Transaktion in Kauf nehmen muss.

2.2.2 Unwiderrufliche Aktionen

Ein Problem existiert bei der Strategie, Transaktionen einfach auszuführen und im Problemfall eine davon zu wiederholen. Dieses Problem ist durch unwiderrufliche Aktionen, wie I/O, gegeben. Eine Ausgabe auf der Konsole oder das Versenden eines Datenpackets kann nicht einfach wieder rückgängig gemacht werden. Das Aufschieben solche Aktionen bis zum Commit ist auch keine Lösung, da dann keine bidirektionale Kommunikation innerhalb von Transaktionen möglich wäre, z. B. die Kommunikation mit einem Webserver. Eine einfache Lösung wäre keine unwiderruflichen Operationen innerhalb von Transaktionen zu erlauben. Somit wäre Transactional Memory aber keine echte Alternative für Locks und Co. mehr. Möchte man weiterhin den kompletten Befehlssatz einer Programmiersprache erlauben, gibt es eine andere Lösung für das Problem. Diese ist der Einsatz eines zweiten Transaktionstyps, der nicht abgebrochen werden kann. Diese unwiderruflichen Transaktionen müssen folglich jeden Konflikt gewinnen. Um dies zu ermöglichen, darf immer nur eine dieser Transaktionen zurzeit im System aktiv sein. Andere spekulativ ausgeführte Transaktionen können weiterhin parallel dazu und zueinander im System aktiv sein. In der Literatur werden diese unwiderrufliche Transaktionen häufig *serial* oder *irrevocable transactions* genannt.

2.3 Umsetzung von Transactional Memory

2.3.1 Software Transactional Memory

Software Transactional Memory (STM) ist die Umsetzung des Transactional Memory Modells als Softwaresystem. Damit STM benutzt werden kann, müssen die transaktionalen Abschnitte eines Programms extra instrumentarisiert werden. Jeder Beginn einer Transaktion und jedes Ende sowie jede Speicheroperation dazwischen müssen durch einen Aufruf an das STM-System ersetzt werden. Dieses betreibt dann mit den erhaltenen Informationen die Buchführung und die damit verbundene Konflikterkennung und -auflösung.
Durch diese zusätzliche Instrumentierung und die Start- und Endkosten für eine Transaktion dauert die Ausführung der Transaktion länger, als wenn man den Programmcode mit fein granulierter Zugriffskontrolle ausführte. Dieser Overhead amortisiert sich bei großen Transaktionen, die auf viele Speicherbereiche zugreifen. Bei kleinen Transaktionen, mit nur wenig Speicherinteraktion, ist dieser Overhead aber gewaltig. In dem Artikel "Software transactional memory: Why is it only a research toy?" [CBM+08] wird sogar behauptet, dass reine Softwaretransaktionssysteme nur in der Wissenschaft eine Rolle

spielen. Das wissenschaftliche Umfeld hat in den letzten 20 Jahren viele Vorschläge für STM-Algorithmen [DDS+10, DSS06, SMP08, DSS10] gemacht, die sowohl *eager* als auch *lazy* Ansätze verfolgen. Sie unterscheiden sich bei der Umsetzung der Konflikterkennung und der Versionsverwaltung.

2.3.2 Hardware Transactional Memory

Hardware Transactional Memory (HTM) ist die Umsetzung des Transactional Memory Modells als Hardwaresystem. Der Gedanke dabei ist, dass das Transaktionssystem eine Hardwarekomponente ist. Diese könnte Teil eines Prozessor oder ein Zusatzchip im Computer sein. Ein Vorteil hierbei gegenüber STM ist, dass die Konflikterkennung und Versionsverwaltung parallel zur regulären Programmausführung laufen. Bis auf die Zeit für Synchronisationen von Speicherzugriffen würde die Transaktion so schnell ausgeführt werden, als wäre sie keine Transaktion. Die Konflikterkennung und Versionsverwaltung finden bei einem HTM-System auf Speicherebene statt. Dieses könnte mit einer Granularität von Speicherzellen oder Cachelines realisiert werden. Ein Vorteil daran ist, dass es unabhängig von der Programmiersprachen funktioniert.
Leider gibt es kein vollständiges Hardwaretransaktionssystem. Die Wissenschaft beschäftigt sich seit längerem mit der theoretischen Umsetzung von HTM. In den letzten 10 Jahren wurden hierzu auch interessante Systeme vorgeschlagen [HWC+04, AAK+06, RHL05, MBM+06, YBM+07]. Die vorgeschlagenen Systeme existieren aber nur als Prof-of-Concept-Prototypen oder sind Prozessor-Simulationen. Die einzigen beiden Hardwaretransaktionssysteme in kommerziellen Prozessoren sind *best effort* HTMs, sie stammen von Intel [Int12] und IBM [WGW+12]. Bei IBM ist ein best-effort HTM-System im BlueGene/Q Prozessor eingebaut. Bei Intel heißt das *best effort* System RTM (Restricted Transactional Memory) und ist Teil der Befehlssatzerweiterung TSX. Diese wurde mit den Hasswell-Prozessoren eingeführt, musste aber aufgrund eines Erratums, einem nicht-lösbaren Fehler, wieder deaktiviert werden. Die Befehlserweiterung steht erst ab den Broadwell-EP- und Skylake-Prozessoren wieder zur Verfügung. Bei den Skylake Prozessoren unterstützen aber nur die leistungsstärksten Modelle TSX. Diese sind namentlich die Prozessoren mit den Modellbezeichnungen 6700, 6600 und 6500 sowohl in ihrer normalen Ausführung, als auch in den Übertakter(K)- und Mobil(T)-Varianten.

Da es nur die beiden Systeme von Intel und IBM kommerziell zu erwerben gibt, wird bei der folgenden Erklärung von *best effort* HTMs immer Bezug auf diese Systeme genommen. *Best effort* HTMs unterstützen nicht den vollen Umfang des Transactional Memory Modells. Es ist mit ihnen aber möglich, Transaktion auszuführen und auch nach den oben definierten Eigenschaften erfolgreich abzuschließen. Es wird hierbei nicht garantiert, dass jede Transaktion abgeschlossen werden kann. Das System gibt, wie der Name sagt, sein bestes. Wenn eine Transaktion nicht erfolgreich ausgeführt werden konnte, gibt das HTM-System die Ausführungskontrolle an einen Softwarehandler ab. Dieser wird vom Programmierer beim Start einer Transaktion mit angegeben und handhabt die weitere Ausführung des Programms. Der Einsatz dieser *best effort* Systeme ist also nicht so trivial, wie vom Transactional Memory Modell eigentlich gedacht, da man

sich alternative nicht-transaktionale Ausführungswege überlegen muss. Dieses Zusammenspiel von Hardwaretransaktionen und nicht-transaktionaler Ausführung erfordert wiederum Überlegungen über Zugriffskontrollen und ist fehleranfällig. Für die Verwendung in Hybrid-Transaktionssystemen, die im nächsten Unterkapitel 2.3.3 beschrieben werden, sind diese HTMs aber sehr nützlich.
Beide *best effort* Systeme arbeiten mit einer *eager* Konflikterkennung und einer *lazy* Versionsverwaltung. Beides wird durch ein Read- und Writeset pro Transaktion im Cache umgesetzt. Diese Sets bestehen aus den Cachelines, auf die eine Transaktion zugegriffen hat. Der Cache führt hierfür mit einer Aliasing-Technik mehrere Versionen der gleichen Cacheline. Möchte eine Transaktion an einer Addresse im Speicher etwas schreiben, so wird die gesamte Cacheline, in der sich die Addresse befindet, mit dem geänderten Wert dem Writeset hinzugefügt. Bei einem Lesezugriff wird die Cacheline, in der sich der Wert befindet, dem Readset hinzugefügt. Kommt eine Transaktion am Ende ihrer Ausführung an und es gab keine Konflikte, dann veröffentlicht sie ihr Writeset und ist damit erfolgreich abgeschlossen.
Ein Konflikt entsteht zwischen zwei Transaktionen, wenn eine Transaktion seinem Writeset eine Cacheline hinzufügt, die sich schon in einem Read- oder Writeset einer anderen Transaktion befindet. Genauso gibt es auch einen Konflikt, wenn eine Cacheline dem Readset hinzugefügt wird, die sich schon in einem Writeset einer anderen Transaktion befindet. Kein Konflikt entsteht bei Read-Read Zugriffen. Eine Cacheline kann sich also in den Readsets von mehreren Transaktionen befinden. Konflikte können aber auch erkannt werden, wenn eigentlich keine existieren (*false-positives*). Dies liegt an der Granularität von Cachelines, mit der die Konflikte erkannt werden. Bei zwei Transaktionen wird ein *false-positive* Konflikt festgestellt, wenn diese während ihrer parallelen Ausführung auf unterschiedliche Speicherbereiche zugreifen, die sich in derselben Cacheline befinden.
Für die Auflösung von Konflikten wird eine *requester wins* Strategie eingesetzt. Dies bedeutet, dass ein Konflikt immer von der Transaktion gewonnen wird, die eine Cacheline als letztes angefordert hat. Diese Konflikterkennung findet nicht nur zwischen Transaktionen, sondern auch zwischen Transaktionen und nicht-transaktionalen Ausführungen statt. Eine nicht-transaktionale Ausführung kann also Transaktionen abbrechen. Dies passiert bei einem Lesezugriff auf eine Cacheline, die sich in einem Writeset befindet, oder bei einem Schreibzugriff auf eine Cacheline, die sich in einem Read- oder Writeset befindet. Die HTM-Systeme erfüllen also eine strenge Isolationseigenschaft. Transaktionen haben umgekehrt keine Auswirkung auf nicht-transaktionale Ausführungen, da Zweitere die Writesets der Transaktionen nicht sehen können und Änderungen von Transaktionen nur nach einem erfolgreichen Commit wahrnehmen.

Die *best effort* Systeme versuchen die Transaktionen nur einmal auszuführen. Ist die Ausführung erfolgreich, setzt die Programmausführung hinter dem Ende der Transaktion fort. Ist die Transaktion nicht erfolgreich, so kommt ein Softwarehandler ins Spiel, der die Transaktion nochmal ausführen oder sich für einen alternativen Ausführungspfad entscheiden kann. Gründe für das nicht erfolgreiche Ausführen von Transaktionen gibt es mehrere. Ein Grund sind die oben genannten Konflikte, die mit anderen Transaktionen

und nicht-transaktionalen Ausführungen auftreten können. Des Weiteren können auch Resourcenbeschränkungen, wie die Größe der Sets oder Kontextwechsel dazu führen, dass eine Transaktion abgebrochen wird. Aber auch die verwendeten Prozessorbefehle haben eine Auswirkung auf den Erfolg von Transaktionen. Um möglichst vielseitig einsetzbar zu sein, machen die beiden HTMs keine Einschränkungen bezüglich der erlaubten Befehle innerhalb von Hardwaretransaktionen. Wie aber im Abschnitt 2.2.2 Unwiderrufliche Aktionen schon beschrieben, gibt es Transaktionen, die nicht spekulativ ausgeführt werden können. Transaktionen, die in diese Kategorie fallen und zum Beispiel I/O verwenden, aber auch Transaktionen, die auf Atomics oder Locks zugreifen, können nie erfolgreich ausgeführt werden.
Die Programmierung des alternativen Softwarepfads, der für gescheiterte Transaktionen gewählt wird, ist nicht trivial. Eine mit Locks arbeitende Software kann nicht einfach durch eines dieser HTM-Systeme beschleunigt werden, indem man alle Locks durch Transaktionen ersetzt und die Lock-Variante als alternativen Pfad angibt. Denn, obwohl die HTMs eine strenge Isolationseigenschaft gewährleisten, kann es trotzdem zu inkonsistenten Zuständen im Speicher kommen, wenn Transaktionen und nicht-transaktionale Ausführungen auf dem gleichen Speicherbereich arbeiten.

Die nachfolgende Abbildung 2.2 enthält ein Beispiel dafür, warum nicht-transaktionale Ausführungen und Hardwaretransaktionen nicht auf den gleichen Speicherbereich zugreifen dürfen. Die Abbildung zeigt eine globale Variable *i*, die von *Thread1* inkrementiert und von *Thread2* dekrementiert wird. *Thread1* hat seine Operationen in einer Hardwa-

```
int i = 0 // global variable
```

Thread1:

```
while (true) {
  if(_xbegin()==_XBEGIN_STARTED)
  {
    i++;
    _xend();
    break;
  }
}
```

Thread2:

```
Mutex.lock();
i--;
Mutex.unlock();
```

Abbildung 2.2: Die parallele Ausführung von *Thread1* und *Thread2* führt zu einem undefinierten Zustand von *i*.

retransaktion verpackt und *Thread2* in einer Lock/Mutex-Zugriffskontrolle. Die Syntax der Hardwaretransaktion entspricht der von RTM im GCC, wobei das _xbegin() die Transaktion startet und _xend() sie committet. Die Umsetzung des Softwarehandlers bei RTM ist so gelösst, dass _xbegin() als Rückgabewert entweder einen erfolgreichen Start oder einen Abbruch signalisiert. Wenn die Transaktion während ihres Laufes abgebrochen wird, springt die Ausführung wieder zum _xbegin() zurück und der Rückgabewert

signalisiert einen Abbruch. Die While-Schleife um die Transaktion sorgt dafür, dass der Thread die Transaktion solange ausführt, bis sie erfolgreich committet hat. Wenn nur mehrere Threads vom gleichen Typ wie *Thread1* arbeiten, dann gibt es zwischen ihnen keine Probleme und i wird konstant hochgezählt. Und auch wenn nur Threads vom Typ *Thread2* ausgeführt werden, ist das Verhalten vorhersehbar. Wenn nun aber *Thread1* und *Thread2* parallel arbeiten, kann das zu unterschiedlichen Ergebnissen führen. Denn die strenge Isolationseigenschaft des HTM-Systems garantiert zwar, dass die Hardwaretransaktion abgebrochen wird, wenn sie eine Änderung durch eine nicht-transaktionale Ausführung wahrnimmt, aber nicht das umgekehrte. Eigentlich würde man nach der Ausführung von beiden Threads das Ergebnis 0 erwarten, da i einmal inkrementiert und einmal dekrementiert wurde. Es kann aber auch das Ergebnis $i = -1$ herauskommen. Denn *Thread2* könnte i, welches 0 ist, lesen und dann unterbrochen werden. Während dieser Unterbrechung führt *Thread1* seine Transaktion erfolgreich aus. Der Wert von i ist 1 nach der Ausführung von *Thread1*. Wenn *Thread2* nun seine Ausführung fortsetzt, dekrementiert er seine lokale Kopie von i und setzt das globale i auf -1. Dieses Ergebnis von i ist unerwünscht aber nicht zu verhindern. Das Beispiel zeigt, dass es nicht so einfach möglich ist, Hardwaretransaktionen parallel mit nicht-transaktionalen Ausführungen laufen zu lassen, wenn diese auf die gleichen Speicherbereiche zugreifen.

Beide HTM-Systeme unterstützen auch das Verschachteln von Transaktionen. Hierbei werden alle inneren Transaktionen durch *flattening* zu einer großen Transaktion zusammengefasst. Befindet sich ein Thread innerhalb einer Transaktion und stößt dann auf einen Transaktionsstart, so erhöht sich nur die Verschachtelungstiefe der gerade laufenden Transaktion. Entsprechend verhält es sich auch beim Transaktionsende. Wenn eine Transaktion an diesem ankommt und die Verschachtelungstiefe angibt, dass es sich um eine innere Transaktion handelt, so wird kein Commit ausgeführt, sondern nur die Verschachtelungstiefe reduziert. Erst bei der äußersten Transaktion wird dann der Commit für die gesamte Transaktion ausgeführt. Es gibt hierbei keine partiellen Rollbacks. Muss eine der inneren Transaktionen abbrechen, beendet dies immer auch die äußeren Transaktionen.
HTM in diesen Systemen bietet auch die Möglichkeit herauszufinden, ob die aktuelle Threadausführung transaktional oder nicht-transaktional ist. Dies ist für die Kombination von HTM und einem komplexen Softwarehandler von Bedeutung. Des Weiteren gibt es auch die Möglichkeit, Hardwaretransaktionen abzubrechen. Diese Abbruchfunktion beendet die aktuelle Transaktion und setzt die Programmausführung beim Softwarehandler fort.

2.3.3 Hybrid Transactional Memory

Hybrid Transactional Memory (HyTM) ist die Umsetzung von Transactional Memory mit einer Kombination aus Hardware- und Softwaretransaktionen. Es ist der Versuch die Vorteile von STM- und HTM-Systemen zu kombinieren, da es kein vollständiges Hardwaretransaktionssystem gibt. Die HTM-Systeme, so wie sie zurzeit zur Verfügung stehen, sind zwar schnell, sie unterstützen aber nicht die erfolgreiche Ausführung aller Transak-

tionen. Und STM-Systeme sind langsam, da sie viele zusätzliche Instruktionen für die Buchführung bearbeiten müssen, sie können dafür aber alle Transaktionen erfolgreich ausführen. Der Gedanke hinter HyTM-Systemen ist nun, dass die kurzen Transaktionen vom HTM-System schnell ausgeführt werden und nur für die restlichen Transaktionen das Softwaresystem bemüht werden muss. Das langsamere Softwaresystem kümmert sich also um die Transaktionen mit unwiderruflichen Aktionen und die Transaktionen, deren Ausführung als Hardwaretransaktion nicht erfolgreich war.
Auch in diesem Feld gibt es unterschiedliche Vorschläge für die Kombination aus Hardware und Software. TMACC [CBO+11] ist ein HyTM-System, das einen FPGA für die Hardwareunterstützung benutzt. Aber dieses System existiert nur auf einer Prototypplattform. Die meisten vorgeschlagenen Systeme verwenden aber eine *best effort* HTM, wie zum Beispiel Invyswell [CGS+14], welches auf RTM aufbaut, BlueGene/Q TM [WGW+12], welches BlueGene/Qs HTM benutzt, und GCCTM [GW12], welches die Transactional Memory Implementierung im GCC ist. Der Ansatz der Kombination mit einer *best effort* HTM scheint der interessantere zu sein, da diese Systeme kommerziell zur Verfügung stehen.
Der grundsätzliche Gedanke von Invyswell, BlueGene/Q TM und GGC-TM ist, dass erst versucht wird, jede Transaktion als Hardwaretransaktion auszuführen, vielleicht auch mehrere Male und wenn das scheitert, dann wird die Transaktion an das Softwaresystem weitergereicht. Über diesen Gedanken hinaus unterscheiden sich die Systeme aber. Denn wie im vorherigen Unterkapitel 2.3.2 HTM beschrieben, gibt es ein Problem, wenn Hardwaretransaktionen und nicht-transaktionale Ausführungen parallel auf den gleichen Speicherbereich zugreifen. Aus der Sicht des Hardwaresystems entsprechen die Softwaretransaktionen beziehungsweise das System, das diese verwaltet, einer nicht-transaktionalen Ausführung.

Im Folgenden werden drei HyTM-Systeme vorgestellt. Das HyTM-System, das im GCC implementiert ist, Invyswell, welches als Vorlage für Prallbrid genommen wurde und BlueGene/Q TM, um der Vollständigkeit halber eine anderes System aufzuzeigen.

GCC-TM

GCC-TM ist ein HyTM-System, das mit dem GCC ab Version 4.7 mitgeliefert wird. Es ist unabhängig von einer speziellen HTM-Implementierung und unterstützt je nach vorhandenem Prozessor entweder Intels RTM oder IBMs BlueGene/Q HTM. Es kann aber auch als reines Softwaretransaktionssystem laufen, wenn keine HTM-Unterstützung vorhanden ist. Das System kennt drei Modi, in denen es Transaktionen ausführen kann. Es werden entweder Hardwaretransaktionen, spekulative Softwaretransaktionen oder eine unwiderrufliche Softwaretransaktion (hier serielle Transaktionen genannt) ausgeführt. Diese drei Modi laufen alle exklusiv und das System befindet sich immer in einem dieser Zustände. Es ist also keine parallele Ausführung von Hard- und Softwaretransaktionen möglich. Dieser Ansatz umgeht das Problem mit dem möglicherweise inkonsistenten Speicherzustand, da der Fall nie auftritt.

Wenn GCC-TM auf einem System mit Unterstützung von Hardware Transactional Memory verwendet wird, dann benutzt es Hardwaretransaktionen für die regulären Transaktionen und serielle Transaktionen für unwiderrufliche Aktionen und *foward progress*. Steht kein HTM zur Verfügung, dann verwendet GCC-TM spekulative Softwaretransaktionen anstatt den Hardwaretransaktionen. Der serielle Modus wird auch in diesem Fall für unwiderrufliche Aktionen und den Fortschritt benutzt. Grundsätzlich versucht GCC-TM so viele Transaktionen wie möglich als spekulative Transaktionen, also Hardware- oder Softwaretransaktionen, auszuführen. Scheitert dies, wird analysiert warum. Wenn der Grund des Scheiterns ein Konflikt war und der Grenzwert an Wiederholversuchen noch nicht überschritten ist, dann wird die Transaktion nochmal ausgeführt. Ansonsten wird sie als serielle Transaktion ausgeführt, um Fortschritt zu garantieren. Da Transaktionen mit unwiderruflichen Aktionen nicht als spekulative Transaktionen ausgeführt werden können, werden diese von Anfang an als serielle Transaktionen ausgeführt. Damit das System eine serielle Transaktion ausführen kann, wechselt es in den seriellen Modus. Der Wechsel aus dem Hardware- oder spekulativen Softwaremodus in den seriellen Modus beendet alle anderen Transaktionen, die parallel ausgeführt wurden. In diesem seriellen Modus kann nur eine Transaktion im ganzen System laufen. Dies ist die Transaktion, die das System in den Modus versetzt hat. Dieser serielle Modus ist durch ein globales Lock umgesetzt, an dem auch alle Transaktionen warten müssen, bis die serielle Transaktion beeendet ist. Weitere Transaktionen werden nur im seriellen Modus ausgeführt, wenn dies zwingend notwendig ist; ansonsten wechselt das System wieder in den Hardware- oder Softwaremodus und versucht die wartenden Transaktionen in diesem auszuführen.
Das GCC-TM-System ist in der Form modular gehalten, da es erlaubt, unterschiedliche, spekulative Softwaretransaktionsalgorithmen zu benutzen. Das System selbst liefert einen Multi-Lock- und einen Global-Lock-Algorithmus mit. Die Umgebungsvariable "ITM_DEFAULT_METHOD" bestimmt, welche STM-Implementierung genommen werden soll. Der Global-Lock-Algorithmus ist nicht so interessant, da auch schon der serielle Modus eine Global-Lock-Variante darstellt. Als Standard ist daher die Multi-Lock-Variante eingestellt. Diese ist eine STM-Implementierung, die eine *eager* Konflikterkennung und eine *eager* Versionsverwaltung besitzt. Sie funktioniert über Orecs (*ownership records*) und erlaubt das parallele Ausführen von mehreren Softwaretransaktionen. Es können aber auch eigene STM-Algorithmen implementiert und eingebunden werden. Diese müssen nicht zwingend *eager* Systeme sein, sondern können auch *lazy* sein. Denn aufgrund der Exklusivität der drei Ausführungsmodi gibt es keine Interaktion mit den Hardware- oder seriellen Transaktionen.

Invyswell

Invyswell [CGS+14] ist ein HyTM-System für Intels RTM. Es ist auch die Grundlage für das in dieser Arbeit vorgestellte Parallbrid, was eine umfangreiche Darstellung des Systems im Vergleich zu dem vorangegangenen System zur Folge hat. Das System hat aber

einen *opacity* Fehler, der am Ende dieses Abschnitts erklärt und bei der Umsetzung in Parallbrid behoben wird. Invyswell kombiniert InvalSTM [GVS10] mit dem HTM-System, welches mit den Haswell-Prozessoren eingeführt wurde. Aus dieser Kombination leitet sich auch der Name ab. Invyswell kennt genauso wie GCC-TM die drei Transaktionsklassen für Hardware-, spekulative Software- und unwiderrufliche Softwaretransaktionen. Es spaltet die Hardware- und unwiderruflichen Softwaretransaktionen aber noch in zwei Typen auf, um mehr Parallelität zu erlauben. Invyswell unterstützt dementsprechend also 5 Transaktionstypen. Diese sind namentlich die beiden Hardwaretransaktionstypen LiteHW und BFHW, der spekulative Softwaretransaktionstyp SpecSW und die beiden unwiderruflichen Softwaretransaktionstypen IrrevocSW und SglSW. Bei der Ausführung von Transaktionen wird auch hier erst versucht, die Transaktion als Hardwaretransaktion auszuführen und nur wenn dies scheitert, wird die Transaktion zu einem der Softwaretypen eskaliert.
Der grundsätzliche Gedanke hinter der parallelen Ausführung von Transaktionen, auch unterschiedlichen Typs, ist, dass eine Transaktion nach ihrem erfolgreichen Commit spekulativ laufende Transaktionen invalidieren kann. Die Invalidation sieht so aus, dass die Transaktion, die ihren Commit abgeschlossen hat, ihr Writeset mit den Readsets der anderen laufenden Transaktionen vergleicht und diejenigen invaldiert, bei denen es eine Schnittmenge zwischen den Sets gibt. Spekulativ laufende Softwaretransaktionen kontrollieren daher bei jeder Speicheroperation, ob sie invalidiert wurden und starten neu, wenn dies der Fall war. Es wird also eine *eager* Konflikterkennung betrieben. Die Versionsverwaltung wird von den spekulativen Transaktionen aber *lazy* gehandhabt. Um Änderungen am Speicher vorzunehmen und anschließend andere Transaktionen zu invalidieren, muss eine Softwaretransaktion ein globales Commit-Lock akquiriert haben. Dieses Commit-Lock wird von den unwiderruflichen Transaktionen zu Begin ihrer Ausführung akquiriert. Die spekulativen Softwaretransaktionen akquirieren das Lock, wenn sie ohne invalidiert zu werden die Commit-Phase erreichen. Durch dieses globale Commit-Lock sind alle Änderungen am Speicher, die von Softwaretransaktionen vorgenommen werden, serialisiert. Es ist zwischen diesen kein paralleler Commit möglich. Die parallele Ausführung von Transaktionen ist aber gestattet.

LiteHW und BFHW sind die beiden Hardwaretransaktionstypen. Sie werden als RTM-Transaktionen ausgeführt, unterscheiden sich aber bei ihrer Laufzeit und der Möglichkeit, parallel zu anderen Transaktionen den Speicher zu verändern. LiteHW ist eine leichtgewichtige Hardwaretransaktion, die keine Möglichkeit zur Konflikterkennung mit den Softwaretransaktionen hat. Es darf daher keine LiteHW-Transaktion ihren Commit ausführen, wenn Softwaretransaktionen aktiv sind, da diese sonst einen inkonsistenten Speicherzustand vorfinden können. Um dies zu gewährleisten, kontrolliert eine LiteHW-Transaktion vor der Ausführung ihres Commits einen Softwarezähler, der von den SpecSW-Transaktionen geführt wird, und das Commit-Lock. Ist der Softwarezähler null und das Commit-Lock frei, dann darf die LiteHW-Transaktion ihren Commit ausführen. Ist das Commit-Lock von einer anderen Transaktion akquiriert oder der Softwarezähler größer als null, dann bricht die LiteHW-Transaktion ab und startet neu. Die strenge Isolationseigenschaft von RTM gewährleistet, dass Änderungen am Commit-Lock

oder dem Softwarezähler von einer LiteHW-Transaktion wahrgenommen werden, wenn die ihren Commit noch nicht ausgeführt hat. LiteHW-Transaktionen können in diesem System also parallel zu allen anderen Transaktionstypen ausgeführt werden, dürfen aber nur parallel zu den anderen Hardwaretransaktionstypen einen Commit ausführen.
BFHW-Transaktionen sind der andere Hardwaretransaktionstyp von Invyswell. Dieser wird auch als RTM-Transaktion ausgeführt. BFHW-Transaktionen führen aber ein zusätzliches Read- und Writeset, um mit den Softwaretransaktionen Konflikterkennung betreiben zu können. Durch diese zusätzlichen Sets ist die Ausführungszeit einer BFHW-Transaktion langsamer als die einer LiteHW-Transaktion. BFHW-Transaktionen sollten daher nur in der Präsenz von Softwaretransaktionen benutzt werden. Durch die Möglichkeit der Konflikterkennung können BFHW-Transaktionen einen Commit ausführen, wenn spekulative Softwaretransaktionen aktiv sind. Diese können dann nämlich nach dem Commit invalidiert werden, wenn es einen Konflikt gab. Es ist auch ein Commit einer BFHW-Transaktion parallel zu dem Commit einer SpecSW-Transaktion beziehungsweise der Ausführung einer IrrevocSW-Transaktion möglich. Hierzu überprüft eine BFHW-Transaktion vor dem Commit den Zustand des Commit-Locks. Ist dieses frei, kann der Commit ausgeführt werden. Ist es von einer anderen Transaktion akquiriert, dann muss die BFHW-Transaktion mit ihrem Read- und Writeset überprüfen, ob es einen Konflikt mit dem Halter des Commit-Locks gibt. Existiert ein Konflikt, bricht die Hardwaretransaktion ab, ansonsten führt sie ihren Commit aus. BFHW-Transaktionen sind in diesem System in der Lage, parallel zu allen anderen Transaktionen ausgeführt zu werden und auch parallel zu allen Transaktionen außer SglSW einen Commit auszuführen.
SpecSW-Transaktionen sind die typischen Softwaretransaktionen in diesem System. Sie werden benutzt, wenn eine Transaktion nicht erfolgreich als Hardwaretransaktion ausgeführt werden konnte. Sie betreiben eine *eager* Konflikterkennung, da sie bei jedem Speicherzugriff prüfen, ob sie noch valide sind. Zusätzlich überprüfen sie bei jedem Lesezugriff, ob es einen Konflikt mit der Transaktion gibt, die das Commit-Lock hält, und warten auf Hardwaretransaktionen, die schon ihren Commit ausgeführt haben, aber noch nicht mit der Invalidierung durch sind. Diese zusätzlichen Prüfungen gewährleisten *opacity*, da eine SpecSW-Transaktion sonst Daten lesen könnte, die von einer Transaktion geändert wurden, die noch nicht mit der Invalidierung fertig ist. Die Änderungen am Speicher, die SpecSW-Transaktionen vornehmen wollen, werden bis zum Commit zwischengespeichert und veröffentlicht, wenn die Transaktion erfolgreich committen darf. Dies entspricht einer *lazy* Versionsverwaltung. SpecSW-Transaktionen können parallel zu LiteHW-, BFHW- und IrrevocSW-Transaktionen ausgeführt werden, sie können aber nur parallel zu BFHW-Transaktionen einen Commit ausführen.
IrrevocSW und SglSW sind die beiden unwiderruflichen Transaktionstypen. Von ihnen kann immer nur eine zurzeit im System aktiv sein und sie werden nie abgebrochen. Neben der Ausführung von unwiderruflichen Aktionen werden sie auch zur Gewährleistung von *forward progress* verwendet. Beide arbeiten mit direkten Speicheränderungen, also einer *eager* Versionsverwaltung. SglSW ist die leichtgewichtigere der beiden Varianten. Sie führt kein Read- und Writeset und kann somit keine Konflikterkennung mit anderen Transaktionen betreiben. Eine parallele Ausführung von anderen Softwaretransaktionen

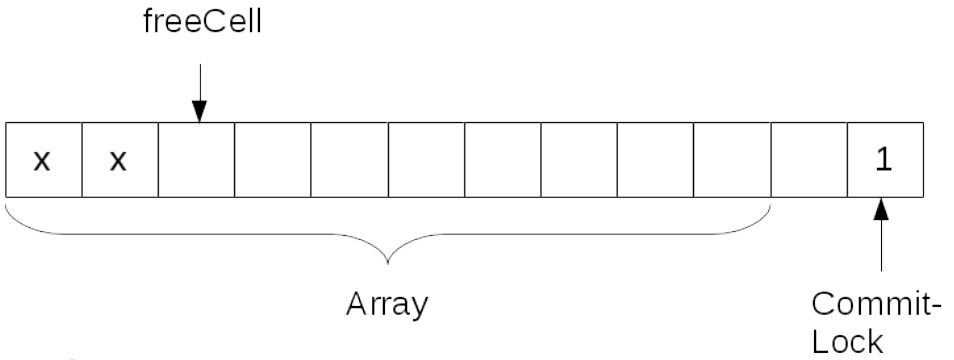
ist parallel zu einer SglSW-Transaktion daher nicht möglich, andere Hardwaretransaktionen dürfen aber ausgeführt werden. Keine der anderen Transaktionen kann aber parallel einen Commit vollziehen. SglSW-Transaktionen sind daher für die Situation gedacht, in der keine anderen Softwaretransaktionen laufen. IrrevocSW ist die langsamere der unwiderruflichen Transaktionen, da sie ein Read- und ein Writeset führt. Diese ermöglichen aber die Invalidierung anderer Transaktionen, wodurch SpecSW-Transaktionen parallel zu einer IrrevocSW ausgeführt werden können. Auch der parallele Commit einer BFHW-Transaktion neben der Ausführung einer IrrevocSW-Transaktion wird durch diese zusätzlichen Sets ermöglicht.
Invyswell ermöglicht also im Vergleich zu GCC-TM das parallele Ausführen von Hardware- und Softwaretransaktionen. Dies wird durch die zusätzlichen Hardware- und unwiderruflichen Softwaretransaktionen ermöglicht, da diese Buchführung betreiben. Leider opfert Invyswell entgegen der Behauptung des Papers [CGS+14] in dem Invyswell vorgestellt wurde, die *opacity* Eigenschaft in Bezug auf die Hardwaretransaktionen. Durch diese fehlende *opacity*-Eigenschaft kann das System nicht garantieren, dass es immer fehlerfrei arbeitet. Das Problem liegt im Zusammenspiel zwischen Hardwaretransaktionen und Transaktionen, die das Commit-Lock beansprucht haben. LiteHW-Transaktionen können nämlich einen Commit machen, auch wenn das Commit-Lock vergeben ist und BFHW-Transaktionen können auch ohne Konflikterkennung ihren Commit vollziehen. Dies geschieht, wenn die Hardwaretransaktionen das Commit-Lock während ihrer Ausführung überschreiben.

Abbildung 2.3 zeigt ein Szenario, in dem eine LiteHW-Transaktion ihren Commit ausführen kann, obwohl das Commit-Lock vergeben ist. Gegeben ist ein Array im Speicher, bei dem ein Pointer auf die nächste freie Zelle zeigt und ein Zähler angibt, wie viele Zellen noch frei sind. Das Commit-Lock liegt durch Zufall genau hinter diesem Array. Die Transaktion T_{irr} ist eine IrrevocSW-Transaktion, die das Commit-Lock akquiriert hat. Im angegebenen Codeabschnitt sieht man einen Ausschnitt der Operationen, die T_{irr} ausführen soll. Diese Operationen nehmen die zwei letzten Elemente aus dem Array, indem sie den Zähler erhöhen und den Pointer verschieben. Die Transaktion T_{htm} ist eine LiteHW-Transaktion, die zehn Nullen in das Array schreiben soll, wenn dafür Platz ist. Danach versucht sie, ihren Commit auszuführen. Für die paralle Ausführung dieser beiden Transaktionen gibt es viele Ausführungspfade, bei denen es keine Probleme gibt und das Verhalten wie gewünscht ist. Es gibt aber auch einen, bei dem es schief geht.
Wenn T_{irr} seine Ausführung bis einschließlich der Operation $freeCount- = 2$ ausführt und dann suspendiert, dann ist der Speicher in einem inkonsistenten Zustand. An sich ist das kein Problem, da SpecSWs eine Konflikterkennung betreiben, bevor sie gelesene Werte verwenden und Hardwaretransaktionen durch das Commit-Lock gestoppt werden sollen; und im Fall von BFHWs dann auch eine Konflikterkennung betreiben. Wird nun aber T_{htm} komplett ausgeführt, während T_{irr} suspendiert, dann schreibt die Transaktion zehn Nullen ab dem $freeCell$ Pointer in den Speicher. Dies ist aus der Sicht von T_{htm} legitim, da $freeCount$ ja den Wert 10 hatte. Da der Pointer aber noch nicht verschoben wurde, schreibt T_{htm} über die Arraygrenze hinaus und überschreibt das Commit-Lock. Wenn T_{htm} nun vor dem Commit prüft, ob das Commit-Lock von einer anderen Transak-

tion akquiriert wurde, dann geht diese Anfrage an das eigene Writeset. Im Writeset wurde das Commit-Lock aber mit einer Null überschrieben, was der Transaktion ein freies Lock signalisiert. T_{htm} führt also seinen Commit aus, obwohl es auf einem inkonsistenten Speicherzustand gearbeitet hat. Dies darf nicht passieren, da es die ACI-Eigenschaften 2.1 von Transaktionen verletzt, die eingehalten werden müssen. Dieses Problem tritt nicht auf, wenn das Commit-Lock nicht von einer Hardwaretransaktion überschrieben werden kann. Dies kann aber nicht allgemein zugesichert werden.
Die beste Lösung für das beschriebene Problem wäre der Einsatz von *escape actions*. Dies sind Operationen, die eine Hardwaretransaktion ohne transaktionalen Kontext ausführt. Die Überprüfung des Commit-Locks mit einer *escape action* würde verhindern, dass der Wert aus dem Writeset genommen wird. Somit würde die Transaktion immer den korrekten Wert für das Commit-Lock lesen, auch wenn es im eigenen Writeset überschrieben wurde. Leider unterstützt keines der momentan verfügbaren HTM-Systeme *escape actions*. Daher ist die einzige Lösung für dieses Problem eine Einschränkung der Parallelität. In Zukunft werden HTM-Systeme möglicherweise *escape actions* unterstützen, sodass wieder ein höherer Grad an Parallelität erreicht werden kann.

BlueGene/Q TM

BlueGene/Q TM [WGW+12] ist speziell auf das HTM System im BlueGene/Q-Prozessor zugeschnitten und nutzt auch eine Kernelerweiterung, um dort zu laufen. Diese Kernelerweiterung mit einem speziellen JailMode, in dem die Transaktionen ausgeführt werden, erlaubt es, das oben genannte Problem zu umgehen. Da dieses System aber nicht auf andere HyTMs übertragbar ist, da es nicht mit jeder *best effort*-HTM funktioniert, wird das System hier nur der Vollständigkeit halber erwähnt. Das System kann Transaktionen als Hardwaretransaktion oder als unwiderrufliche Softwaretransaktion ausführen. Von letzterer darf immer nur eine im System aktiv sein. Die Hardwaretransaktionen können dazu aber parallel ausgeführt werden. Die unwiderrufliche Softwaretranskation ist für alle Transaktionen gedacht, die nicht als Hardwaretransaktion ausgeführt werden konnten oder dort zuhäufig wiederholt wurden. Letzteres garantiert in dem System *forward progress*. Auch *opacity* ist in diesem HyTM-System gegeben, da die Hardwaretransaktionen diese durch ihre Isolationseigenschaft einhalten und die Softwaretransaktion durch den JailMode geschützt ist.



Int freeCount = 8;

freeCell ist ein Pointer auf das erste freie Arrayfeld.

T_{irr}: Nimmt 2 Elemente	T_{HTM}: Setzt 10 Elemente auf 0

`freeCount += 2;`
`// get suspended here`
`freeCell -= 2;`
	`_xbegin()`
	`if (freeCount >= 10) {`
	`  freeCell[0] = 0;`
	`  …`
	`  freecell[9] = 0;`
	`  freeCell +=10;`
	`}`
	`// if commitLock free`
	`if (commitLock == 0)`
	`  _xend();`
	`else`
	`  _xabort();`

Abbildung 2.3: Ein Beispiel, bei dem Invyswell die Consistency-Eigenschaft verletzen kann.

3 Parallbrid

Parallbrid ist ein HyTM-System, das im Kontext dieser Arbeit entwickelt wurde. Der Wunsch bei der Entwicklung von Parallbrid war es, mehr Parallelität bei der Ausführung von Transaktionen zu ermöglichen, als dies bei GCC-TM der Fall ist. Daher wurde Invyswell, das einen hohen Grad an Parallelität ermöglicht, als Vorbild genommen und um ein paar Anforderungen erweitert. Parallbrid ist als Bibliothek für den GCC implementiert und nutzt dessen Compilerintegration für Transactional Memory. Diese Compilerintegration setzt die Vorschläge einer Spracherweiterung für Transactional Memory in C++ [ATSG12] um und ist der Grund für die erweiterten Anforderungen, die über die Fähigkeiten von Invyswell hinaus gehen.
Im Folgenden werden diese neuen Anforderungen (Kapitel 3.1) vorgestellt und die Funktionsweise von Parallbrid (Kapitel 3.2) erläutert. Anschließend werden die einzelnen Transaktionstypen (Kapitel 3.3) beschrieben und erklärt, warum manche von ihnen parallel ausgeführt werden können (Kapitel 3.4).

3.1 Anforderungen

Die Spracherweiterung für C++ "Draft Specification of Transactional Language Constructs for C++" schlägt Befehle und Funktionen für die Benutzung von Transactional Memory in C++ vor. Der GCC unterstützt die Version 1.1 dieser vorgeschlagenen Spracherweiterung ab Compilerversion 4.7. Die Spezifikation definiert die Syntax und Semantik für Transaktions-Statements, Transaktions-Expressions und Funktions-Transaktionsblöcke. Allen liegt ein gemeinsames Transaktionssystem zu Grunde, da sie die gleichen Anforderungen an dieses stellen. Da Transaktions-Statements die häufigste Verwendung von Transaktionen sein dürften, werden anhand dieser die Anforderungen an das TM-System im Folgenden erklärt.

Um Transaktions-Statements zu benutzen, klammert man den Codebereich, der als Transaktion ausgeführt werden soll, mit geschweiften Klammern. Vor der Klammer muss der Befehl stehen, der die Klasse der Transaktion angibt, entweder *__transaction_atomic{}* oder *__transaction_relaxed{}*. Es ist die Aufgabe des Programmierers seine Transaktionen mit der richtigen Klasse zu annotieren und dem Transaktionssystem so etwas über den Inhalt der Transaktionen mitzuteilen.
__transaction_atomic{} kann für Transaktionen verwendet werden, die keine unwiderruflichen Aktionen ausführen. Diese Klasse ist für Transaktionen gedacht, die spekulativ ausgeführt und abgebrochen werden können. Sie hat daher die Beschränkung, dass in dem Block keine I/O-Aktionen oder andere unwiderrufliche Aktionen stehen dürfen.

Diese Transaktionsklasse könnte von Invyswells Hard- und Softwaretransaktionen ausgeführt werden.
__transaction_relaxed{} ist die Transaktionsklasse, die keine Beschränkung ihrer möglichen Befehle kennt. Mit diesen Transaktionen darf der vollständige Funktionsumfang der Sprache C++ genutzt werden. Aktionen wie I/O erfordern aber, dass manche dieser Transaktionen nicht abgebrochen werden. Auch diese Anforderung kann durch die Verwendung von Invyswell und dessen unwiderrufliche Transaktionen erfüllt werden.

Die Spezifikation sieht zusätzlich noch die Möglichkeit eines Useraborts vor. Die Abort-Funktion kann vom Programmierer im Kontrollfluss einer spekulativen Transaktion verwendet werden. Sie bewirkt, dass die Transaktion abgebrochen wird und die Ausführung hinter der Transaktion fortsetzt. Die Transaktion wird hierbei nicht erfolgreich abgeschlossen und nimmt keine Veränderungen am System vor. Bei einem Abort einer eingebetteten Transaktion wird nur diese innere Transaktion abgebrochen. Die äußeren Transaktionen werden dabei weiter ausgeführt. Es gibt aber auch die Möglichkeit von einer inneren Transaktion, den Abbruch aller äußeren Transaktionen zu erzwingen. Hierzu wird dem Abort-Befehl das Attribut *outer* hinzugefügt.
Diese Abort-Funktionalität wird von Invyswell nicht unterstützt. Parallbrid baut daher nur auf Invyswell auf und erweitert es um die Möglichkeit des Useraborts. Hierfür wurde ein weiterer Transaktionstyp und die Möglichkeit zum Checkpointing hinzugefügt.

Auch *progress* muss von einem TM-System, welches die Spezifikation implementiert, gewährleistet werden. Von jeder Transaktion, ob *atomic* oder *relaxed*, wird erwartet, dass sie irgendwann einmal erfolgreich abgeschlossen werden kann. Die einzige Ausnahme bilden Transaktionen, die einen Userabort ausgeführt haben. Transaktionen dürfen also keinen Dead- oder Livelock erfahren. Die Anforderungen an *progress* werden von Invyswell erfüllt. Die Erweiterungen von Parallbrid vervollständigen dieses Fortschrittsverhalten für die Transaktionen mit Abort-Möglichkeit. Dies ist auch der Grund für die Einführung des weiteren Transaktionstyps.

Opacity ist eine weitere Eigenschaft, die von einem TM-System unterstützt werden muss. Denn die Möglichkeit, dass ein Thread durch eine Zombie-Transaktion in eine Endlosschleife läuft oder durch eine Null-Pointer-Exception abstürzt, ist nicht akzeptabel. Invyswell erfüllt *opacity* in vielen Fällen, hat aber ein Problem, das im Abschnitt über Invyswell in Kapitel 2.3.3 schon beschrieben ist. Dieses Problem existiert bei heutigen HTMs ohne Escape-Aktionen immer, wenn Hardwaretransaktionen und nicht-transaktionale Ausführungen parallel auf dem gleichen Speicherbereichen arbeiten. Da Hardwaretransaktionen keine herkömmlichen Synchronisationsmechanismen benutzen können, ist die Synchronisation zwischen diesen beiden Ausführungensarten schwierig. Eine Folge daraus ist, dass nicht-transaktionale Ausführungen und Hardwaretransaktionen nicht gleichzeitig den Speicher verändern dürfen.
Eine HyTM-Implementierung mit einem *eager* Versionsverwaltungs STM-System kann daher keine parallele Ausführung von Hard- und Softwaretransaktionen erlauben. Die *lazy* Versionsverwaltung des Invyswell-STM-Systems erfüllt die gewünschte Anforderung.

Der Grad an Parallelität, der bei Invyswell existiert, muss aber etwas beschnitten werden, um das Problem der parallelen Ausführung zu umgehen. Es dürfen Hardwaretransaktionen parallel zu spekulativen Softwaretransaktionen ausgeführt werden und diese dürfen auch ihren Commit ausführen. Wenn aber eine spekulative Softwaretransaktion in ihre Commit-Phase kommt, dann müssen die Hardwaretransaktionen abbrechen, da sonst *opacity* nicht gewährleistet werden kann. Genauso ist eine parallele Ausführung von unwiderruflichen Transaktionen und Hardwaretransaktionen nicht möglich. Es bleibt aber gegenüber GCC-TM die parallele Ausführung von Hardware- und spekulativen Transaktionen und zwischen spekulativen und unwiderruflichen Transaktionen.
Der Unterschied zwischen Invyswell und Parallbrid ist dabei, dass Invyswell *lazy subscribtion* beim Lesen des Commit-Locks nutzt und Parallbrid *eager subscribtion*, da Hardwaretransaktionen das Commit-Lock zu Beginn ihrer Ausführung lesen.

3.2 Algemeine Funktionsweise

Parallbrid versucht, wie jedes andere HyTM-System, so viele Transaktionen wie möglich als schnelle Hardwaretransaktionen auszuführen. Wenn dies nicht möglich ist, werden die Transaktionen als Softwaretransaktionen ausgeführt. Für die Ausführung von Transaktionen bietet Parallbrid zwei Hardwaretransaktionstypen, einen spekulativen Softwaretransaktionstyp und drei serielle Softwaretransaktionstypen. Mit Ausnahme des neuen Softwaretransaktionstyps, sind die Namen der Transaktionstypen von Invyswell übernommen. Ihre Funktionsweise unterscheidet sich aber an einigen Stellen.
Die zwei Hardwaretransaktionstypen sind LiteHW und BFHW. LiteHW ist eine leicht gewichtige Hardwaretransaktion, die nur parallel zu anderen Hardwaretransaktionen ausgeführt werden kann. Und BFHW ist eine Hardwaretransaktion, die zusätzliche Informationen führt um auch parallel zu den spekulativ laufenden Softwaretransaktionen operieren zu können. Der spekulative Softwaretransaktionstyp heißt SpecSW. SpecSW-Transaktionen sind typische Softwaretransaktionen, die spekulativ ausgeführt und bei Bedarf neugestartet werden können. Sie arbeiten mit einer *lazy* Versionsverwaltung. Die drei seriellen Transaktionstypen sind IrrevocSW, SglSW und SerialAboSW. Sie alle arbeiten mit einer *eager* Versionsverwaltung. IrrevocSW ist ein Transaktionstyp, der nicht abgebrochen werden kann. Er ist für die Ausführung von unwiderruflichen Aktionen da. Eine parallele Ausführung mit anderen seriellen Software- oder Hardwaretransaktionen ist daher nicht möglich. Er führt aber ähnlich wie die BFHW-Transaktionen zusätzliche Informationen, um eine parallele Ausführung mit SpecSW Transaktionen zu ermöglichen. SglSW ist ein weiterer unwiderruflicher Transaktionstyp. Dieser betreibt aber keine Buchführung und kann daher nur alleine im System laufen. Er ist als leichtgewichtige Varante von IrrevocSW-Transaktionen gedacht. Der letzte serielle Transaktionstyp, SerialAboSW, kann keine unwiderruflichen Aktionen ausführen, da er abbrechen und neustarten kann. Er ist für den *forward progress* von Transaktionen mit einem Userabort gedacht. Eine parallele Ausführung ist nur mit SpecSW-Transaktionen möglich.

Man kann die Transaktionen, die von Parallbrid ausgeführt werden können, in drei Klas-

sen einteilen. Spekulative Transaktionen, spekulative Transaktionen mit Userabort und unwiderrufliche Transaktionen. Im folgenden werden die drei Klassen und ihr Ablauf in Parallbrid erläutert. Danach wird der Vorgang der Invalidierung (Kapitel 3.2.4) und des Commits (Kapitel 3.2.5) beschrieben. Anschließend folgt die Konflikterkennung (Kapitel 3.2.6) und Versionsverwaltung (Kapitel 3.2.8) von Parallbrid.

3.2.1 Spekulative Transaktionen

Einfache spekulative Transaktionen sind Transaktionen, die in C++ durch das Schlüsselwort *__transaction_atomic* definiert werden und keinen Userabort enthalten.
Parallbrid versucht so eine Transaktion T_{spec} zuerst als Hardwaretransaktion auszuführen. Hierfür gibt es zwei Möglichkeiten, entweder als LiteHW oder BFHW. Wenn zum Zeitpunkt des Transaktionsstarts SpecSW-Transaktionen ausgeführt werden, dann führt Parallbrid T_{spec} als BFHW-Transaktion aus. Denn BFHW-Transaktionen können parallel zu SpecSW-Transaktionen ausgeführt werden. Werden aber beim Transaktionsstart keine SpecSW-Transaktionen parallel ausgeführt, dann kann T_{spec} als LiteHW-Transaktion gestartet werden. Zweiteres ist schneller, da LiteHW-Transaktionen gegenüber BFHW-Transaktionen keinen zusätzlichen Overhead haben.
Scheitert die Ausführung von T_{spec} als Hardwaretransaktion, dann versucht Parallbrid sie als Hardwaretransaktion zu wiederholen. Dies wird so häufig versucht, bis eine Grenzwert an Wiederholungsversuchen für Hardwaretransaktionen überschritten ist. Das Scheitern einer Hardwaretransaktion ist garantiert, wenn serielle Transaktionen im System aktiv sind. Hat T_{spec} alle seine Wiederholungsversuche als Hardwaretransaktion aufgebraucht, dann eskaliert Parallbrid die Transaktion zu einer SpecSW-Transaktion.
Als SpecSW-Transaktion versucht Parallbrid T_{spec} nun wieder erfolgreich abzuschließen. Ist dies nicht möglich, dann wird T_{spec} bis zum einem Grenzwert an Wiederholversuchen als SpecSW-Transaktion wiederholt. Ist auch dieser Grenzwert erreicht, dann muss die Transaktion zu einer seriellen Transaktion eskaliert werden. Dies ist notwendig, um *foward progress* zu gewährleisten. Die Transaktion kann sonst ewig laufen, ohne jemals ihren Commit erfolgreich auszuführen, zum Beispiel durch *starvation* oder einen *livelock*.
Parallbrid hat nun wieder zwei Möglichkeiten T_{spec} auszuführen, entweder als Irrevocs- oder als SglSW-Transaktion. Sind zu diesem Zeitpunkt SpecSW-Transaktionen aktiv, dann wählt Parallbrid IrrevocSW als Transaktionstyp für T_{spec}, da dieser parallel zu SpecSW ausgeführt werden kann. Sind aber keine SpecSW-Transaktionen aktiv, so wird SglSW gewählt, da dieser serielle Transaktionstyp schneller ist.

3.2.2 Spekulative Transaktionen mit Userabort

Diese Klasse von Transaktionen sind die Transaktionen, die in mindestens einem ihrer Ausführungspfade einen Userabort haben. Sie werden in C++ auch durch das Schlüsselwort *__transaction_atomic* definiert. Diese Klasse von Transaktionen muss aufgrund ihres Useraborts anders behandelt werden als die Klasse der spekulativen Transaktionen. Parallbrid kann diese Transaktionen nicht als Hardwaretransaktionen ausführen. Denn

die Semantik des Useraborts in C++ schreibt vor, dass eine Transaktion, die einen Userabort ausführt, zu dem Beginn dieser Transaktion zurückrollt und sie dann überspringt. Dieses Verhalten ist bei verschachtelten Transaktionen nur mit Checkpointing möglich. Hardwaretransaktionen behandeln verschachtelte Transaktionen aber als Teil der äußeren Transaktion. Es ist damit also nicht möglich eine verschachtelte Transaktion zu überspringen. Das HTM-System hat zwar auch einen Befehl für einen Abort, dieser hat aber eine andere Semantik und bricht immer die komplette Transaktion ab. Er kann damit also nicht das verhalten des C++ Useraborts umsetzen.
Parallbrid führt so eine Transaktion $T_{specAbo}$ daher als SpecSW- oder als SerialAboSW-Transaktion aus. Werden zum Zeitpunkt des Transaktionsstarts noch andere SpecSW-Transaktionen ausgeführt, so versucht Parallbrid $T_{specAbo}$ auch als SpecSW-Transaktion auszuführen. Dies wird im Fall, dass die Ausführung nicht erfolgreich ist, so lange wiederholt, bis ein Grenzwert an Wiederholungen erreicht ist. Um *forward progress* zu garantieren, wird so eine Transaktion in den seriellen Zustand eskaliert. Sie wird dann als SerialAboSW-Transaktion ausgeführt. Dies ist auch der bevorzugte Ausführungstyp, wenn zum Transaktionsstart keine anderen SpecSW-Transaktionen aktiv sind. SerialAboSW-Transaktionen sind seriell, was heißt, dass sie nicht von anderen Transaktionen abgebrochen werden können. Sie können aber im Gegensatz zu den beiden anderen seriellen Transaktionen abbrechen, wenn ein Userabort ausgeführt wird. Eine weitere Eigenheit für eine serielle Transaktion ist, dass SerialAboSW-Transaktionen Checkpointing betreiben und damit partiell zurückgerollt werden können.

3.2.3 Unwiderrufliche Transaktionen

Die dritte Klasse an Transaktionen, die von Parallbrid ausgeführt werden können, sind die unwiderruflichen Transaktionen. Sie beinhalten unwiderrufliche Aktionen und dürfen nicht abgebrochen und wiederholt werden. In C++ wird diese Klasse durch das Schlüsselwort *__transaction_relaxed* definiert.
Parallbrid hat bei der Ausführung dieser Transaktionen die Wahl zwischen IrrevocSW- und SglSW-Transaktionen. Eine Transaktion $T_{irrevoc}$ wird als IrrevocSW-Transaktion ausgeführt, wenn andere SpecSW-Transaktionen parallel ausgeführt werden. Ist dies nicht der Fall, wird $T_{irrevoc}$ als SglSW-Transaktion ausgeführt, da diese schneller ausgeführt werden kann.

3.2.4 Invalidierung

Parallbrid arbeitet mit einem Invalidierungsschema, um SpecSW-Transaktionen, die nicht mehr erfolgreich ausgeführt werden können, zu beenden. Jede spekulativ laufende Softwaretransaktion hat daher ein Invalidierungsflag, das bei jedem Speicherzugriff überprüft wird. Wurde eine Transaktion invalidiert, muss sie abbrechen und neustarten. Kommt eine Transaktion in ihrer Commit-Phase an, dann gab es keine Konflikte und sie darf ihren Commit vollziehen und andere Transaktionen invalidieren. Alle Transaktionstypen, die parallel zu spekulativ laufenden Softwaretransaktionen ausgeführt werden können, betreiben eine Invalidierung. Hierzu überprüft so eine Transaktion nach ihrem

erfolgreichen Commit, ob es Transaktionen gibt, die durch den Commit nicht mehr korrekt weiterlaufen können und invaldiert diese. Für die Hardwaretransaktionen ist eine Invalidierung nicht notwendig, da diese aufgrund der strengen Isolationseigenschaft der HTM-Systeme ihre Ausführung abbrechen, wenn sich im Speicher etwas ändert, auf das sie bereits zugegriffen haben. Die seriellen Transaktionen werden auch nicht invalidiert, dass diese nicht durch andere Transaktionen abgebrochen werden dürfen.

3.2.5 Commit

Wie bereits erwähnt können bei Parallbrid unterschiedliche Transaktionstypen parallel ausgeführt werden. Die Parallelität hat aber ein Ende, wenn es um den Commit geht. Denn nur Hardwaretransaktionen können untereinander parallel einen Commit ausführen. Alle Softwaretransaktionen sind bei ihrem Commit durch ein globales Commit-Lock serialisiert. Dieses Commit-Lock wird von den SpecSW-Transaktionen akquiriert, wenn diese in ihre Commit-Phase eintreten und auch erst wieder freigegeben, wenn sie mit der Invalidierung fertig sind. Die seriellen Transaktionen akquirieren das Commit-Lock gleich zu Beginn ihrer Ausführung und verhindern so, dass SpecSW-Transaktionen ihren Commit beginnen können.

3.2.6 Konflikterkennung

Konflikte werden bei den Softwaretransaktionen von Parallbrid mit einer Granularität von Speicherzellen und bei den Hardwartransaktionen mit einer Granularität von Cachelines erkannt. Die Konflikterkennung der Hardwaretransaktionen untereinander ist durch das HTM-System vorgegeben und ändert sich durch Parallbrid auch nicht. Es wird daher im Folgenden nicht weiter darauf eingegangen.
SpecSW-Transaktionen führen je ein Read- und Writeset, in dem sie ihre Speicherzugriffe in einer Bloomfilter-Struktur (Kapitel 3.2.7) vermerken. Die anderen Transaktionen, die parallel zu SpecSWs ausgeführt werden können, namentlich BFHW, IrrevocSW und SerialAboSW, führen jeweils ein Writeset, das ein Bloomfilter ist. Dieses Writeset dient der Konflikterkennung mit SpecSW-Transaktionen und deren möglicher Invalidierung.
Ein Konflikt zwischen einer Transaktion T und einer SpecSW-Transaktion wird in der Invalierungsphase von T, die die Commit-Phase abschließt, erkannt. Nachdem T seine Änderungen am Speicher abgeschlossen hat, beginnt diese Invalierungsphase. Hierbei vergleicht T sein Writeset mit den Readsets aller laufenden SpecSW-Transaktionen. Gibt es einen Schnitt zwischen dem Writeset und einem Readset, dann hat die SpecSW-Transaktion etwas gelesen, dass von T verändert wurde. Es existiert also ein Konflikt und die SpecSW-Transaktion kann nicht mehr erfolgreich abgeschlossen werden. T setzt daher bei dieser Transaktion das Invalierungsflag, sodass diese Transaktion bei ihrem nächsten Check dieses Flags abbricht. Dieser Check findet bei SpecSW-Transaktionen vor jedem Speicherzugriff statt, sodass nicht auf einem inkonsistenten Speicherzustand gearbeitet wird.

Konflikte werden auf diese Art zwar erkannt, aber es besteht die Möglichkeit, dass sie

erst zu spät erkannt werden und so die *opacity* Eigenschaft nicht eingehalten wird. Dies ist der Fall, wenn eine SpecSW-Transaktion etwas im Speicher liest, nachdem eine Transaktion in ihrer Commit-Phase Änderungen am Speicher vorgenommen, aber bevor sie mit ihrer Invalierungsphase begonnen hat. Um diese Verletzung der *opacity*-Eigenschaft zu verhindern, müssen SpecSW-Transaktionen mehr tun, als nur ihr Invalidierungsflag auszulesen. Sie müssen zusätzlich noch eine Konflikterkennung mit der Transaktion führen, die gerade einen Commit ausführt. Gibt es keinen Schnitt des Readsets mit dem Writeset der Transaktion, die gerade den Commit ausführt, dann gibt es auch keinen Konflikt. Dieses Problem existiert auch im Zusammenspiel mit Hardwaretransaktionen. Diese betreten nach ihrem Commit eine Post-Commit-Phase, in der sie die Invalierungen vollziehen. Ist eine ehemalige Hardwaretransaktion in dieser Post-Commit-Phase, dann müssen SpecSW-Transaktionen warten, bis die Invalidierung abgeschlossen wurde.

3.2.7 Bloomfilter

Parallbrid benutzt für die Read- und Writesets der Transaktionen Bloomfilter. Bloomfilter sind probabilistische Datenstrukturen. Sie bestehen aus einem Bit-Array, das eine feste Größe hat, und einer Anzahl von Hashfunktion. Für die Verwendung von Bloomfiltern als Read- und Writeset wird allerdings nur eine Hashfunktion benötigt. Bloomfilter bieten die Funktionen an, ein Element der Datenstruktur hinzuzufügen und die Mitgliedschaft eines Elements in der Datenstruktur zu überprüfen.
Soll ein Element einem Bloomfilter hinzugefügt werden, so wird von diesem ein Hashwert berechnet und das entsprechende Bit im Array gesetzt. Beim Prüfen, ob ein Element Mitglied der Datenstruktur ist, wird auch wieder der Hashwert berechnet und im Array ausgelesen, ob das entsprechende Bit schon gesetzt ist. Ist das Bit noch nicht gesetzt, dann ist das Element nicht Mitglied des Bloomfilters. Wenn das Bit aber gesetzt ist, dann ist das Element möglicherweise im Bloomfilter. Aufgrund der Hashfunktion und der festen Array-Größe kann nicht mit Sicherheit gesagt werden, ob das Element Mitglied ist, oder ob das Bit durch ein anderes Element gesetzt wurde. Es handelt sich daher um eine probabilistische Datenstruktur.
Der Vorteil von Bloomfilter ist, dass die Funktionen, die ein Bloomfilter bietet, immer in konstanter Zeit ausgeführt werden, unabhängig von der Anzahl der Elemente im Bloomfilter. Im Kontext der Verwendung als Read- und Writeset bedeutet dies, dass eine Überprüfung unabhängig von der Anzahl der gelesenen und geschriebenen Daten ist.
Für die Verwendung als Read- und Writeset wird von den Transaktionen jeweils ein Bloomfilter pro Set geführt. Diese werden während der Ausführung mit den Adressen der gelesenen und geschriebenen Bytes gefüllt. Soll nun überprüft werden, ob es einen Konflikt zwischen zwei Transaktionen gibt, so wird ein Schnitt der beiden Bit-Arrays gebildet. Dieser Schnitt wird durch ein bitweises logisches “Und” der beiden Arrays vollzogen. Besteht das resultierende Array nur aus Nullen, dann haben beide Transaktionen auf unterschiedlichen Speicherbereichen operiert und es existiert kein Konflikt. Existieren aber Einsen in dem Array, dann waren diese Einsen auch in den beiden Sets vorhanden und es existiert ein möglicher Konflikt. Da nicht nachvollziehbar ist, ob es sich um einen

echten Konflikt oder nicht handelt, muss von einem Konflikt ausgegangen werden.
Parallbrid verwendet als Hashfunktion für die Bloomfilter eine vereinfachte Variante der Spooky-Hashfunktion[Jen10] von Jenkins.

3.2.8 Versionsverwaltung

Bei der Versionsverwaltung kombiniert Parallbrid sowohl *eager* als auch *lazy* Versionsverwaltungsstrategien.
Die Hardwaretransaktionen betreiben ihre eigene Versionsverwaltung auf Cache-Ebene. Es handelt sich hierbei um eine *lazy* Versionsverwaltung, bei der die Daten im Speicher nicht verändert werden, bis die Transaktion ihren Commit ausführt. Die spekulativ geschriebenen Daten sind hierbei nur von der Transaktion selbst und nicht von außen einsehbar. Muss die Hardwaretransaktion abbrechen, dann verwirft sie ihre im Writeset vorhandenen Cachelines.
Die SpecSW-Transaktionen betreiben auch eine *lazy* Versionsverwaltung. Sie führen hierzu einen eigenen Writelog, in dem alle spekulativ zu schreibenden Daten zwischengespeichert werden. Dieser Writelog wird bei einem Commit veröffentlicht, indem die entsprechenden Stellen im Speicher durch die Einträge aus dem Log ersetzt werden. Auch während der Ausführung werden Daten aus dem Writelog benutzt, falls etwas gelesen werden soll, das von derselben Transaktion bereits geschrieben wurde. Bricht eine SpecSW-Transaktion ab, so verwirft sie den Writelog und startet neu.
Die beiden unwiderruflichen Transaktionstypen, IrrevocSW und SglSW, benutzen eine *eager* Versionsverwaltung. Sie nehmen Änderungen während ihrer Ausführung direkt im Speicher vor. Einen Writelog oder Undolog müssen diese Transaktionen nicht führen, da ihre Ausführung nicht abgebrochen werden kann und sie somit immer erfolgreich abgeschlossen wird.
Der letzte Transaktionstyp, SerialAboSW, verwendet auch eine *eager* Versionsverwaltung. Diese Transaktionen müssen aber einen Undolog führen, obwohl sie nicht durch andere Transaktionen beendet werden können. Denn SerialAboSW-Transaktionen können durch einen Userabort komplett oder partiell abgebrochen werden. In diesem Fall müssen sie die Änderungen, die sie schon im Speicher vorgenommen haben, wieder rückgängig machen. Dieser Undolog muss aufgrund des möglichen partiellen Abbruchs durch Checkpointing partitioniert sein.

3.3 Die Transaktionstypen

Im Folgenden werden die sechs Transaktionstypen von Parallbrid vorgestellt. Für jeden dieser Transaktionstypen werden die zusätzlichen Instrumentierungen angegeben und erklärt.

3.3.1 SpecSW

SpecSW-Transaktionen sind die typischen Softwaretransaktionen von Parallbrid. Sie werden für spekulative Transaktionen mit und ohne Userabort verwendet. SpecSW-Transaktionen können parallel zu BFHW-, IrrevocSW- und SerialAboSW-Transaktionen ausgeführt werden. Im Folgenden wird die zusätzliche Instrumentierung einer SpecSW-Transaktion T beim Start, Schreiben, Lesen und Commit beschrieben.

Start einer Transaktion
Wenn T gestartet werden soll, dann muss als erstes überprüft werden, ob diese ausgeführt werden darf. Dies ist der Fall, wenn keine SglSW-Transaktion aktiv ist. Um dies zu gewährleisten, gibt es den globalen Zähler *commit_ sequence*. Dieser wird von einer startenden SglSW-Transaktion auf einen ungraden Wert erhöht und bei ihrem Commit wieder auf einen geraden Wert erhöht. Wenn also der *commit_ sequence*-Zähler einen graden Wert hat, dann darf eine SpecSW-Transaktion starten. Um auch spätere Starts von SglSW-Transaktionen mitzubekommen, wird der Wert des Zählers von jeder SpecSW-Transaktion lokal gespeichert. Wenn T starten durfte, dann erhöht sie einen globalen Zähler *sw_ count*, der die Anzahl der laufenden Softwaretransaktionen angibt. Dieser Zähler dient Parallbrid für die Entscheidung beim Start einer Hardwaretransaktion oder unwiderruflichen Transaktion die richtige zu wählen. Zusätzlich beendet eine Veränderung dieses Zählers alle LiteHW-Transaktionen. Denn die dürfen nicht parallel zu SpecSW-Transaktionen ausgeführt werden und überprüfen den Wert daher bei ihrem Start.

Schreibzugriff
Soll T einen Wert in den Speicher schreiben, dann überprüft sie erst ihr Invalidierungsflag, ob sie überhaupt noch ausgeführt werden soll. Wurde sie invalidiert, dann bricht sie ab. Ansonsten fügt sie die Adresse, an die geschrieben werden soll, ihrem Writeset hinzu. Danach wird ein Paar aus Adresse und dem zu schreibenden Wert in den Writelog geschrieben und die Ausführung der Transaktion fortgesetzt.

Lesezugriff
Wenn eine T einen Wert lesen soll, dann sind die folgenden Schritt notwendig. Als erstes wird die Adresse dem Readset hinzugefügt. Erst danach wird der Wert aus dem Speicher gelesen. Dreht man diese Reihenfolge um, dann kann es zu einer Verletzung der *opacity*-Eigenschaft kommen. Denn T kann den Wert lesen und dann unterbrochen werden. Eine andere Transaktion kann in der Zwischenzeit den Wert im Speicher ändern und den Invalidierungsprozess durchführen. Da die Adresse aber noch nicht im Readset steht, wird T nicht invalidiert und arbeitet mit einem falschen Wert weiter.
Nachdem nun also die Adresse dem Readset hinzugefügt und der Wert aus dem Speicher gelesen wurde, findet eine Validierung statt. Diese dient der Einhaltung von *opacity*. Also erstes wird bei dieser Validierung der oben erwähnte *commit_ sequenze*-Zähler überprüft. Wenn dieser sich geändert hat, dann wird oder wurde eine SglSW-Transaktion seit dem Start von T ausgeführt. In diesem Fall muss T abbrechen.

Falls eine Transaktion T_{commit} das Commit-Lock hält, dann muss als nächstes mit dieser eine Konflikterkennung betrieben werden. Diese ist notwendig, da T sonst auf einem inkonstistenten Datenbestand arbeiten kann. Denn T_{commit} kann die Werte an zwei Adressen a und b verändert und noch nicht den Invalidierungsprozess beendet haben. T kann zuvor den alten Wert von a und nun den neuen von b gelesen haben. Da der Invalidierungsprozess nocht nicht abgeschlossen ist, kann T weiter ausgeführt werden und so mit zwei nicht zusammengehörenden Werten arbeiten.
Beim dritten Schritt der Validierung wird überprüft, ob es Hardwaretransaktionen gibt, die in ihrer Invalidierungsphase sind. Ist dies der Fall, muss darauf gewartet werden, dass diese die Invalidierung abschließen. Ansonsten könnte die SpecSW-Transaktion einen inkonsistenten Speicherzustand lesen.
Der letzte Schritt der Validierung ist das Auslesen des Invalidierungsflags. Wurde dieses gesetzt, muss T abbrechen. Ansonsten darf T mit seiner Ausführung fortsetzen und den Wert verwenden, der bei dieser Leseoperation aus dem Speicher geholt wurde.

Commit
Wenn T versucht einen Commit auszuführen, dann wird als erstes überprüft, ob es sich um eine rein lesende Transaktion handelt. Ist dies der Fall, dann kann der Commit sofort erfolgreich abgeschlossen werden. Handelt es sich aber um eine Transaktion, die auch Veränderungen am Speicher vornehmen will, dann muss sie sich um das Commit-Lock bewerben. Hat T das Commit-Lock akquiriert, dann folgt als nächstes eine Validierung. Diese verläuft wie bei einem Lesezugriff, mit der Ausnahme, dass keine Konflikterkennung mit der Transaktion gemacht wird, die das Commit-Lock hält, da T dies selbst ist. Ist das Ergebnis dieser Validierung, dass T invalidiert wurde, dann gibt T das Commit-Lock wieder frei und startet neu. Ansonsten darf T die spekulativen Werte aus dem Writelog an die entsprechenden Adressen im Speicher schreiben. Danach findet der Invalidierungsprozess statt, bei dem T alle anderen SpecSW-Transaktionen invalidiert, die einen Konflikt mit T haben. Als letzter Schritt wird dann das Commit-Lock wieder freigegeben und der *sw_ count*-Zähler dekrementiert.

Verschachtelte Transaktionen
Wenn T in seiner transaktionalen Ausführung ist und auf den Beginn einer Transaktion trifft, dann handelt es sich um eine innere Transaktion. Für diese verfolgen SpecSW-Transaktionen zwei Strategien. Wenn die innere Transaktion keinen Userabort enthält, dann kann T diese einfach in seine bisherige Transaktion integrieren. Es wird also *flattening* benutzt. Um herauszufinden, wann ein Commit zu einer inneren oder der äußersten Transaktion gehört, führt T einen *nesting*-Zähler. Dieser wird bei jedem Start einer inneren Transaktion inkrementiert und bei dem entsprechenden Commit wieder dekrementiert.
Handelt es sich bei der inneren Transaktion allerdings um eine Transaktion mit Userabort, dann darf T sie nicht einfach in die äußere integrieren. In diesem Fall legt T beim Betreten der Transaktion einen Checkpoint an. Dieser Checkpoint enthält eine Kopie des Read- und Writesets und einen Verweis, wie viel des Writelogs zu der äußeren Transaktion gehört. Zusätzlich enthält der Checkpoint noch Informationen, die von dem

longjmp-Befehl benötigt werden, um die Ausführung hinter der Transaktion fortzusetzen. Wird die innere Transaktion erfolgreich abgeschlossen oder durch einen Konflikt beendet, dann wird der Checkpoint verworfen. Kommt es aber zu einem Userabort, dann wird der Checkpoint benutzt, um die alten Buchführungsinformationen wieder herzustellen und die Ausführung nach dem Ende der Transaktion fortzusetzen.

3.3.2 BFHW

BFHW-Transaktionen sind die Hardwaretransaktionen von Parallbrid, die durch zusätzliche Instrumentierung auch parallel zu SpecSW-Transaktionen ausgeführt werden können. Sie können spekulative Transaktionen ohne Userabort ausführen. Im Folgenden werden die zusätzlichen Instrumentierung einer BFHW-Transaktion T beim Start, Schreiben und Commit beschrieben. Das Lesen von Werten ist nicht extra instrumentiert. Und auch bei verschachtelten Transaktionen werden keine zusätzlichen Befehle ausgeführt, da das HTM-System nur *flattening* unterstützt.

Start einer Transaktion
Die BFHW-Transaktion T startet ihre transaktionale Ausführung, indem sie eine Hardwaretransaktion des HTM-Systems startet. Als erste Aktion in dieser Transaktion liest T das Commit-Lock aus. Ist dieses von einer Transaktion akquiriert, dann muss T sofort wieder abbrechen, da BFHW-Transaktionen nicht ausgeführt werden dürfen, während Änderungen am Speicher vorgenommen werden. Ist das Commit-Lock frei, dann darf T mit der Ausführung der Transaktion beginnen. Durch das Lesen des Commit-Locks wird T auch im späteren Verlauf sofort abgebrochen, wenn eine Transaktion das Commit-Lock akquiriert. Dies liegt daran, dass das Commit-Lock dem Readset von T hinzugefügt wird, und darüber hinaus garantiert die strenge Isolationseigenschaft von HTMs, dass Hardwaretransaktionen abbrechen, wenn sich etwas im Speicher ändert, das auch im Readset vorhanden ist. Dieses Vorgehen wird auch *eager subscription* genannt und verhindert die Probleme von *lazy subscription* in Invyswell.

Schreibzugriff
Bei einem Schreibzugriff fügt das HTM-System die zu schreibenden Cachelines dem Hardware-Writeset von T hinzu. Dies passiert automatisch im Prozessor und Parallbrid hat keinen Einfluss darauf. Zusätzlich wird die Adresse, an die geschrieben werden soll, auch dem Bloomfilter-Writeset von T hinzugefügt. Dieses Writeset dient später der Invalidierung von SpecSW-Transaktionen. Es ist während der Ausführung von T ein Teil des Hardware-Writesets und wird erst beim Commit im Speicher sichtbar. Hardwaretransaktionen müssen daher keine Aufräumarbeiten ausführen, wenn sie Abbrechen müssen.

Commit
Soll T seinen Commit ausführen, dann erhöht sie den globalen *hw_post_commit*-Zähler. Dieser gibt an, wie viele fertige Hardwaretransaktionen noch die Invalidierung ausführen müssen. Danach versucht T dann den Hardware-Commit-Befehl auszuführen. Scheitert der Commit, dann bricht T ab und alle Änderungen, inklusive der Inkrementierung des

Zählers, werden verworfen. Ist der Commit erfolgreich, dann werden alle Änderungen von T für alle anderen Transaktionen sichtbar. Die von T ausgeführte Transaktion ist somit abgeschlossen. T führt als nächstes aber noch die Invalidierung aus, um SpecSW-Transaktionen zu beenden, die durch den Commit ungültig geworden sind. Erst hier nach wird der *hw_post_commit*-Zähler wieder dekrementiert. Diese Dekrementierung wird mit einer neuen, kleinen Hardwaretransaktion erledigt. Dies ist notwendig, damit keine Veränderungen am Zähler von Transaktionen, die ihn inkrementieren und dekrementieren, verloren gehen.

3.3.3 IrrevocSW

IrrevocSW-Transaktionen sind unwiderrufliche Transaktionen, die parallel zu SpecSW-Transaktionen ausgeführt werden können. Sie können für spekulative Transaktionen ohne Userabort oder unwiderrufliche Transaktionen genutzt werden. Bei ihnen sind, wie bei BFHW-Transaktionen, der Start, das Schreiben und der Commit zusätzlich instrumentiert. Beim Lesen sind keine zusätzlichen Befehle notwendig.

Start einer Transaktion
Beim Start einer IrrevocSW-Transaktion akquiriert diese das Commit-Lock. Wenn sie dieses erhalten hat, darf sie mit ihrer Ausführung beginnen.

Schreibzugriff
Bei einem Schreibzugriff wird die Adresse, an der etwas geschrieben werden soll, dem Writeset hinzugefügt. Danach ändert eine IrrevocSW-Transaktion die Daten im Speicher direkt. Dies darf sie machen, da sie das Commit-Lock hält und keine andere Transaktion parallel Änderungen am Speicher vornehmen kann. Das Hinzufügen der Adresse zum Writeset muss vor dem Ändern des Speichers geschehen. Denn nur so kann die *opacity*-Eigenschaft für SpecSW-Transaktionen, die bei ihrer Validierung eine Konflikterkennung mit der IrrevocSW-Transaktion machen, gewährleistet werden.

Commit
Der Commit einer IrrevocSW-Transaktion besteht nur aus dem Invalidieren von parallel laufenden SpecSW-Transaktionen, mit denen es einen Konflikt gibt. Danach gibt die Transaktion das Commit-Lock frei und ist damit abgeschlossen.

Verschachtelte Transaktionen
Verschachtelte Transaktionen werden von einer IrrevocSW-Transaktion durch *flattening* in die äußere Transaktion integriert. Dies ist kein Problem, da unwiderrufliche Transaktionen keinen Userabort beinhalten dürfen.

3.3.4 SerialAboSW

SerialAboSW-Transaktionen sind Transaktionen, die für den *forward progress* von spekulativen Transaktionen mit Userabort genutzt werden. Sie können nicht durch andere

Transaktionen invalidiert werden, da sie, wie die unwiderruflichen Transaktionen, das Commit-Lock für ihre gesamte Ausführungszeit besitzen. Um die Funktionalität des Useraborts und die parallele Ausführung von SpecSW-Transaktionen zu ermöglichen, sind der Start, das Schreiben und der Commit bei diesen Transaktionen mit zusätzlichen Befehlen versehen.

Start einer Transaktion
Wenn eine SerialAboSW-Transaktion gestartet wird, muss sich diese um das Commit-Lock bewerben. Hat die Transaktion dieses akquiriert, dann darf sie mit ihrer Ausführung beginnen.

Schreibzugriff
Bei einem Schreibzugriff fügt die Transaktion die Adresse, an der geschrieben werden soll, seinem Writeset hinzu. Dieses hat die gleichen *opacity* Gründe - wie bei IrrevocSW-Transaktionen. Danach werden die im Speicher vorhandenen Daten an der Adresse in einem Undolog gesichert. Der Undolog wird bei einem Abort benutzt, um den alten Zustand des Speichers wieder herzustellen. Nach der Sicherung der alten Daten ändert die Transaktion den Speicher direkt.

Commit
Bei dem Commit einer SerialAboSW-Transaktion invalidert diese alle im System laufenden SpecSW-Transaktionen, mit denen es einen Konflikt gibt. Danach gibt die Transaktion das Commit-Lock wieder frei und ist damit abgeschlossen.

Verschachtelte Transaktionen
Verschachtelte Transaktionen können aufgrund des möglichen Useraborts nicht durch *flattening* behandelt werden. Eine SerialAboSW-Transaktion legt daher bei jeder inneren Transaktion einen Checkpoint an. Dieser Checkpoint beinhaltet eine Kopie des Writesets, einen Verweis auf die Position im Undolog und Informationen für den *longjmp*. Kommt es zu keinem Userabort bis zum Ende der eingebetteten Transaktion, so wird der Checkpoint verworfen. Im Falle eines Useraborts wird dieser aber benutzt, um die entsprechenden Informationen wieder herzustellen und die Ausführung hinter der Transaktion fortzusetzen.

3.3.5 LiteHW

LiteHW-Transaktionen sind der zweite, einfachere Hardwaretransaktionstyp in Parallbrid. Diese Transaktionen können nur parallel zu anderen Hardwaretransaktionen ausgeführt werden. Sie bedürfen daher keiner besonderen Instrumentierung bei ihrer Ausführung, da das HTM-System für ihre Korrektheit sorgt. Einzig beim Start müssen sie zusätzliche Befehle ausführen, um sicherzustellen, dass sie ausgeführt werden dürfen.

Start einer Transaktion
Beim Start einer LiteHW-Transaktion wird eine Hardwaretransaktion des HTM-Systems

gestartet. Danach werden das Commit-Lock und der globale *sw_ count*-Zähler gelesen. Ist das Commit-Lock frei, so ist keine serielle Transaktion aktiv und auch keine SpecSW in der Commit-Phase. Dies muss gegeben sein, damit die LiteHW-Transaktion mit ihrer Ausführung beginnen darf. Ist dies nicht der Fall, dann bricht sie sofort ab. Der *sw_ count*-Zähler erfüllt eine ähnliche Rolle. Ist dieser ungleich Null, so darf die LiteHW-Transaktion nicht ausgeführt werden, da SpecSW-Transaktionen aktiv sind. Sind beide Bedingungen erfüllt, dann startet die LiteHW-Transaktion mit ihrer Ausführung. Ändert sich einer der beiden Werte zur Laufzeit der Transaktion, dann bricht diese aufgrund der Isolationseigenschaft des HTM-Systems ab. Es ist somit sichergestellt, dass LiteHW-Transaktionen nie parallel zu Softwaretransaktionen ausgeführt werden.

3.3.6 SglSW

SglSW-Transaktionen sind der zweite unwiderrufliche Transaktionstyp in Parallbrid. Diese Transaktionen können für die gleichen Transaktionsklassen benutzt werden wie IrrevocSW-Transaktionen. SglSW-Transaktionen erlauben aber keine parallele Ausführung von anderen Transaktionstypen. Um diese Exklusivität zu gewährleisten, sind der Start und der Commit instrumentiert.

Start einer Transaktion
Bei dem Start einer SglSW-Transaktion akquiriert diese das Commit-Lock. Somit verhindert sie die Ausführung aller Transaktionstypen außer SpecSW-Transaktionen. Um auch diese an ihrer Ausführung zu hindern oder abzubrechen, inkrementiert eine SglSW-Transaktion im zweiten Schritt den globalen *commit_ sequence*-Zähler. Dieser bekommt durch die Inkrementierung einen ungerade Wert, was den SpecSW-Transaktionen signalisiert, dass sie nicht ausgeführt werden dürfen.

Commit
Beim Commit einer SglSW-Transaktion inkrementiert diese den *commit_ sequence*-Zähler wieder auf einen geraden Wert. Danach gibt sie das Commit-Lock frei und ist fertig mit ihrer Ausführung.

3.4 Parallele Ausführung - Korrektheit

Parallbrid erlaubt die parallele Ausführung unterscheidlicher Transaktionstypen. Allerdings können nicht alle Transaktionstypen parallel zu jedem Transaktionstyp ausgeführt werden. Dies liegt einerseits an den Transaktionstypen selbst, wenn diese keine Buchführung betreiben, anderseits aber auch an dem Zusammenspiel von Hardwaretransaktionen und nicht-transaktionalen Ausführungen.
In Tabelle 3.1 ist zusammengefasst, welche Transaktionstypen mit welchen parallel ausgeführt werden können. Im folgenden wird für jede Kombination argumentiert warum dies erlaubt ist.

	LiteHW	BFHW	SpecSW	IrrevocSW	SerialAboSW	SglSW
LiteHW	x	x	-	-	-	-
BFHW	x	x	x	-	-	-
SpecSW	-	x	x	x	x	-
IrrevocSW	-	-	x	-	-	-
SerialAboSW	-	-	x	-	-	-
SglSW	-	-	-	-	-	-

Tabelle 3.1: Transaktionstypen, die parallel ausgeführt werden können.

LiteHW vs BFHW

LiteHW- und BFHW-Transaktionen sind Hardwaretransaktionen. Für ihre korrekte parallele Ausführung sorgt das verwendete HTM-System. Es können daher so viele LiteHW- und BFHW-Transaktionen parallel ausgeführt werden und auch committen, wie das HTM-System erlaubt. Typischerweise sind dies eine pro physikalischen oder logischen Kern.

BFHW vs SpecSW

BFHW- und SpecSW-Transaktionen dürfen parallel ausgeführt werden, können aber nicht parallel ihren Commit ausführen. BFHW-Transaktionen arbeiten spekulativ und verändern den Inhalt des Speicher während ihrer Laufzeit nicht. Sie merken sich die Veränderungen am Speicher in ihrem Hardware-Writeset, welches nur von ihnen selbst eingesehen werden kann. Der Inhalt des Writesets einer BFHW-Transaktion wird erst für das System sichtbar, wenn eine BFHW-Transaktion ihren Commit ausführt. SpecSW-Transaktionen arbeiten auch spekulativ und verändern den Speicher nur, wenn sie ihren Commit machen dürfen. Diese *lazy* Versionverwaltung erlaubt es, die beiden Transaktionstypen parallel auszuführen, da diese immer einen Speicherzustand sehen, als wären sie die einzige aktive Transaktion.
Eine BFHW-Transaktion kann einen Commit ausführen, während parallel SpecSW-Transaktionen aktiv sind. Hiernach ändert sich der Zustand des Speichers und es kann SpecSW-Transaktionen geben, die nicht mehr korrekt weiter ausgeführt werden können. Damit die *opacity*-Eigenschaft von SpecSW-Transaktionen durch diesen Commit nicht verletzt wird, betreiben BFHW-Transaktionen Invalidierung von SpecSW-Transaktionen. Letztere müssen darauf warten, dass die Invalidierung abgeschlossen ist, bevor sie ihre Ausführung fortsetzen dürfen.
Eine SpecSW-Transaktion kann auch einen Commit machen, wenn sie parallel zu BFHW-Transaktionen ausgeführt wird. Sie beendet aber die Ausführung von allen laufenden BFHW-Transaktionen, wenn sie in der Commit-Phase das Commit-Lock akquiriert. Dies liegt daran, dass die BFHW-Transaktionen das Commit-Lock zum Start ihrer Ausführung lesen und abbrechen, sobald sich dessen Zustand ändert. Die Beendung aller BFHW-Transaktionen ist leider notwendig, da sonst das Problem von Invyswell 2.3.3

auftreten kann, welches durch *lazy subscription* zustande kommt.

SpecSW vs SpecSW

SpecSW-Transaktionen können untereinander parallel ausgeführt werden, da diese das System nur bei ihrem Commit verändern. Solange also kein Commit stattfindet, sieht jede der parallel laufenden Transaktionen den Speicher so, als wenn sie die einzige Transaktion wäre. Der Commit von SpecSW-Transaktionen ist durch das Commit-Lock serialisiert, sodass auch hier keine Probleme auftreten, was die Veränderung des Speichers angeht.
Durch den Commit einer Transaktion kann es sein, dass anderen SpecSW-Transaktionen nicht mehr erfolgreich abgeschlossen werden können. Die entsprechenden Transaktionen werden daher von der committenden Transaktion invalidert. Um eine Verletzung der *opacity*-Eigenschaft zu verhindern, die in dem Zeitfenster zwischen der Veränderung des Speichers und dem Invalidierungsprozess auftreten kann, betreiben die SpecSW-Transaktionen eine Konflikterkennung mit der Transaktion, die das Commit-Lock hält. Diese Konflikterkennung findet, wie im Abschnitt 3.3.1 über SpecSW-Transaktionen beschrieben, in dem Validierungsprozess bei jedem Lesezugriff statt.

SpecSW vs IrrevocSW

SpecSW-Transaktionen können parallel zu einer IrrevocSW-Transaktion ausgeführt werden. Sie können aber keinen Commit ausführen, während eine IrrevocSW-Transaktion aktiv ist, da diese das für den Commit nötige Commit-Lock hält. Die Ausführung der IrrevocSW-Transaktion wird durch die parallelen SpecSW-Transaktionen nicht gestört, da diese den Speicher nicht verändern können.
Die SpecSW-Transaktionen bekommen aber die Änderungen am Speicher durch die IrrevocSW-Transaktion mit. Dies liegt daran, dass IrrevocSW-Transaktionen den Speicher direkt während ihrer Ausführung verändern. Diese *eager* Versionsverwaltung bereitet den SpecSW-Transaktionen Probleme. Daher ist die Konflikterkennung mit der Transaktion, die das Commit-Lock hält, der IrrevocSW-Transaktion, in der Validierungsphase bei lesenden Speicherzugriffen wichtig. Wird hier ein Konflikt festgestellt, dann hat die IrrevocSW-Transaktion etwas verändert, das von der SpecSW-Transaktion gelesen wurde. Diese muss aus *opacity*-Gründen daher abbrechen. Die von der IrrevocSW-Transaktion vollzogene Invalidierung in ihrer Commit-Phase informiert auch die SpecSW-Transaktionen, die einen Konflikt mit dieser haben, aber keine Validierung zur Laufzeit der IrrevocSW-Transaktion betrieben haben.

SpecSW vs SerialAboSW

SpecSW-Transaktionen können parallel zu einer SerialAboSW-Transaktion ausgeführt werden. Sie können aber, genauso wie mit einer parallel laufenden IrrevocSW-Transaktion, keinen Commit ausführen, wenn eine SerialAboSW ausgeführt wird. Diese besitzt nämlich das Commit-Lock. SerialAboSW-Transaktionen werden durch die parallele Ausführung von SpecSW-Transaktionen nicht gestört, da diese den Speicher nicht verändern

können.
Umgekehrt ist es wie beim Zusammenspiel von SpecSW- und IrrevocSW-Transaktionen. SpecSW-Transaktionen sind für ihre korrekte Ausführung auf die Konflikterkennung während ihrer Validierung und auf die Invalidierung beim Commit der SerialAboSW-Transaktion angewiesen. Eine Besonderheit bei SerialAboSW-Transaktionen ist, dass sie partiell oder vollständig durch einen Userabort abbrechen können. Hierbei Stellen sie den alten Speicherzustand durch Sbwicklung ihres Undologs wieder her. Damit hierbei Konflikte zwischen SpecSW- und SerialAboSW-Transaktionen korrekt erkannt werden, muss die SerialAboSW-Transaktion erst den Speicher zurückändern und erst danach das eigene Writeset auf den Zustand vor der abgebrochenen Transaktion zurücksetzen.

4 Implementierung

Die Implementierung von Parallbrid ist als geteilte Bibliothek realisiert. Der Grund dafür ist, dass die Compilerintegration für Transactional Memory des GCCs genutzt wird. Der GGC-Compiler übersetzt Programme, die die transaktionalen Sprachkonstrukte für C++ benutzen. Bei dieser Übersetzung instrumentiert er den Code an den Stellen, an denen Transaktionen ausgeführt werden sollen, mit Aufrufen an ein Transactional Memory Application Binary Interface (TM-ABI). Dieses TM-ABI wurde von Intel [ATSG12] vorgeschlagen und wird mit kleinen Abweichungen auch so vom GCC benutzt. Der Vorteil an der Verwendung eines ABI und der Compilierung von transaktionalen Programmen gegen diese ABI ist, dass Programme nur einmal kompiliert werden müssen und die Implementierung des TM-Systems beliebig ausgetauscht werden kann.
GCC-TM ist hierbei in der *libitm*-Bibliothek des GCCs zu finden. Parallbrid überschreibt diese Bibliothek, sodass Programme Parallbrid statt GCC-TM benutzen. Eine Integration von Parallbrid in die vorhandene Bibliothek ist nicht möglich, da die Bibliotheksarchitektur auf die Exklusivität der Transaktionstypen festgelegt ist. Die Möglichkeit, das TM-System von GCC-TM in die neue Bibliotheksarchitektur zu integrieren, besteht, wurde aber im Rahmen dieser Arbeit nicht umgesetzt.

4.1 TM-ABI im GCC

Das ABI von Intel schreibt Funktionen vor, die von einer TM-Blibliothek implementiert werden müssen. Die Funktionen können von einem Compiler benutzt werden, um sie in den Programmcode eines transaktionalen Programms zu integrieren. Zusätzlich werden noch eine Reihe von Eigenschaften definiert, die der Compiler diesen Funktionsaufrufen mitgeben muss, damit dass TM-System korrekt funktionieren kann. Auch Rückgabewerte dieser Funktionen werden von der ABI vorgeschrieben. Diese dienen dem aufrufenden Programm dazu, den richtigen Ausführungspfad zu wählen. Die Ausführungspfade werden von einem Compiler in den Programmcode eingebettet.

Von der ABI werden Funktionen definiert, die das Starten, Abbrechen und Committen von Transaktionen ermöglichen. Des Weiteren sind Funktionen für den Lese- und Schreibzugriff und die Überprüfung, ob es sich um eine transaktionale Ausführung handelt, definiert. Es gibt auch eine Funktion, die den Wechsel einer Transaktion in den serielen Modus zu erzwingen. Weiter sind noch Funktionen definiert, die es dem Compiler ermöglichen, Optimierungen vorzunehmen. Hierzu gehören Logging Funktionen.
Im Folgenden werden die wichtigsten dieser Funktionen erklärt. Als Namenskonvention beginnen alle von der ABI definierten Funktionen mit $\_ITM\_$.

instrumentedCode	0x0001
uninstrumentedCode	0x0002
hasNoAbort	0x0008
doesGoIrrevocable	0x0040

Tabelle 4.1: Die wichtigsten Parameter, die *_ITM_beginTransaction* übergeben werden.

runInstrumentedCode	0x01
runUninstrumentedCode	0x02
saveLiveVariables	0x04
restoreLiveVariables	0x08
abortTransaction	0x10

Tabelle 4.2: Die Bitmuster mit Bedeutung, aus denen sich der Rückgabewert von *_ITM_beginTransaction* zusammensetz.

Start einer Transaktion

Für das Starten einer Transaktion gibt es die Funktion:

```
uint32_t _ITM_beginTransaction(uint32_t, ...)
```

Diese wird vom Compiler in den Programmcode für den Start jeder Transaktion integriert. Es rufen also auch innere Transaktionen diese Funktion auf. Für das richtige Verhalten ist das TM-System verantwortlich.
Als Parameter werden Informationen über die Art der Transaktion und den vom Compiler generierten Code übergeben. Zu diesen gehört die Information, ob der Compiler einen instrumentierten Codepfad, uninstrumentierten Codepfad oder beides erzeugt hat. Aber auch die Information, ob es sich um eine Transaktion mit Abort oder eine unwiderrufliche Transaktion handelt, wird als Parameter mit übergeben. Die Tabelle 4.1 gibt eine Aufzählung der Parameter an, die von Parallbrid ausgewertet werden. Es gibt in der ABI noch 11 weitere, die für Optimierungen genutzt werden können.

Der Rückgabewert dieser Funktion gibt dem aufrufenden Programm die Informationen, wie es weiter verfahren soll. Tabelle 4.2 zeigt die Bitmuster, aus denen dieser zusammengesetzt ist. Dieser Rückgabewert enthält die Information, welcher Codepfad gewählt werden soll und ob Register gesichert werden müssen.
Es gibt aber auch Rückgabewerte die einen Abort und das Wiederherstellen von Registern beauftragen. Diese werden zurück gegeben, wenn eine Transaktion abbricht und zu dem Beginn der Transaktion zurückspringt. Der aufrufende Code darf dann nicht einen Codepfad zum Ausführen der Transaktion wählen, sondern muss diese überspringen.

Abbruch einer Transaktion

Für den Abbruch einer Transaktion, also einen Userabort, gibt es die folgende Funktion:

```
void _ITM_abortTransaction(_ITM_abortReason)
```

Diese Funktion wird von einem transaktionalen Programm aufgerufen und sagt dem TM-System, dass die aktuelle Transaktion abbrechen soll. Der Parameter, der mit übergeben wird, weist darauf hin, ob es sich um einen Abort der aktuellen Transaktion oder auch aller äußeren Transaktionen handelt. Aus dieser Funktion kehrt die Ausführung nie zur aufrufenden Funktion zurück. Das liegt daran, dass vom TM-System erwartet wird, dass es einen *longjmp* zu dem entsprechenden Start der Transaktion macht.

Commit einer Transaktion

Für den Commit einer Transaktion, unabhängig davon, ob es eine äußere oder innere ist, gibt es die Funktion:

```
void _ITM_commitTransaction (void)
```

Diese Funktion weißt das TM-System an, einen Commit auszuführen. Ist dieser erfolgreich, dann setzt die Ausführung in der nächsten Zeile des aufrufenden Programms fort. Es kann bei spekulativen Transaktionen aber auch der Fall sein, dass der Commit nicht erfolgreich ist und die Ausführung beim Start der Transaktion wieder fortgesetzt wird.

Lesezugriff

Für die instrumentieren Lesezugriffe stellt die ABI mehrere Funktionen bereit. Diese sind für die primitiven Datentypen: Ein, Zwei, Vier und Acht-Byte lange *uint*, *float*, *double*, *long double*, *float _ Complex*, *double _ Complex* und *long double _ Complex*. Hieraus ergeben sich elf verschiedene Lesefunktionen. Die ABI bietet aber für jeden dieser Datentypen vier unterschiedliche Varianten an. Diese sind für den ersten Lesezugriff (R), einen Lesezugriff mit vorherigem Lesezugriff (RaR), einen Lesezugriff mit vorherigen Schreibzugriff (RaW) und einen Lesezugriff mit später folgendem Schreibzugriff (RfW). Es ergeben sich hieraus also 44 verschiedene Lesefunktionen, die der Compiler für die Instrumentierung verwenden kann. Wenn die Computerarchitektur auch MMX, SSE und AVX unterstützt, kommen für diese jeweils wieder 4 hinzu. Hier ein Beispiel der vier Funktionen für den Ein-Byte langen *uint*:

```
uint8_t _ITM_RU1 (const uint8_t *);
uint8_t _ITM_RaRU1 (const uint8_t *);
uint8_t _ITM_RaWU1 (const uint8_t *);
uint8_t _ITM_RfWU1 (const uint8_t *);
```

Schreibzugriff

Auch für die Schreibzugriffe stellt die ABI mehrere Funktionen zur Verfügung. Diese operieren auf den gleichen primitiven Datentypen wie die Lesezugriffe. Es werden für

jeden dieser Datentypen drei Funktionen bereitgestellt. Eine für den ersten Schreibzugriff (W), eine für den Schreibzugriff mit einem vorherigen Lesezugriff (WaR) und eine für den Schreibzugriff mit einem vorangegangenen Schreibzugriff (WaW). Es werden auch, wenn verfügbar, Funktionen für die Vektoroperationen bereitgestellt. Im Folgenden, als Beispiel, die drei Funktionen für den Ein-Byte langen *uint*:

```
void _ITM_WU1 (uint8_t *, uint8_t);
void _ITM_WaRU1 (uint8_t *, uint8_t);
void _ITM_WaWU1 (uint8_t *, uint8_t);
```

Mem-Operationen

Damit man in Transaktionen auch Mem-Operationen wie *memset*, *memmove* und *memcpy* benutzen kann, stellt die ABI auch für diese Funktionen bereit. Von diesen gibt es für *memset* drei und für die anderen beiden jeweils 15 Varianten. Diese unterscheiden sich durch den, von den Lese- und Schreibzugriffen bekannten, Varianten mit vorherigem Lese- und Schreibzugriff aber auch dadurch, dass sie nur eins der beiden oder beide Argumente transaktional behandeln. Als Beispiel sind hier die drei *memset* Funktionen gezeigt:

```
void _ITM_memsetW (void * target, int src, size_t count);
void _ITM_memsetWaR (void * target, int src, size_t count);
void _ITM_memsetWaW (void * target, int src, size_t count);
```

4.2 Bibliotheksstruktur

Die Implementierung der Bibliothek ist in C++ mit C-Kompatibilität realisiert. Sie ist dabei für die x86-Architektur entwickelt worden. Die GCC-Compilerimplementierung unterstützt auch andere Architekturen. Auf diesen wurde Parallbrid nicht getestet und diese Arbeit macht keine Aussage über die Performanz auf den Systemen.
Die Bibliothek besteht aus 3 großen Hauptklassen. Die Erste ist die *method_group*-Klasse. Sie ist bis auf zwei statische Member *pure virtual* und muss von jedem TM-System implementiert werden. An eine Instanz der *method_group*-Klasse gehen die wichtigsten Aufrufe, die die ABI-Schnittstelle zur Verfügung stellt. Diese umfassen den Start, den Commit, den Abort und die Abfrage von Statusinformationen von Transaktionen. Die einzigen Aufrufe, die nicht an die *method group* gehen, sind die Speicherzugriffsoperationen. Diese ABI-Aufrufe gehen an die zweite wichtige Klasse, *abi_dispatch*. Diese Klasse ist auch *pure virtual* und muss von jedem Transaktionstyp eines TM-Systems implementiert werden, der einen instrumentierten Programmpfad ausführen soll. Die Methoden, die von erbenden Klassen von *abi_dispatch* implementiert werden können, umfassen auch den Start und Commit von Transaktionen. Die dritte Klasse ist *gtm_thread*. Eine Instanz dieser Klasse kann für jeden Thread, der Transaktionen ausführt, genutzt werden, um transaktionsspezifische Informationen zu speichern. Ein TM-System muss keine Ableitung dieser Klasse implementieren, aber für die transaktionsspezifischen Informationen Unterklassen bereitstellen. Diese erben von *gtm_transaction_data* und können

Read- und Writeset oder andere Buchführungsinformationen enthalten.
In den Abbildungen 4.1 und 4.2 ist schematisch dargestellt, wie eine Programmausführung, beim Beginn einer Transaktion (Abbildung 4.1) und bei einem Lezezugriff (Abbildung 4.2), die Objekte, der oben genannten Klassen, durchläuft.

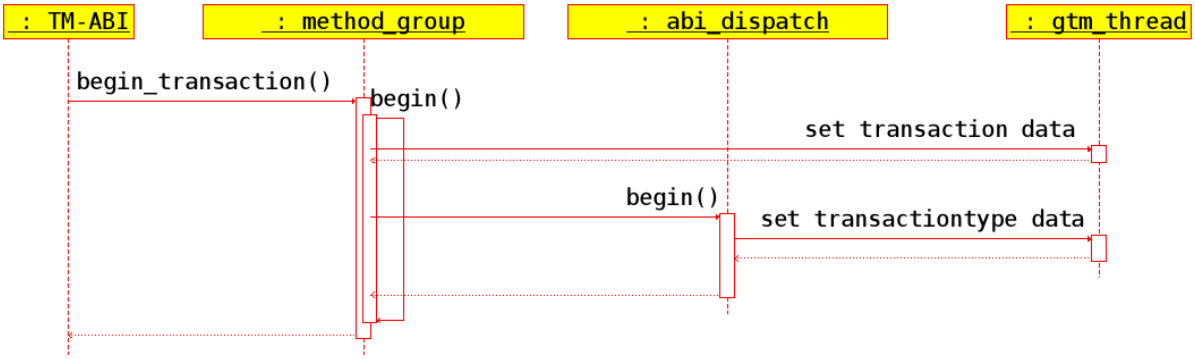


Abbildung 4.1: Ein Beispiel dafür, wie der Aufruf zum Beginn einer Transaktion durch die Objekte gereicht wird.

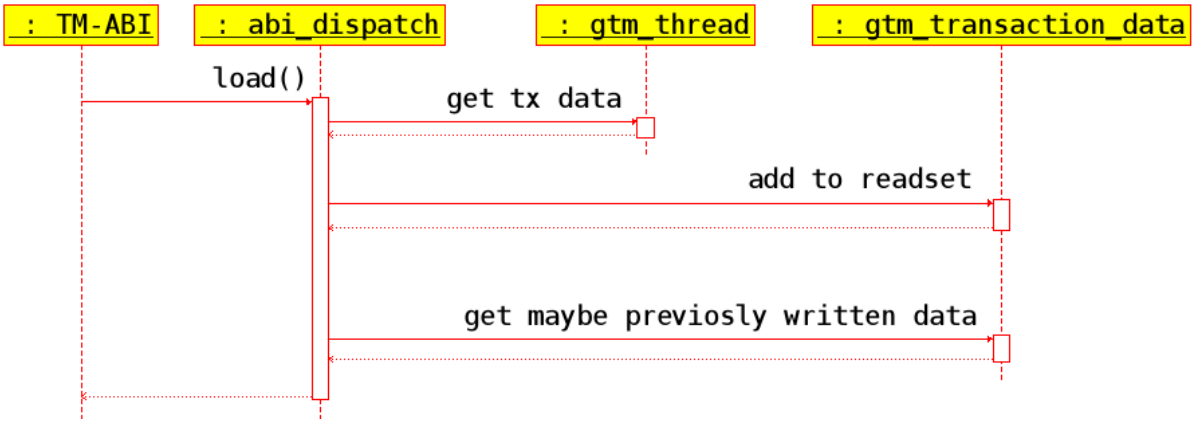


Abbildung 4.2: Ein Beispiel dafür, wie der Aufruf zum Lesen an einer Adresse durch die Objekte gereicht wird.

4.2.1 method_group

Die *method_group*-Klasse, wie auf Abbildung 4.3 zu sehen, besitzt zwei statische Member. Das eine ist eine Pointer Membervariable und das andere ist eine statische Memberfunktion. Der Pointer zeigt immer auf die *method_group*-Instanz des aktuellen TM-Systems. Die Memberfunktion heißt *begin_transaction* und leitet den Transaktionsstart an die entsprechende *method_group* weiter. Sie initialisiert den *method_group*-Pointer, falls dieser auf noch keine *method_group* zeigt. Diese Beginnfunktion, die bei allen Instanzen gleich ist, wird vom hardwarespezifischen Assemblercode aufgerufen. Dieser Assemblercode übergibt die Liste der Transaktionseigenschaften, die der Compiler bereit-

gestellt hat, und einen hardwarespezifischen JumpBuffer.

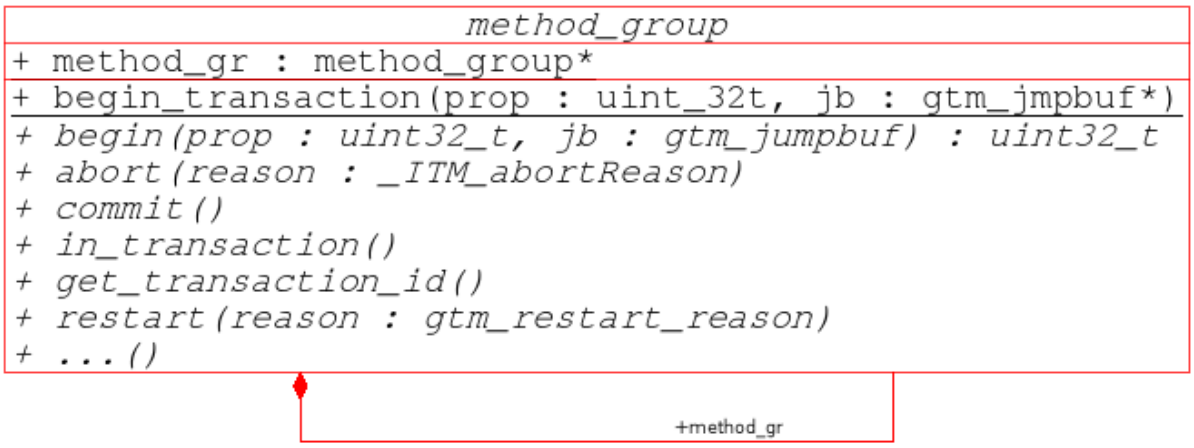


Abbildung 4.3: *method_ group*-Klasse.

Die weiteren Methoden der *method_ group* müssen von dem jeweiligen TM-System implementiert werden. Jedes TM-System erstellt von der *method_ group*-Klasse eine erbende Klasse und nutzt diese als Singleton Object. Die erbende Klasse soll auch die globalen Informationen, die das TM-System benötigt, beinhalten. Wie man in Abbildung 4.3 sieht, gehören zu den Methoden, die dort implementiert werden müssen, der Start, Commit, Abort, Restart und ein paar weitere Methoden. Ein TM-System kann in dieser Startmethode anhand der Informationen über den Systemzustand entscheiden, als welcher Transaktionstyp eine Transaktion ausgeführt wird. Bei den Commit- und Restart-Methoden kann das TM-System entscheiden, wie das weitere Vorgehen einer Transaktion ist und somit an dieser Stelle einen Contention Manager implementieren.

4.2.2 abi_dispatch

Die von der Klasse *abi_ dispatch* erbenden Klassen implementieren jeweils einen Transaktionstyp. Sie sind statische Klassen, von denen pro Transaktionstyp nur ein Objekt erstellt wird. Diese Klassen handhaben das Vorgehen der speziellen Transaktionstypen. Sie speichern die Informationen für die Buchführung in ihrer globalen *method_ group* oder dem zum Thread gehörenden *gtm_ thread*-Objekt. Die Methoden, die von jeder dieser Klassen implementiert werden müssen, sieht man in Abbildung 4.4. Es sind Methoden für den Start, Commit und Rollback von Transaktionen. Da die ABI sehr viele Speicherzugriffsfunktionen definiert, werden diese durch C-Makros erstellt. Diese Makros werten zu *load*- und *store*-Template-Methoden aus, die die Art des Speicherzugriffs (W, RaW, etc.) als Argument erhalten.

Um einen schnellen Zugriff auf das aktuelle Dispatch-Objekt dieser Transaktion zu haben, ist ein Pointer auf dieses im Thread-Local-Storage des aktuellen Threads gespeichert.

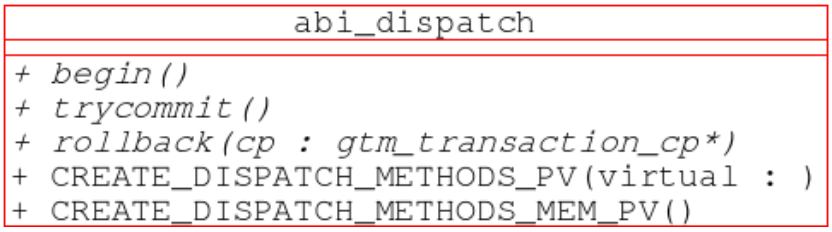


Abbildung 4.4: *abi_ dispatch*-Klasse.

4.2.3 gtm_thread

Die *gtm_ thread*-Klasse ist eine etwas größere Klasse. Sie enthält alle wichtigen Informationen, die ein Thread während seiner Ausführung einer Transaktion braucht. Abbildung 4.5 zeigt einen Ausschnitt davon. Zu diesen Informationen gehören unter anderem eine Kopie des JumpBuffers, um im Fall eines Abbruchs die Register wiederherzustellen, und eine Kopie der vom Compiler generierten Codeeigenschaften, um bei einem Abbruch das weitere Vorgehen zu bestimmen. Die Transaktions-Id und die aktuelle Verschachtelungstiefe sind weitere Informationen, die hier gespeichert werden. Die Threads kennen sich untereinander durch eine verkettete Liste, bei der jeder Thread ein Element mit Zeiger auf das nächste ist. Änderungen an dieser Liste sind durch ein globales Lock, eine statischer Membervariable dieser Klasse, geschützt. Dieses Lock kann von mehreren Threads als Leser akquiriert werden, aber nur von einem Thread als Schreiber. Das Lock ist als Spinlock mit 2 atomic *ints* implementiert. Mehrere Leser können dieses Lock parallel akquirieren, Schreiber werden aber bei der Akquise des Locks bevorteiligt und halten es exklusiv. Für die Buchführungsinformationen gibt es einen Pointer auf ein *gtm_ transaction_ data*-Objekt für Softwaretransaktionen, das durch ein *atomic* geschützt ist. Ein weiterer Pointer existiert für Hardwaretransaktionen. Dieser ist nicht durch ein *atomic* geschützt, da Hardwaretransaktionen diese nicht benutzen können. Weiter hält jedes Thread-Objekt noch Informationen über den Zustand des Threads, mögliche Exceptions, eine Liste der Checkpoints, die Restartanzahl und Loginformationen für die Compiler-Log-Funktion.

Die wichtigste Methode, die die *gtm_ thread*-Klasse bereitstellt und die unabhängig vom TM-System implementiert ist, ist die *rollback*-Methode. Diese versetzt je nach Übergabeparameter das Thread-Objekt in den Zustand des letzten Checkpoints, wenn es einen gab, oder in den Zustand vor der Ausführung von Transaktionen.
Damit das Thread-Objekt überall zur Verfügung steht, ist auch hierfür ein Pointer im Thread-Local-Storage hinterlegt.

4.3 Parallbrid Implementierung

Parallbrid ist durch eine Ableitung der *method_ group*-Klasse, fünf Ableitungen der *abi_ dispatch*-Klasse und zwei Ableitungen von *gtm_ transaction_ data* implementiert.

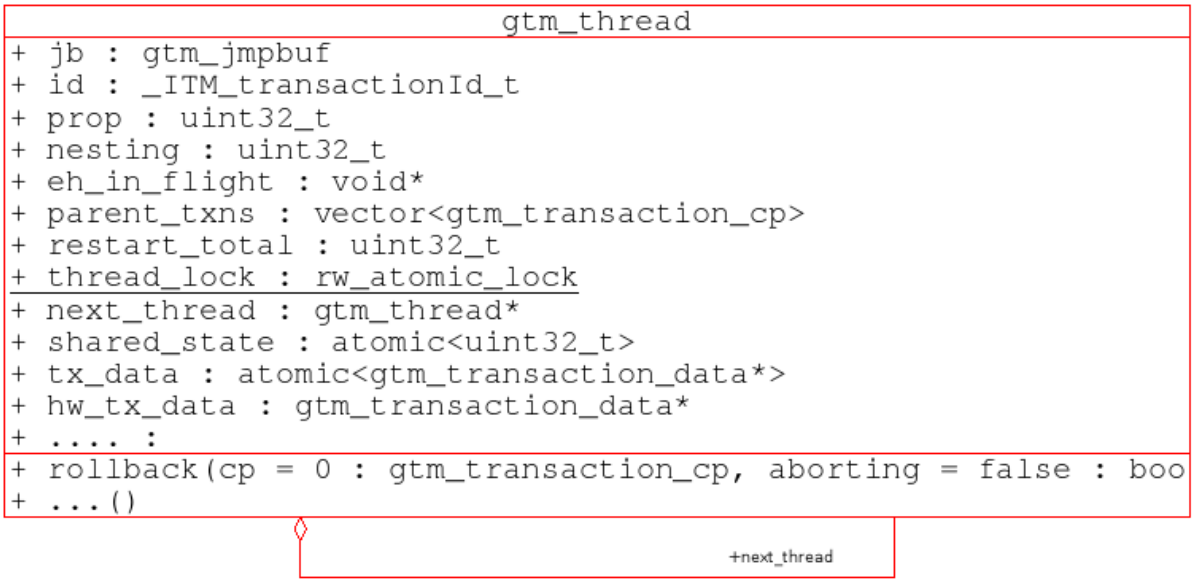


Abbildung 4.5: *gtm_ thread*-Klasse.

4.3.1 parallbrid-mg

Dies ist die Klasse, die die Methoden der *method_ group*-Klasse implementiert und noch eigene Informationen hält, um den Ablauf von Parallbrid zu gewährleisten. Abbildung 4.6 zeigt die zusätzlichen Membervariablen, die von *parallbrid-mg* verwaltet werden. Dazu gehört das Commit-Lock, das ein PThread-Mutex ist und ein Flag, das von jedem Thread gesetzt werden muss, der das Commit-Lock akquiriert. Dieses Flag wird von den Hardwaretransaktionen ausgelesen, da diese nicht auf Synchronisationsmechanismen wie Mutexes zugreifen dürfen.
Der Softwarezähler für die SpecSW-Transaktionen und die Commitsequenz für die SglSW-Transaktionen sind beide als *atomic unsigned integers* realisiert. Durch die Benutzung von *atomics* ist gewährleistet, dass es bei den beiden Zählern keine Lost-Updates gibt, weder durch parallelen Zugriff noch durch Cache *lazyness*. Selbiges gilt auch für den Pointer auf die Transaktion, die das Commit-Lock hält, da dieser auch als *atomic* realisiert ist. Der Hardware-Post-Commit-Zähler ist nicht als *atomic* realisiert, da nur Hardwaretransaktionen auf ihn zugreifen.

invalidate()

Als zusätzliche Methode, zu denen durch *method_ group* geforderten, bietet *parallbrid-mg* eine Methode zur Invalidierung an. Diese Methode wird von den SpeSW-, BFHW-, Inbalbrid- und SerialAboSW-Transaktionen zum Abschluss ihres Commits benutzt. Die Methode holt sich über das zum aufrufenden Thread gehörende Thread-Objekt das Writeset. Danach findet einen Iteration über die Liste aller Threads statt. Diese

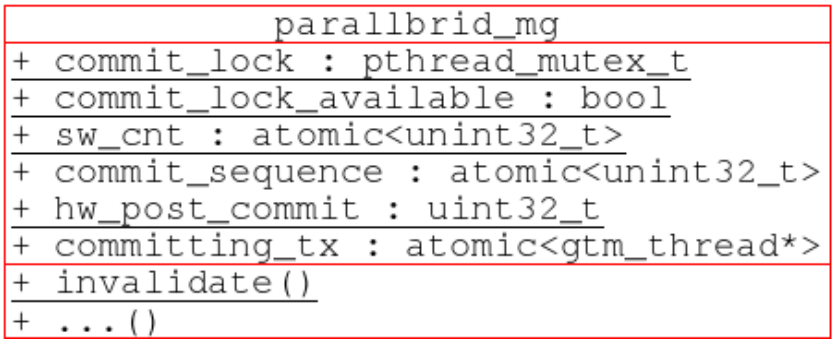


Abbildung 4.6: *parallbrid-mg*-Klasse.

Iteration ist über das Thread-Lock als Leser geschützt, so dass kein Thread während dieses Vorgangs zerstört werden kann. Bei jedem Thread wird der Zustand ausgelesen. Handelt es sich um eine SpecSW-Transaktion, so wird deren Readset mit dem Writeset der invalidierenden Transaktion verglichen. Gibt es einen Schnitt, dann wird die SpecSW-Transaktion invalidiert.

begin()

Die Beginn-Methode der *method_group* ist die Methode, die entscheidet, als welcher Transaktionstyp eine Transaktion gestartet werden soll. Hierbei werden die Informationen, die vom Compiler übergeben wurden, und die Informationen aus dem *method_group*- und *gtm_thread*-Objekt ausgewertet. Außer LiteHW-Transaktionen benötigen alle Transaktionstypen ein *gtm_thread*-Objekt. Dieses wird pro Thread nur einmal erstellt und dann wiederverwendet. Wurde noch kein *gtm_thread*-Objekt erstellt, dann wird dies im Ablauf der Beginn-Methode erledigt.

Die Entscheidungsfindung, welcher Transaktionstyp gestartet werden soll, sieht folgendermaßen aus. Zuerst wird überprüft, ob es sich um eine transaktionale Hardwareausführung handelt. Ist dies der Fall, dann ist dies eine verschachtelte Hardwaretransaktion, die mit einem Aufruf an die HTM-Beginn-Funktion flachgeklopft wird. In diesem Fall ist *begin()* fertig mit der Ausführung. Handelt es sich aber nicht um eine verschachtelte Hardwaretransaktion, dann wird überprüft, ob es sich um eine verschachtelte Softwaretransaktion handelt. Ist dies der Fall, dann wird versucht, sie weiter mit dem gleichen Transaktionstyp auszuführen. Vorher werden aber noch die Eigenschaften über die verschachtelte Transaktion überprüft. Verlangen diese einen Checkpoint, dann muss ein Checkpoint erstellt werden, ansonsten kann mit *flattening* gearbeitet werden. Des Weiteren wird auch überprüft, ob der Compiler auch einen Codepfad anbietet, der mit dem aktuellen Transaktionstyp ausführbar ist. Wenn nicht, dann muss die Transaktion neustarten. Gibt es keine Probleme, dann wird die Verschachtelungstiefe erhöht und die Transaktion setzt fort.
Wenn es sich um eine neue Transaktion bei dem Aufruf von *begin()* handelt, dann werden als erstes die Eigenschaften der Transaktion ausgewertet. Handelt es sich um ei-

ne Transaktion ohne Userabort, die keine unwiderruflichen Operationen ausführt, dann kann die Transaktion als Hardwaretransaktion ausgeführt werden. Ist der Softwarezähler von Parallbrid gleich Null, dann wird als Transaktionstyp LiteHW und ansonsten BFHW verwendet. Beide Transaktiontypen lesen nach dem Start der Hardwaretransaktion das Commit-Lock-Flag, um dieses zu abonieren. Die LiteHW-Transaktion liest zusätzlich noch den Softwarezähler, für den selben Zweck. Da fehlgeschlagene Hardwaretransaktionen nach ihrem Abbruch beim Aufruf ihres Starts fortsetzen, ist der Start der Hardwaretransaktionen in einer Schleife verpackt und wird 20 Mal ausgeführt.
Schlägt der Versuch der Hardwareausführung zu häufig fehl oder war nicht möglich, dann wird bestimmt, als welcher Softwaretransaktionstyp die Transaktion ausgeführt werden soll. Hierzu werden wieder die übergebenen Eigenschaften und die Variablen der *method_group* ausgwertet. Hat die Transaktion einen Userabort, dann kann sie nur als SpecSW oder SerialAboSW-Transaktion ausgeführt werden. Zweitere wird hier beim Start nur ausgewählt, wenn keine anderen Softwaretransaktionen laufen und der Pointer auf die Transaktion, die das Commit-Lock hält, auf Null zeigt. Handelt es sich um eine Transaktion ohne Userabort, dann wird auf Unwiderrufbarkeit geprüft. Führt die Transaktion unwiderrufliche Transaktionen aus, dann wird sie entweder als IrrevocSW oder als SglSW ausgeführt. Ist der Softwarezähler gleich Null und der Compiler hat einen instrumentierten Codepfad erzeugt, dann wird IrrevocSW benutzt und sonst SglSW. Führt die Transaktion keine unwiderruflichen Aktionen aus, dann wird sie als SpecSW-Transaktion ausgeführt, wenn andere Softwaretransaktionen laufen und es einen instrumentierten Codepfad gibt. Ansonsten wird sie als SglSW-Transaktion ausgeführt.

Nachdem ausgewählt wurde, als welcher Transaktionstyp die Transaktion ausgeführt wird, muss bei den fünf Typen, die das *gtm_thread*-Objekt benutzen, dieses initialisiert werden. Dazu gehört das Speichern der Aufruf-Parameter und das Setzen der Transaktions-Id und der Verschachtelungstiefe. Als letztes wird die Beginn-Funktion des jeweiligen Transaktionstyps aufgerufen, damit dieser seine Initialisierung machen kann.

commit()

Die Commit-Methode ruft im Falle einer LiteHW-Transaktion die HTM-Commit-Methode auf und ansonsten die Commit-Methode des aktuellen Transaktionstyps. Eine LiteHW-Transaktion ist mit diesem Aufruf abgeschlossen oder startet wieder in der Beginn-Methode. Bei allen anderen Transaktionstypen regelt deren spezifische Commit-Methode den Ablauf des Commits und gibt zurück, ob dieser erfolgreich war. Ist er dies gewesen, dann kann die Commit-Methode der *method_group* die Transaktion beenden. Ansonsten muss sie, da der Commit gescheitert ist, die Restart-Methode aufrufen.

abort()

Die Abort-Funktion rollt das *gtm_thread*-Objekt dieser Transaktion auf den letzten Checkpoint oder Start zurück. Danach führt sie einen LongJump mit dem JumpBuffer aus.

restart()

Die Restart-Methode wird aufgerufen, wenn eine Softwaretransaktion scheitert. Dies ist nur bei SpecSW-Transaktionen der Fall. Die Restart-Methode erhöht bei ihrem Aufruf den transaktionseigenen Restartzähler und setzt das Threadobjekt in seinen Ausgangzustand zurück. Ist der Grenzwert von 5 Softwareausführungen nocht nicht überschritten, dann wird die Transaktion erneut als SpecSW-Transaktion ausgeführt. Ansonsten wird sie in einen seriellen Zustand eskaliert. Bei einer Transaktion mit Userabort wird diese als eine SerialAboSW-Transaktion erneut ausgeführt. Bei allen anderen Transaktionen erfolgt die erneute Ausführung als IrrevocSW, wenn der Softwarezähler ungleich Null ist, und ansonsten als SglSW.

4.3.2 parallbrid_tx_data

Die *parallbrid_tx_data*-Klasse beinhaltet die Informationen und Daten, die für die Transaktionstypen von Parallbrid wichtig sind. Sie ist von der *gtm_transaction_data*-Klasse abgeleitet. Von dieser Klasse wird pro Thread nur ein Objekt für die transaktionale Ausführung erstellt. Dieses wird von nachfolgenden Transaktionen des gleichen Threads wiederverwendet. Die erste Transaktion eines Threads, die dieses Objekt für die Datenhaltung benötigt, erstellt dieses. Das *gtm_thread*-Objekt eines Threads zeigt also bei jeder Transaktion auf das gleiche Datenhaltungsobjekt. Kopien des *parallbrid_tx_data*-Objekts werden nur beim Checkpointing erstellt. Diese Kopien werden beim Laden eines Checkpoints nur benutzt, um die Daten aus ihnen zu beziehen.

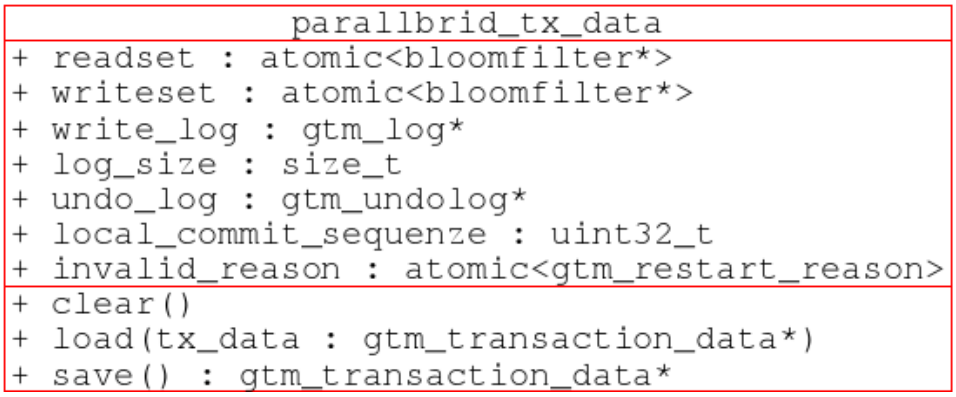


Abbildung 4.7: *parallbrid_tx_data*-Klasse.

Die *parallbrid_tx_data*-Klasse ist in Abbildung 4.7 zu sehen. Nicht alle Informationen, die in dieser Klasse gesammelt werden können, sind für alle Transaktionstypen interessant. Die meisten werden für die SpecSW-Transaktionen benötigt. Zu diesen gehören die lokale Commit-Sequenze, das Invalidierungsflag, was als ein *atomic* Enumerator umgesetzt ist, der Writelog mit Größenzähler und das Readset. Das Writeset wird von allen Transaktionstypen benötigt, die den Invalidierungsprozess betreiben. Der Undolog ist

nur für die SerialAboSW-Transaktionen von Bedeutung, da diese als einzige Änderungen am Speicher rückgängig machen müssen.
Die Verweise auf die Bloomfilter des Read- und Writesets und das Invalidierungsflag sind als *atomics* umgesetzt, um die Aktualität dieser Informationen bei der Intertransaktionskommunikation zu gewährleisten.

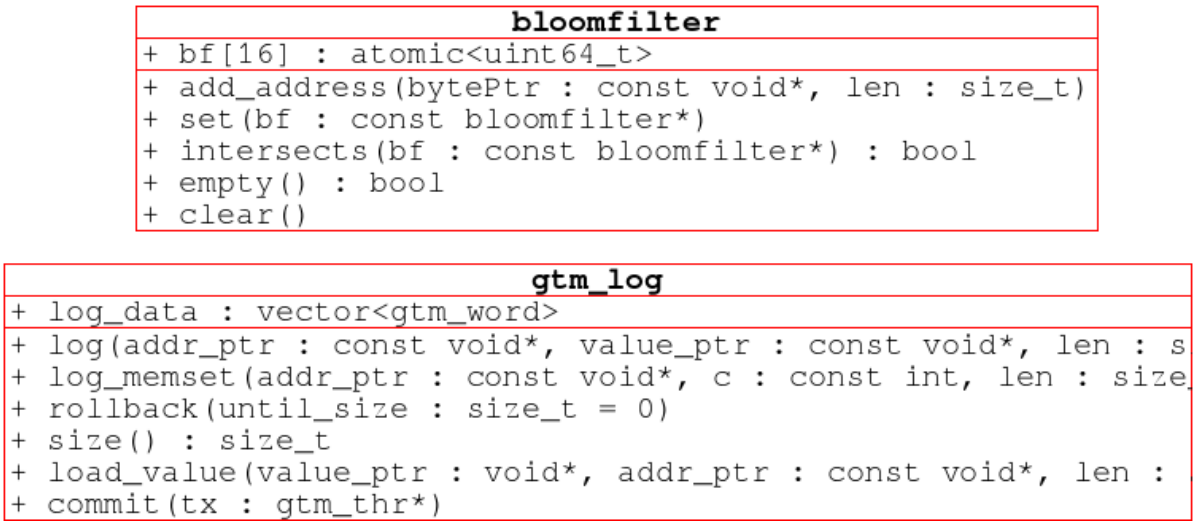


Abbildung 4.8: *bloomfilter*- und *gtm_log*-Klasse.

Die Bloomfilter, die für die Read- und Writesets benutzt werdenen, sind in Abbildung 4.8 zusehen. Sie benutzen für die Speicherung der Bits ein Array aus *atomic*, acht Byte langen, Integern. Auch hier sorgen die *atomics* wieder dafür, dass andere Transaktionen immer einen aktuellen Zustand der Bloomfilter lesen.
Da das gesamte Array nicht aus einem *atomic* bestehen kann, ist es möglich, den Bloomfilter einer anderen Transaktion auszulesen, in dem noch nicht alle Adressen einer Operation eingetragen sind. Dies stellt aber kein Problem dar, da alle Transaktionen zu erst den Eintrag im Set machen, bevor sie auf den Speicher zugreifen.

Der Writelog, in dem die SpecSW-Transaktionen ihre Daten bis zum Commit zwischenspeichern, ist auch in Abbildung 4.8 dargestellt. Hier werden die Daten in einem Vektor aus Wörtern gespeichert. Der Log bietet die Möglichkeit, Werte und ihren zukünftigen Zielort zu loggen, den Log zu einem bestimmten Punkt zurückzurollen und die Daten aus dem Log in den Speicher zu committen. Des Weiteren bietet der Log noch die Funktion *load_value()* an. Diese dient dazu, Daten, die zuvor geschrieben wurden, bei einer Leseoperation aus dem Log zu bekommen. Hierbei können auch nur einzelne Bytes durch die Einträge aus dem Log ersetzt werden. Dies ist zum Beispiel der Fall, wenn zwei Bytes zuvor geschrieben wurden, die Teil einer vier Byte Leseoperation sind.
Beim Commit des Logs werden alle Einträge, mit den ältesten angefangen, an ihre jeweiligen Zieladressen geschrieben. Hierdurch landen die richtigen Einträge im Speicher, falls eine Transaktion mehrfach an die gleichen Adressen schreibt.

Der hier nicht gezeigte Undolog für die SerialAboSW-Transaktionen ist ähnlich wie der Writelog aufgebaut. Bei seinem Rollback werden die Daten des Logs in umgekehrter Reihenfolge zum Commit des Writelogs in den Speicher geschrieben. Es beginnt also mit dem neusten Eintrag und endet mit dem ältesten.

4.3.3 abi_dispatch

Für jeden Transaktionstyp, außer LiteHW, existiert bei Parallbrid eine abgeleitete Klasse von *abi_dispatch*. In diesen Klassen sind die für den Transaktionstyp spezifischen Operationen für die Ausführung von Transaktionen implementiert. Es handelt sich bei diesen Klassen um statische Klassen, die keine Informationen halten. Für jeden Transaktionstyp gibt es daher nur ein *abi_dispatch*-Objekt, das von allen Transaktionen dieses Typs genutzt wird.
LiteHW-Transaktionen haben kein *abi_dispatch*-Objekt. Sie sind als leichtgewichtige Transaktionen gedacht und benötigen keine extra Instrumentierung bei der Ausführung von Transaktionen. Der Start und Commit von LiteHW-Transaktionen wird daher von den *begin()*- und *commit()*-Methoden der *method_group*-Klasse erledigt. LiteHW-Transaktionen können nur uninstrumentierte Codepfade ausführen, da die instrumentierten bei Speicheroperationen ein *abi_dispatch*-Objekt benötigen.

Die Klassen für die Transaktionstypen sind so implementiert, wie sie in Kapitel 3.3 beschrieben sind. Es werden daher hier nur ein paar Besonderheiten erwähnt.

SpecSW

Bei der Implementierung der SpecSW-Transaktionen ist als Besonderheit zu erwähnen, dass diese an einer Stelle Hardwaretransaktionen nutzen. Dies ist bei der Validierung der Fall, wenn das System Hardwaretransaktionen unterstützt. BFHW-Transaktionen informieren mit dem Hardware-Post-Commit-Zähler andere Transaktionen darüber, dass sie noch den Invalidierungschritt vollziehen müssen. Solange dieser Zähler also ungleich Null ist, müssen SpecSW-Transaktionen während ihrer Validierung warten. Dieser Zähler wird von den BFHW-Transaktionen während ihrer transaktionalen Ausführung hochgezählt. Wenn die SpecSW-Transaktionen diesen Zähler einfach so auslesen, dann ist nicht garantiert, dass sie eine aktuelle Version dieses Zählers lesen. Es kann sich um einen gecacheten Wert handeln, der zu einem fehlerhaften Verhalten des TM-Systems führt. Den Zähler zu einem *atomic* zu machen, hilft auch nicht weiter, da er dann nicht mehr von den Hardwaretransaktionen hoch gezählt werden kann.
Wenn die Architektur daher Hardwaretransaktionen unterstützt und somit möglicherweise BFHW-Transaktionen ausgeführt werden, dann lesen SpecSW-Transaktionen während ihrer Validierung diesen Zähler mit einer kleinen Hardwaretransaktion aus. Hiermit ist gewährleistet, dass die SpecSW-Transaktionen immer den aktuellen Zählerstand lesen und korrekt ausgeführt werden können.

SglSW

SglSW-Transaktionen werden normalerweise auf dem uninstrumentierten Codepfad ausgeführt. Sie können aber auch für einen instrumentierten Ausführungspfad gewählt werden. Der Grund dafür ist, dass für sie eine *abi_dispatch*-Klasse implementiert ist, die auch Speicherzugriffsoperationen beinhaltet. Die Speicherzugriffe werden von der Implementierung aber nur durchgereicht und nicht weiter instrumentiert.

BFHW

Für die BFHW-Transaktionen sind zwei Besonderheiten zu erwähnen. Zum Einen nutzen sie für die Buchführung des Writesets nicht das gleiche *parallbrid_tx_data*-Objekt wie die anderen Transaktionen. Und zum Anderen benutzen sie nach ihrer eigentlichen transaktionalen Ausführung nochmal eine kleine Hardwaretransaktion.
Die Verwendung des eigenen Buchführungsobjekts für diesen Transaktionstyp hat seinen Grund bei den *atomics*. Hardwaretransaktionen können auf diese nicht zugreifen. Für die BFHW-Transaktionen führt jedes *gtm_thread*-Objekt daher einen Pointer auf ein *gtm_transaction_data*-Objekt, der nicht durch *atomics* geschützt ist. Die Ableitung des *gtm_transaction_data*-Objekts für Hardwaretransaktionen heißt *parallbrid_hw_tx_data*. In diesem ist nur ein Writeset gespeichert, welches eine Bloomfilter-Implementierung ohne *atomics* verwendet. Dieses Buchführungsobjekt wird von der ersten BFHW-Transaktion eines Threads erstellt und von den nachfolgenden weiter verwendet. Das aktuelle Writeset einer BFHW-Transaktion ist von den anderen Transaktionen aber nicht einsehbar, da es sich bis zum Commit dieser im Writeset der Hardwaretransaktion befindet.
Die zusätzliche Hardwaretransaktion kommt in der Post-Commit-Phase einer BFHW-Transaktion vor. Nachdem die Hardwaretransaktion der regulären transaktionalen Ausführung ihren Commit gemacht hat, betreiben BFHW-Transaktionen eine Invalidierung von SpecSW-Transaktionen. Wenn diese abgeschlossen ist, muss der Hardware-Post-Commit-Zähler wieder dekrementiert werden, damit SpecSW-Transaktionen ihre Ausführung fortsetzen können. Bei dieser Dekrementierung besteht das gleiche Problem, wie beim Auslesen dieses Zählers durch die SpecSW-Transaktionen. Um das Problem zu umgehen, wird eine kleine Hardwaretransaktion verwendet, die diese Dekrementierung des Zählers korrekt ausführt.

5 Verwendung

Damit ein Softwaresystem Parallbrid als TM-System benutzt, muss diese Software die vom GCC unterstützen Sprachkonstrukte für Transactional Memory verwenden und mit diesem kompeliert werden. Zusätzlich muss noch die Variante der geteilten *libitm*-Bibliothek vorhanden sein, die Parallbrid enthält.

5.1 Verwendung der Sprachkonstrukte

Da Parallbrid die GCC-Compilerintegration für TM nutzt, müssen Transaktion in C++ mit den vom GCC vorgegebenen Sprachkonstrukten verwendet werden. Diese Sprachkonstrukte sind für die allgemeine Verwendung von Transactional Memory gedacht und in der GCC-Dokumentation [GW12] kurz erklärt. Für eine ausführlichere Dokumentation, wie und wo Transaktionen verwendet werden können, ist das Draftpaper [ATSG12] für die Sprachspezifikation zu nennen. In dieser werden alle Verwendungsmöglichkeiten von Transaktionen, als Statements, Expressions und Funktionsblöcke erklärt. Da die Verwendung als Transaktions-Statement die häufigste und intuitivste ist, wird diese im Folgenden erklärt.

Wenn ein Programmierer einen Codeblock identifiziert hat, der als Transaktion ausgeführt werden soll, dann muss er diesen als solche kennzeichnen. Dies geschieht, indem man den Codeblock in geschweifte Klammern einklammert und dem ein Schlüsselwort voranstellt, das den Typ der Transaktion angibt. Die beiden Typen, die zur Auswahl stehen, sind Transaktionen, die keine unwiderruflichen Aktionen enthalten dürfen und solche, die unwiderrufliche Aktionen tätigen dürfen. Enthält eine Transaktion keine unwiderruflichen Aktionen, dann kann man sie mit dem Schlüsselwort *__transaction_atomic* kennzeichnen. Transaktionen, die hingegen auch solche Aktionen enthalten können, müssen mit dem Schlüsselwort *__transaction_relaxed* ausgezeichnet werden.

Transaktionen müssen natürlich auch andere Funktionen aufrufen können. Hierfür gibt es die Funktionsattribute *transaction_pure* und *transaction_unsafe*. Mit diesen kann man Funktionen attribuieren und so bestimmen, von welchem Transaktionstyp diese aufgerufen werden darf. Ist eine Funktion nicht attribuiert und wird in einer Transaktion verwendet, dann inferiert der Compiler den Typ dieser Funktion.
Das Attribut *transaction_pure* kann dafür verwendet werden, um eine Funktion zu kennzeichnen, die von Atomic-Transaktionen benutzt werden darf. Dieses Attribut schließt die Verwendung dieser Funktion in einer Relaxed-Transaktion nicht aus. Es sagt dem Compiler nur, dass diese Funktion von Atomic-Transaktionen ausgeführt werden darf.

Mit dem Attribut *transaction_unsafe* verhält es sich genau umgekehrt. Eine Funktion, die mit diesem Attribut versehen ist, darf nur von Relaxed-Transaktionen ausgeführt werden. Atomic-Transaktionen dürfen diese Funktion nicht aufrufen und es kommt zu einem Compilerfehler, wenn man dieses versucht.
Die Attribute geben einen Ansatz von Typsicherheit beim Schreiben von transaktional genutzten Funktionen. Sie sind aber auch für die Verwendung von externen Legacy-Bibliotheken nützlich. Diese können vom Compiler nicht instrumentiert werden und dürfen daher nur von Relaxed-Transaktionen verwendet werden.

Eine besondere Stärke des Transactional Memory Modells ist, dass Transaktionen ohne große Bedenken verschachtelt benutzt werden können. Hierdurch können sich große zusammengesetzte Transaktionen aus vielen kleinen ergeben. Es gibt bei dieser Verschachtelung aber Einschränkungen aufgrund des Transaktionstyps. Atomic-Transaktionen können hierbei als *transaction_pure* und Relaxed-Transaktionen als *transaction_unsafe* angesehen werden. Relaxed-Transaktionen können also andere Relaxed- und Atomictransaktionen aufrufen. Atomic-Transaktionen dürfen nur andere Atomic-Transaktionen als innere Transaktionen ausführen.

```
1  int global_cnt = 0;
2
3  foo() __attribute__((transaction_pure)) {
4    global_cnt += 10;
5  }
6
7  bar() __attribute__((transaction_unsafe)) {
8    cout << global_cnt;
9  }
10
11 do() {
12   __transaction_atomic {
13     global_cnt = 9;
14     foo();              // OK, foo() is transaction_pure.
15   }
16   __transaction_relaxed {
17     foo();              // OK, relaxed can call everything.
18     bar();              //  - " -
19   }
20   __transaction_atomic {
21     foo();              // OK, foo() is transaction_pure.
22     bar();              // Compiler ERROR. bar() is unsafe.
23   }
24 }
```

Listing 5.1: Ein Beispiel dafür, wie Transaktionen und die Attribute verwendet werden.

In dem Codesegement 5.1 sieht man zwei Funktionen, die jeweils attribuiert wurden. Die Funktion *foo()* kann von allen Transaktionen aufgerufen werden und die Funktion *bar()* nur von Relaxed-Transaktionen. In der *do()*-Methode werden drei Transaktionen nacheinander ausgeführt. Die ersten beiden Transaktionen sind hierbei korrekt. Die dritte Transaktion ist eine Atomic-Transaktion, die versucht, eine Funktion aufzurufen, die als *transaction_unsafe* gekennzeichnet ist. Dies wird beim Compilierversuch zu einem Compilerfehler führen.

Die Möglichkeit zum Userabort ist durch das Schlüsselwort *__transaction_cancel* gegeben. Immer wenn dieses in dem Ausführungspfad einer Transaktion vorkommt, wird die umschließende Transaktion zurückgerollt und übersprungen. Mit dem Attribut *outer* kann man der Userabort-Anweisung auch sagen, dass sie nicht nur die direkt umschließende, sondern auch die äußeren Transaktionen abbrechen soll. Die Anweisung *transaction_cancel* darf daher nur in Atomic-Transaktionen benutzt werden. Diese dürfen dann nicht von Relaxed-Transaktionen aufgerufen werden. Um zu kennzeichnen, dass eine Funktion eine Transaktion mit Abort enthält, gibt es das Funktionsattribut *transaction_may_cancel_outer*.

```
1  int a, b = 0;
2
3  foo(x) __attribute__((transaction_may_cancel_outer)) {
4    __transaction_atomic {
5      if (x > 10) __transaction_cancel;
6      if (x < 0) __transaction_cancel [[outer]];
7      a = x;
8      b = x;
9    }
10 }
11
12 bar() {
13   __transaction_atomic {
14     a = 1;
15     __transaction_atomic {
16       b = 1;
17       __transaction_cancel;
18     }
19 }
20
21 do() {
22   __transaction_atomic {
23     foo(5);
24     foo(-1);
25     }
26 }
```

Listing 5.2: Ein Beispiel dafür, wie ein Userabort in Transaktionen benutzt werden kann.

In dem Codesegment 5.2 ist ein Beispiel für die Verwendung des Useraborts gegeben. Die Funktion *foo()* führt eine Transaktion aus, die a und b auf den Wert x setzt, falls $0 \leq x \leq 10$ gilt. Ist x größer als der Wert 10, dann wird die Transaktion, die *foo()* ausführt, abgebrochen. Bei einem negativen Wert von x wird die Transaktion auch abgebrochen. Es werden aber auch die umschließenden Transaktionen abgebrochen, falls es solche gibt.
Die Funktion *bar()* verändert mit ihrer Ausführung nur den Wert von a, da die Transaktion, die b verändert, immer abgebrochen wird.
Die Funktion *do()* verändert die Werte von a und b gar nicht, da der zweite Aufruf von *foo()* die Transaktion von *do()* beendet.

5.2 Kompilieren von Software mit TM

Wenn eine Software, die Transactional Memory verwendet, geschrieben wurde, dann muss diese mit einem GCC-Compiler ab Version 4.7 übersetzt werden. Damit ein GCC-Compiler, wie *gcc* oder *g++*, weiß, dass er die Transactional Memory-Unterstützung aktivieren soll, muss ihm das Flag *-fgnu-tm* übergeben werden. Ein Aufruf könnte dann folgendermaßen aussehen:

```
g++ -fgnu-tm philosophers.cc
```

5.3 Kompilieren der Bibliothek

Wenn ein Programm mit einem GCC-Compiler und dem Flag *-fgnu-tm* übersetzt wurde, dann benötigt dieses Programm die geteilte Bibliothek *libitm*. Mit dem GCC wird eine Version dieser Bibliothek ausgeliefert, die das in Kapitel 2.3.3 beschrieben GCC-TM-System beinhaltet. Um Parallbrid zu verwenden, kann man sich die Quellen aus dem Git-Repository [Bal16] herunterladen und die *libitm* Bibliothek selbst übersetzen. Leider ist es beim GCC und seinen mitgelieferten Bibliotheken nicht einfach, nur einzelne Komponenten zu kompilieren. Es wird daher empfohlen, den gesamten GCC-Quellcode für die angestrebte Verwendung zu übersetzen, auch wenn man man nur eine Bibliothek benötigt. Die GCC-Dokumentation [GW] gibt viele Hinweise und Tipps, wie man den GCC-Quellcode übersetzt und welche zusätzlichen Softwarekomponenten man hierzu benötigt.
Das Listing 5.3 stammt aus der GCC-Dokumentation und zeigt einen exemplarischen Prozess zum Konfigurieren, Kompilieren und Installieren des GCC. Die erste und zweite Zeile zeigen das Entpacken der Packetquellen und Betreten des Source-Ordners. Bei Verwendung des Git-Repositorys entfällt der Schritt des Entpackens. In Zeile 3 wird ein Skript ausgeführt, das die für die Übersetzung benötigten Softwarekomponenten aus dem Internet herunter lädt, falls diese nicht auf dem System vorhanden sind. In der vierten Zeile wird ein Objekt-Ordner erstellt. In diesem wird in Zeile 6 die Konfiguration des zu kompelierenden GCCs vorgenommen. Mit den Parametern können die Sprachen ausgewählt werden, für die ein Compiler übersetzt werden soll. In der achten Zeile findet

die Übersetzung und in der neunten die Installation statt.

```
1 tar xzf gcc-4.6.2.tar.gz
2 cd gcc-4.6.2
3 ./contrib/download_prerequisites
4 cd ..
5 mkdir objdir
6 cd objdir
7 $PWD/../gcc-4.6.2/configure --prefix=$HOME/gcc-4.6.2 --enable-↩
    languages=c,c++,fortran,go
8 make
9 make install
```

Listing 5.3: Das Beispiel der GCC-Dokumentation dafür, wie man den GCC konfigurieren, kompilieren und installieren kann.

Möchte man auf diesem Weg nur an die *libitm*-Bibliothek kommen, dann lässt man den letzten Schritt (*make install*) weg und verschiebt die Bibliothek selbst an seinen Zielort.

6 Tests

Für die Bewertung und den Vergleich von Transactional Memory Systemen gibt es keinen offiziellen Standard. Es wird im wissenschaftlichen Umfeld aber häufig die STAMP-Benchmark-Suite [MCKO08] der Stanford Universität für diese Zwecke verwendet. Die Abkürzung STAMP steht hierbei für *Stanford Transactional Applications for Multi-Processing*. Da die Benchmark-Suite 2008 veröffentlicht wurde und erst danach HTMs in kommerziellen Prozessoren verfügbar wurden, haben sich W. Ruan et al. die Frage gestellt, ob STAMP noch zeitgemäß ist. Im Zuge ihres Papers *STAMP Need Not Be Considered Harmful* [RLS14] haben sie die STAMP-Benchmark-Suite auf die C++ Sprachspezifikation angepasst und einige Fehler behoben. Sie kommen zu dem Schluss, dass STAMP als Benchmark für Transactional Memory noch tauglich ist und als Grundlage für zukünftige Benchmarks verwendet werden kann. Da STAMP auf die C++ Sprachspezifikation angepasst wurde und als sinnvoller Benchmark angesehen wird, dient die Benchmark-Suite auch für die Bewertung von Parallbrid.
STAMP besteht aus 8 individuellen Testszenarien, um die verschiedenen Einsatzfelder von Transactional Memory zu bewerten. Einer dieser Benchmarks, namens Bayes, hat ein nicht-deterministisches Verhalten, weshalb W.Ruan et al. empfehlen, ihn nicht mehr zu verwenden. Für die Testläufe mit Parallbrid wurden daher nur die anderen 7 Benchmarks verwendet. Im Folgenden (Kapitel 6.1) wird kurz beschrieben, welche diese sieben Benchmarks sind und was sie tun. Die Beschreibung basiert auf der Beschreibung von W. Ruan et al. Die Ergebnisse von Parallbrid und GCC-TM und eine Erklärung für diese werden im Anschluss daran thematisiert (Kapitel 6.2).

6.1 STAMP-Benchmarks

Genome

Der Genome-Benchmark ist ein Algorithmus zur Gensequenzierung. Er benutzt *string matching*, um eine Gensequenz aus einer Menge von DNA-Fragmenten zu rekonstruieren. Der Algorithmus läuft in zwei Phasen ab, bei denen die zweite viele Transaktionen enthält, die nur Lesezugriffe vollziehen. Die Zugriffsmuster der einzelnen Transaktionen bei diesem Benchmark sind klein, aber dafür nicht trivial, was eine Locking-Strategie, anstelle von Transactional Memory, sehr schwierig macht.

Intruder

Der Intruder-Benchmark ist ein Analysator für Netzwerkpakete. Transactional Memory wird hierbei als Ersatz für eine grob granulierte Locking-Strategie verwendet. Der Analysator überprüft die Netzwerkpakete, ob sie eine verdächtige Signatur enthalten. Die geteilten Speicherbereiche, die benutzt werden, sind eine *queue*, für die eingehenden Pakete, und ein *dictionary (a balanced tree)*, für das Sammeln von aufgeteilten Paketen. Der Benchmark benutzt eine Pipeline, bei der die dritte Phase keine Transaktionen ausführt.

KMeans

Der KMeans-Benchmark ist ein Algorithmus zur iterativen Clusteranalyse. Die Daten für die Cluster sind auf die Threads aufgeteilt, sodass es zu unregelmäßigen, meist kleinen, Transaktionen kommt. Transactional Memory wird in diesem Algorithmus für zwei Dinge benutzt. Erstens, um die iterativen Updates an den Clustern zu schützen, was ohne TM mit Locks passieren muss. Und zweitens, um ein Update an einer Menge von Clustern als atomare Operation auszuführen.

Labyrinth

Der Labyrinth-Benchmark ist eine Implementierung des Lee-TM[AKW+08] *circuit-routing*-Benchmarks. Die Transaktionen versuchen bei diesem Algorithmus in einem dreidimensionalen Array so viele Start- und Endpunkte miteinander zu verbinden, ohne das sich die Verbindungen überlappen. Hierfür arbeiten die Transaktionen auf einer lokalen Kopie des geteilten Arrays. In dieser lokalen Kopie berechnen sie einen Weg und versuchen diesen dann in das geteilte Array einzutragen, wenn der Weg dort noch frei ist.

Ssca2

Ssca2 ist ein Benchmark, bei dem ein gerichteter und gewichteter Multi-Graph konstruiert wird. Der Benchmark bietet mehrere Kernel für diese Aufgabe an, von denen aber immer nur der Erste in wissenschaftlichen Publikationen verwendet wird. Dieser erste Kernel besteht hauptsächlich aus kleinen *read-modify-write*-Operationen, mit weniger als fünf Lese- und weniger als 4 Schreibzugriffen. Die Menge der Zugriffsorte einer Transaktion wird während der Laufzeit dynamisch bestimmt. Transactional Memory wird hier benutzt, um Dead-Locks zu vermeiden, die bei einer fein granularen Locking-Strategie auftreten können.

Vacation

Der Vacation-Benchmark simuliert die Verarbeitung von Onlinetransaktionen. Die Datenbank ist durch mehrere geteilte Bäume simuliert. Jede Transaktion muss bei ihrer Ausführung auf mehrere dieser Bäume zugreifen.

Yada

Bei dem Yada-Benchmark arbeiten die Threads zusammen, um eine Delaunay Verfeinerung, unter Benutzung des Ruppert's-Algorithmus, durchzuführen. Als Eingabe ist ein großes geteiltes Netz gegeben. Die Threads iterieren über die Dreiecke dieses Netzes und suchen nach Winkeln, die einen Grenzwert unterschreiten. Ist so ein kleiner Winkel gefunden, dann wird dem Netz ein weiterer Punkt hinzugefügt und das Umfeld um diesen neu trianguliert. Die Speicherstellen, auf die eine Transaktion zugreifen muss, werden dynamisch zur Laufzeit bestimmt, da sie von dem Eingangsnetz abhängen. Diese dynamische Speicherbestimmung macht den Einsatz einer Locking-Strategie anstelle von TM sehr kompliziert, weshalb TM hierfür gut geeignet ist.

6.2 Ergebnisse

Die Benchmarks zum Vergleich von Parallbrid und GCC-TM wurden auf einem Core i7-6700K mit acht Gigabyte Hauptspeicher ausgeführt. Da dieser Prozessor Intels TSX-Erweiterung unterstützt, benutzen beide TM-Systeme RTM als Hardwaretransaktionen. Der Prozessor hat vier Kerne und unterstützt Hyperthreading, sodass bis zu acht Hardwaretransaktionen parallel ausgeführt werden können.
Bei GCC-TM wurde der Standartbetriebsmodus gewählt, der, aufgrund der Verfügbarkeit von HTM, aus Hardware- und seriellen Softwaretransaktionen besteht. Bei Parallbrid kam eine Konfiguration zum Einsatz, die Hardwaretransaktionen bis zu 20-mal wiederholt, falls diese aufgrund von Konflikten scheitern, und Softwaretransaktionen bis zu 5-mal. Die Zahl der Wiederholversuche ist so hoch gewählt, um viel Parallelität zu ermöglichen und nicht zu schnell in einer seriellen Ausführung zu landen.
Für die Ergebnisse der Benchmarks wurde jeder Benchmark mit jeder Kernanzahl, zwischen eins und acht, insgesamt fünf Mal ausgeführt und der Durchschnitt für die Ausführungszeit verwendet. Die einzelnen Benchmarks wurden mit den Parametern aufgerufen, die von W. Ruan et al. mit dem Quellcode des STAMP-Benchmarks mitgeliefert wurden. Die Aufrufe in Listing 6.1 sind die Aufrufe, mit denen die folgenden Benchmarks gemacht wurden. Hinter jedem Aufruf gehört noch der Parameter *-t* und eine Zahl, die angibt, wie viele Threads und somit wie viele Kerne benutzt werden sollen.

```
genome -s64 -g16384 -n16777216
intruder -a10 -l128 -n262144 -s1
kmeans -m40 -n40 -T0.00001 -i ../../inputs/kmeans/random-n65536-↩
   d32-c16.txt
labyrinth -i ../../inputs/labyrinth/random-x512-y512-z7-n512.txt
ssca2 -s20 -i1.0 -u1.0 -l3 -p3
vacation -n2 -q90 -u98 -r1048576 -T4194304
yada -a 15 -i ../../inputs/yada/ttimeu1000000.2
```

Listing 6.1: Aufrufe der Benchmarks mit Parametern. (Ohne Threadanzahl-Parameter)

Im Folgenden werden die Ergebnisse der Benchmarks von Genome (Kapitel 6.2.1), Intruder (Kapitel 6.2.2), KMeans (Kapitel 6.2.3), Labyrinth (Kapitel 6.2.4), Ssca2 (Kapitel

6.2.5), Vacation (Kapitel 6.2.6) und Yada (Kapitel 6.2.7) ausgewertet.

6.2.1 Genome

Abbildung 6.1 zeigt das Ergebnis des Genome-Benchmarks. Wie man in dieser sieht, verhält sich Parallbrid deutlich langsamer als GCC-TM.

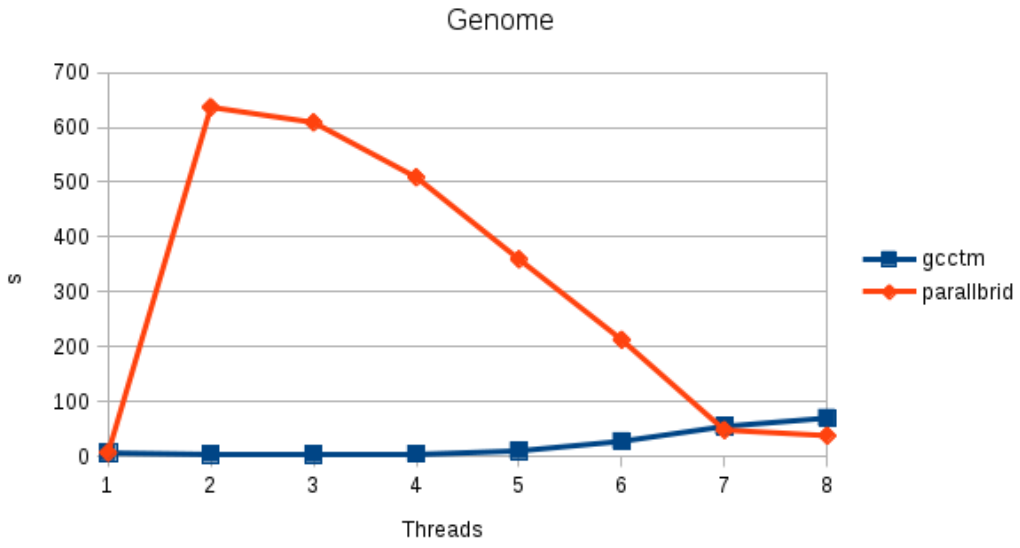


Abbildung 6.1: Ergebnisse des Genome-Benchmarks

Für einen Kern ist der Unterschied der beiden Systeme nicht so groß; GCC-TM brauchte 5,8 s und Parallbrid 6,3 s. Bei mehr als einem Kern ist die Ausführungszeit von Parallbrid aber sehr langsam. Dies liegt an einem hohen Grad an Konflikten zwischen den Transaktionen, die zu einem Livelock zwischen Hardwaretransaktionen führen. Dieser wird erst durch den Wechsel zu einem der Softwaretransaktionestypen gelöst. Denn Softwaretransaktionen werden nur abgebrochen, wenn eine andere Transaktion einen Commit vollzogen hat, womit Fortschritt garantiert ist. Eine Transaktion, die als Hardwaretransaktion aufgrund von Konflikten immer gescheitert ist, kann aber auch als langsamere SpecSW-Transaktion bei all ihren Versuchen scheitern, wenn andere BFHW-Transaktionen sie invalidieren. Diese Transaktion muss dann als unwiderrufliche Transaktion abgeschlossen werden. Eine Transaktion, die in diesen Modus eskaliert wurde, hat viel Rechenzeit bis zu ihrem erfolgreichen Commit gekostet.
Auch GCC-TM wird, ab einer Anzahl von fünf Kernen, langsamer in der Ausführung dieses Benchmarks. Eine ähnliche Beobachtung haben auch schon W. Ruan et al. [RLS14] bei der Verwendung der Hardwaretransaktionen gegenüber den spekulativen Softwaretransaktionen von GCC-TM gemacht. Dass Parallbrid ab zwei Kernen so langsam ausgeführt wird, hängt mit dem erhöhten Grad an Parallelität und dem extra Overhead, um diesen Grad zu gewährleisten, zusammen. Denn während GCC-TM sehr schnell in

den seriellen Modus wechselt, versucht Parallbrid die Transaktionen häufiger zu wiederholen und verliert so, aus dem oben beschriebenen Grund, viel Zeit durch Abbrüche. Die Ausführungszeit des Benchmarks mit Parallbrid wird mit zunehmenden Kernen wieder besser, als im Fall von zwei Kernen, da sich die vergeudete Rechenzeit auf mehr Kerne aufteilt. Bei acht Kernen ist die Ausführungzeit von Parallbrid auf 37,4 s gesunken. Dies ist immer noch ein Vielfaches der schnellst möglichen Ausführungzeit mit nur einem Kern, schlägt aber die Ausführungszeit von GCC-TM, welche bei acht Kernen auf 69,4 s degradiert ist.
Ein Analyse, wie viele Transaktionen bei zwei Kernen mit den einzelnen Transaktionstypen gestartet und erfolgreich abgeschlossen wurden, ist leider nicht so einfach. Denn in den Läufen mit Protokollierung dieser Informationen verhielt sich Parallbrid anders als ohne die Protokollierung. Die Ausführungszeit für zwei Kerne lag hierbei im Durchschnitt bei 165,68 s. Es konnte leider nicht endgültig geklärt werden, was dieses Verhalten hervorbringt. Eine Möglichkeit ist, dass durch die Protokollierung die Transaktionen nicht so häufig parallel ausgeführt werden und es daher zu weniger Konflikten kommt.
Ein weiterer Testlauf von Parallbrid mit einer Konfiguration von nur fünf Versuchen für Hardware- und drei für Softwaretransaktionen führte dazu, dass die Ergebnisse für zwei bis acht Kerne immer auf dem Niveau von acht Kernen aus Abbildung 6.1 lagen. Dies liegt daran, dass hierbei früher die seriellen Transaktionen, zur Gewährleistung des Fortschritts, übernehmen und nicht soviel Zeit durch die Abbrüche von Software- und Hardwaretransaktionen verloren geht. Bei dieser Variante wird also die Parallelität beschränkt, um zu einer schnelleren Ausführung zu kommen. Diese liegt für zwei bis fünf Kerne aber trotzdem ein Vielfaches hinter der von GCC-TM.

6.2.2 Intruder

In Abbildung 6.2 sind die Ergebnisse des Intruder-Benchmarks dargestellt. Dieser Benchmark gehört wieder zu der Kategorie der Benchmarks, bei denen sich GCC-TM mit zunehmender Kernzahl verschlechtert, da auch die Konflikte mehr werden. Parallbrid verhält sich diesbezüglich ähnlich wie bei dem Genome-Benchmark. Bei nur einem Kern ist Parallbrid mit 14,9 s sogar ein bisschen schneller als GCC-TM mit 15,5 s. Bei mehr Kernen wird die verlorene Arbeit und der Overhead für die Softwaretransaktionen aber deutlich bemerkbar. Parallbrid ist bei diesem Benchmark GCC-TM deutlich unterlegen.

6.2.3 KMeans

Die Ergebnisse des KMeans-Benchmarks sind in Abbildung 6.3 dargestellt. Bei diesem Benchmark verhalten sich GCC-TM und Parallbrid für ein bis fünf Kerne sehr ähnlich. Es ist kein Vorteil durch die erhöhte Parallelität von Parallbrid vorhanden, aber auch kein Nachteil. Ab sechs Kernen verschlechtert sich die Ausführungszeit von Parallbrid aber wieder. Bei sieben Kernen dauert die Ausführung des Benchmarks fast so lange wie mit nur einem Kern und bei acht sogar länger.

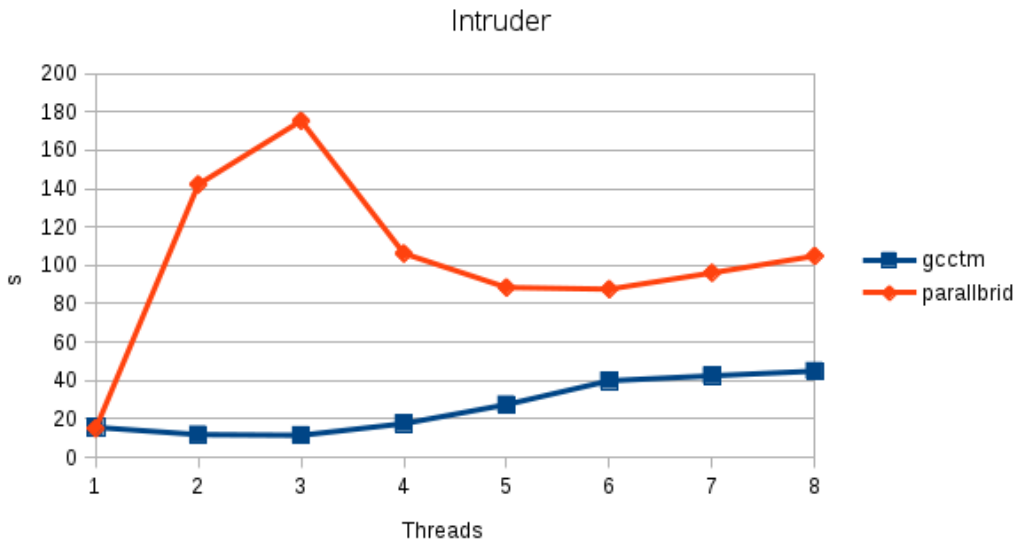


Abbildung 6.2: Ergebnisse des Intruder-Benchmarks

Tabelle 6.1 zeigt exemplarisch die Anzahl der gestarteten und abgeschlossenen Transaktionstypen von einem der sechs Threads, die diesen Benchmark ausgeführt haben.

	gestartet	committed
LiteHW	1856528	1422072
BFHW	334095	4319
SPECSW	132676	13342
IrrevocableSW	16561	16561
SerialAboSW	0	0
SglSW	40514	40514

Tabelle 6.1: Gestartete und comittete Transaktionen eines Threads bei der Ausführung des KMeans-Benchmarks mit sechs Threads

Ein Livelock der Hardwaretransaktionen, wie bei den vorherigen beiden Benchmarks, kann ausgeschlossen werden, da die meisten Transaktionen als LiteHW-Transaktionen ausgeführt wurden, und es zu wenigen Abbrüchen kam. Dass die Ausführungzeit ab sechs Kernen schlechter wird, hängt damit zusammen, dass bei mehr Kernen manche Transaktionen häufiger Konflikte haben und daher mehr Transaktionen in den langsameren Softwaremodus wechseln. Wie man in Tabelle 6.1 sieht, können nur ca. 10% der SpecSW-Transaktionen und knapp mehr als 1% der BFHW-Transaktionen einen erfolgreichen Commit vollziehen. Die anderen brechen ab und werden irgendwann in den seriellen Status eskaliert und als IrrevocSW-Transaktionen ausgeführt. Insgesamt ist die Anzahl der unwiderruflichen Transaktionen bei steigender Kernzahl immer größer.

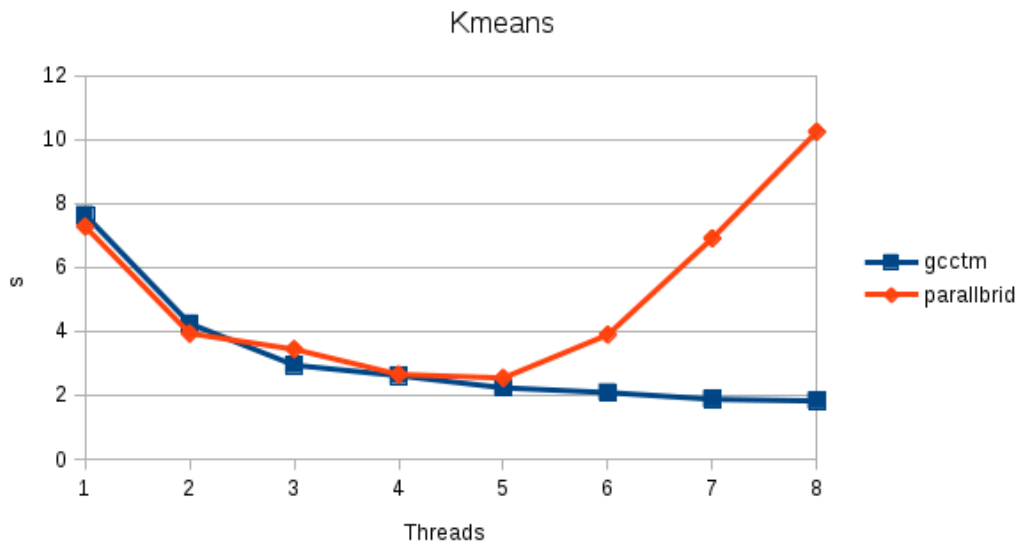


Abbildung 6.3: Ergebnisse des KMeans-Benchmarks

Vier Threads haben bei diesem Benchmark im Schnitt zusammen 50336 unwiderrufliche Transaktionen ausführen müssen. Bei acht Threads sind es dagegen zusammen im Schnitt 1598694 unwiderrufliche Transaktionen, die ausgeführt werden müssen. Dies und die verschwendete Arbeit bei den SpecSW- und BFHW-Transaktionen erklärt, warum die Laufzeit bei acht Threads so schlecht ist.

6.2.4 Labyrinth

Abbildung 6.4 zeigt die Ergebnisse des Labyrinth-Benchmarks, bei dem sich die Kurven von Parallbrid und GCC-TM sehr gleichen. Bei diesem Benchmark werden die Transaktionen selten zur gleichen Zeit ausgeführt, wodurch sich die Anzahl der Konflikte beschränkt. Parallbrid benutzt für diesen Benchmark fast ausschließlich LiteHW- und SglSW-Transaktionen. Diese beiden Transaktionstypen werden exklusiv zueinander ausgeführt und ähneln denen von GCC-TM. Dies erklärt auch die fast identischen Kurven. Parallbrid kann bei diesem Benchmark seine Fähigkeiten für mehr Parallelität nicht ausspielen, was aber auch zu keiner Bestrafung bei der Ausführungszeit führt.

6.2.5 Ssca2

In Abbildung 6.5 ist das Ergebnis des Ssca2-Benchmarks zu sehen. Parallbrid ist bei der Ausführung mit nur einem Kern schneller als GCC-TM. GCC-TM führt diese Transaktionen alle als serielle Softwaretransaktionen aus, während Parallbrid diese hauptsächlich als LiteHW-Transaktionen bearbeitet. Das schlechtere Verhalten von GCC-TM hängt wahrscheinlich mit dem Overhead für das globale Lock zusammen, welches serielle Transaktionen verwenden. Bei zwei bis vier Threads verhält sich Parallbrid ähnlich

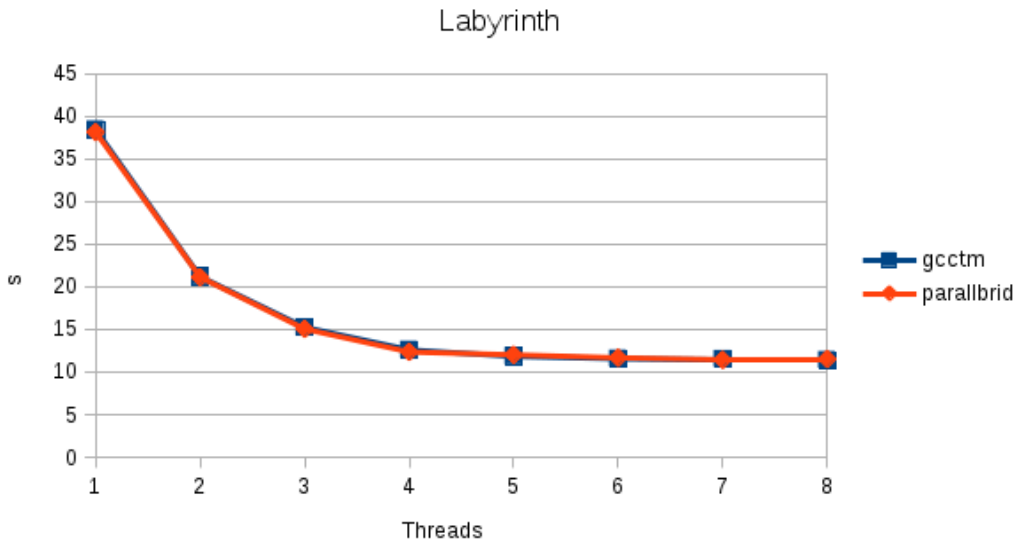


Abbildung 6.4: Ergebnisse des Labyrinth-Benchmarks

wie GCC-TM. Der Grund dafür ist, dass beide Systeme die Transaktionen hauptsächlich als Hardwaretransaktionen ausführen. Im Fall von Parallbrid sind dieses LiteHW-Transaktionen, die schnell sind.
Ab fünf Transaktionen verschlechtert sich die Laufzeit des Benchmarks mit Parallbrid aber deutlich gegenüber der Laufzeit mit GCC-TM. Der Grund dafür ist, dass die Anzahl der Konflikte steigt, wodurch mehr Transaktionen als Softwaretransaktionen ausgeführt werden. Wenn aber eine Softwaretransaktion läuft, können keine LiteHW-Transaktionen mehr benutzt werden und es müssen langsamere BFHW-Transaktionen benutzt werden. Diese brechen dann auch immer ab, wenn eine SpecSW-Transaktion einen Commit vollzieht oder in den unwiderruflichen Zustand wechselt.

	gestartet	committed
LiteHW	1269	598
BFHW	5353884	195757
SPECSW	7455563	2045658
IrrevocableSW	551998	551998
SerialAboSW	0	0
SglSW	516	516

Tabelle 6.2: Gestartete und comittete Transaktionen eines Threads bei der Ausführung des Ssca2-Benchmarks mit acht Threads

Tabelle 6.2 zeigt exemplarisch, welche Transaktionen ein Thread ausgeführt hat, der bei einer Ausführung von insgesamt acht Threads lief. Wie man sieht, gab es bei dieser Ausführung 5,4 Milionen Abbrüche bei SpecSW-Transaktionen und eine halbe Million

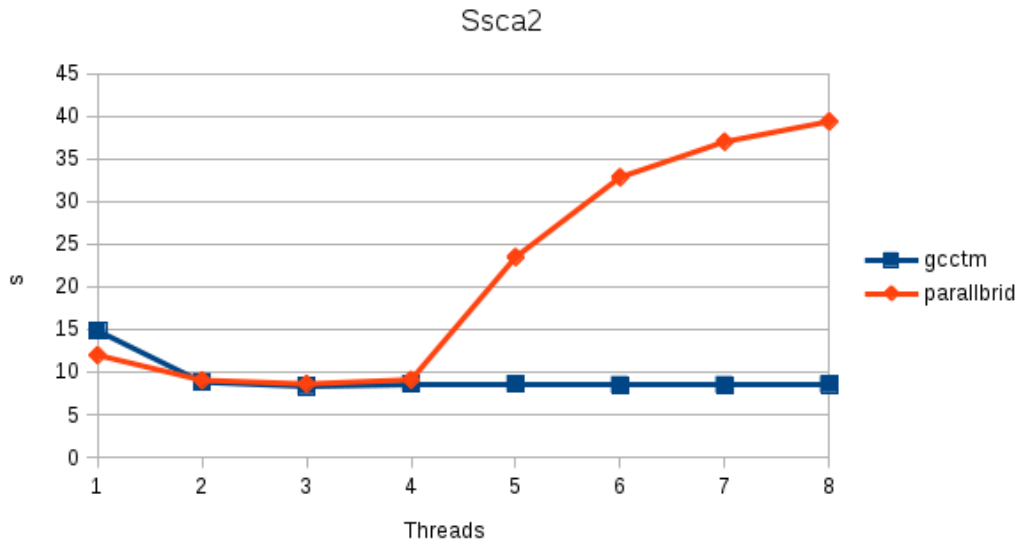


Abbildung 6.5: Ergebnisse des Ssca2-Benchmarks

bei BFHW-Transaktionen. Diese vergeudete Rechenzeit ist Schuld an dem schlechten Ergebnis von Parallbrid bei mehr als vier Kernen.

6.2.6 Vacation

Das Ergebnis des Vacation-Benchmarks ist in Abbildung 6.6 dargestellt. Wie man hier sehen kann ist die Ausführungszeit sowohl von Parallbrid als auch von GCC-TM am besten für nur einen Thread. Bei mehr Threads verschlechtert sich bei beiden die Laufzeit des Benchmarks, wobei sie bei GCC-TM stabil bleibt und sich bei Parallbird stets verschlechtert.
Bei diesem Benchmark kommen nur Transaktionen vor, die einen Userabort haben. Solche Transaktionen können weder als Hardwaretransaktionen noch als unwiderrufliche Transaktionen ausgeführt werden. Bei GCC-TM werden daher alle als serielle Transaktionen ausgeführt. Parallbrid führt die Transaktionen hauptsächlich als SerialAboSW-Transaktionen aus. Diese sind langsamer als GCC-TMs serielle Transaktionen, da SerialAboSW-Transaktionen nebenbei auch ein Writeset führen. Dies ist der Grund, warum Parallbrid bei einem Kern langsamer ist als bei GCC-TM. Bei mehr Kernen verschlechtert sich die Ausführungszeit mit Parallbrid, weil jeder Thread versucht, SpecSW-Transaktionen zu benutzen, wenn ein anderer Thread eine SerialAboSW-Transaktion benutzt. Diese SpecSW-Transaktionen sind aber fast nie erfolgreich, da es zu Konflikten kommt, die von den SerialAboSW-Transaktionen gewonnen werden. Die Fähigkeit zu parallelen Berechnungen durch SpecSW-Transaktionen, während IrrevocSW-Transaktionen ausgeführt werden, ist der Grund für das immer schlechtere Ergebnis bei steigender Anzahl von Kernen.

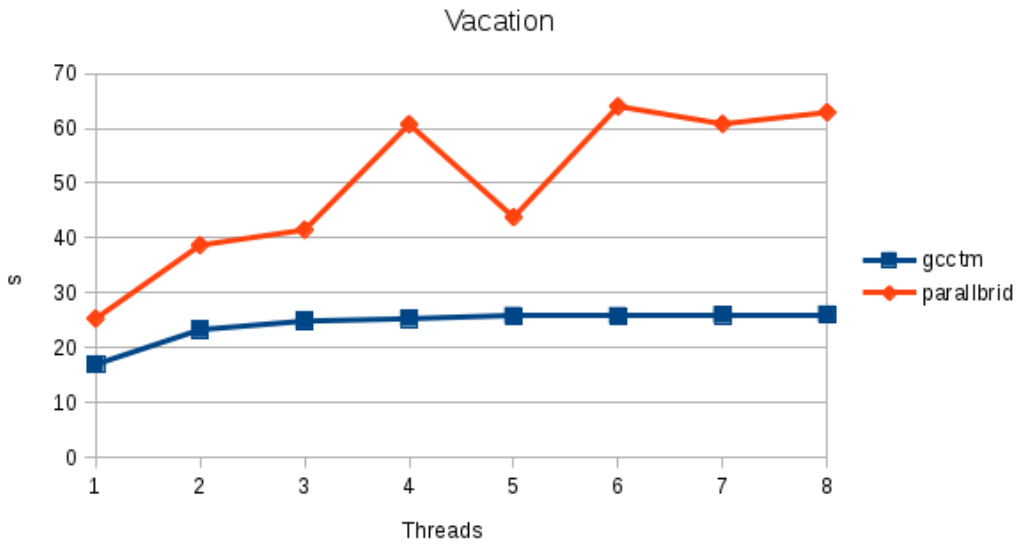


Abbildung 6.6: Ergebnisse des Vacation-Benchmarks

6.2.7 Yada

Die Ergebnisse des Yada-Benchmarks sind in Abbildung 6.7 zu sehen. Dieser Benchmark gehört auch wieder in die Kategorie der Benchmarks, bei denen mehr Threads als Einer, sowohl GCC-TM als auch für Parallbrid, zu einem schlechteren Ergebnis führen. Bei diesem Benchmark kommen wie bei Vacation auch wieder Transaktionen mit Userabort vor. Es sind aber nicht ausschließlich solche Transaktionen, sondern auch andere, die von Hardwaretransaktionen ausgeführt werden können. Bei einem Kern ist der Geschwindigkeitsunterschied zwischen GCC-TM und Parallbrid auf die langsammeren SerialAboSW-Transaktionen zurückzuführen, da die restlichen Transaktionen als LiteHW-Transaktionen ausgeführt werden. Die ansteigende Laufzeit bei Parallbrid für sechs oder mehr Kerne konnte nicht eindeutig geklärt werden. Es treten hierbei nicht mehr Konflikte auf und es werden auch nicht mehr Transaktionen abgebrochen. Wahrscheinlich hängt die höhere Laufzeit bei steigender Threadanzahl mit dem höheren Synchronisationsaufwand zwischen den Threads für die Konflikterkennung und Invalidierung zusammen.

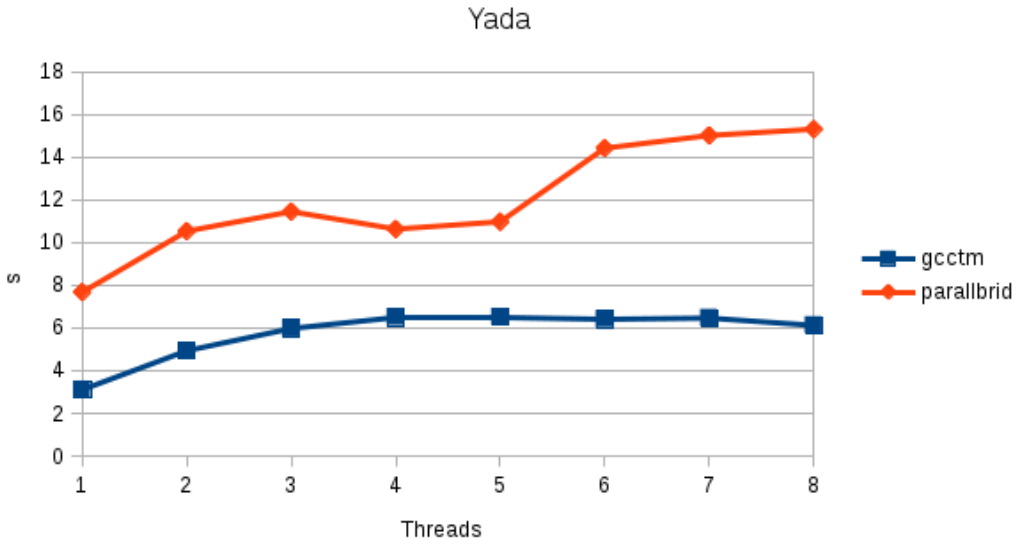


Abbildung 6.7: Ergebnisse des Yada-Benchmarks

7 Fazit

Parallbrid ist ein HyTM-System geworden, welches den Wunsch nach mehr Parallelität, als sie durch GCC-TM gegeben ist, erfüllt. Diese Parallelität führt aber nicht wie gehofft zu einer schnelleren Ausführung von Transaktionen. Denn einerseits dauern die Transaktionen durch den zusätzlichen Overhead, der die parallele Ausführung ermöglicht, länger als bei GCC-TM. Und andereseits zeigen die Benchmarks, dass die zusätzliche Parallelität auch zu mehr Konflikten und somit zu mehr Abbrüchen führt.
Die erhöhte Anzahl an Konflikten ist für die SpecSW-Transaktionen besonders ungünstig. Denn ein Konflikt zwischen zwei SpecSW-Transaktionen wird erst bei dem Commit einer dieser beiden Transaktionen entdeckt. Die Transaktion, die nicht den Commit gemacht hat, muss bei einem Konflikt dann neugestartet werden und ihre bisher vebrauchte Rechenzeit war vergeudet. Hierbei wird mit einer erhöhten Anzahl von Konflikten auch mehr Rechenzeit verloren. Die Verwendung eines *eager* STM-Systems, wie es in der STM-Implementierung von GCC-TM vorhanden ist, würde zu weniger verschwendeter Rechenzeit und wahrscheinlich zu einer besseren Skalierung führen. Leider ist die parallele Ausführung von Hardwaretransaktion und einem *eager* STM-System aus den Gründen, die am Ende von Kapitel 2.3.2 beschrieben werden, nicht möglich.
Der zusätzliche Overhead, den die Hardwaretransaktion, BFHW, und die seriellen Transaktionen, IrrevocSW und SerialAboSW, mit sich bringen, damit eine parallele Ausführung von SpecSW-Transaktionen möglich ist, scheint sich nicht auszuzahlen. Beim momentanen Stand der Dinge ist es daher zu empfehlen, die Transaktionsmodi lieber exklusiv zu nutzen und somit den Overhead einzusparen. Bei Arbeitsszenarien, die durch die STAMP-Benchmarksuite wiedergespiegelt werden, gibt es keinen Grund, Parallbrid anstatt der GCC-TM-Implementierung zu verwenden. In der STAMP-Benchmarksuite werden nur widerrufliche Transaktionen benutzt. Wenn in Zukunft ein typisches Einsatzszenario auch die Kombination von widerruflichen und unwiderruflichen Transaktionen enthält, die nicht viele Konflikte erzeugen, dann kann Parallbrid möglicherweise auftrumpfen.

8 Ausblick

Die Ideen, die zu einer Verbesserung von Parallbrid führen können, kann man in zwei Kategorien unterteilen. Dieses sind Veränderungen, die beim heutigem Stand der Technik vorgenommen werden können, aber auch welche, die nur mit einem erweiterten Typ von HTM-Unterstützung funktionieren.

Verbesserungen beim heutigen Stand der Technik:

- Das Managment von Transaktionen, die neustarten müssen, unterliegt in der momentanen Fassung von Parallbrid einem linearen Ablauf. Erst wird versucht eine Transaktion als Hardwaretransaktion, dann als SpecSW-Transaktion und dann als unwiderrufliche Transaktion auszuführen. Wenn also eine Transaktion T startet, während eine andere Transaktion im seriellen Modus ist, dann werden alle Versuche scheitern, T als Hardwaretransaktion auszuführen. T wird dann als langsamere SpecSW-Transaktion ausgeführt. Ist die serielle Transaktion abgeschlossen und Ts Ausführung muss neustarten, dann startet T wieder als SpecSW-Transaktion. Eine serielle Transaktion kann also somit eine andere Transaktion daran hindern jemals als Hardwaretransaktion ausgeführt zu werden.
  Der Vorschlag zur Verbesserung ist, dass eine Transaktion, die neustarten muss, nicht nur als SpecSW-Transaktion neugestartet wird, sondern auch wieder als Hardwaretransaktion ausgeführt werden kann.

- Eine andere Idee wäre es, den Start von Hardwaretransaktionen zu verzögern, wenn eine SglSW-Transaktion ausgeführt wird. Dieser Transaktionstyp erlaubt sowieso keine parallelen Ausführungen von anderen Transaktionen, sodass auch die SpecSW-Transaktionen, zu denen eine Hardwaretransaktion eskaliert, warten müssen.

- Eine andere Implementierung der Read- und Writesets könnte von Vorteil sein, da die Bloomfilter für kleine Transaktionen einen sehr großen Overhead bedeuten. Dieser Overhead der Bloomfilter amortisiert sich bei Transaktionen mit vielen Speicherzugriffen. Für BFHW-Transaktionen und die beiden seriellen Transaktionen, die schnell abgeschlossen werden sollen, ist dieser Overhead aber zu viel und es könnte ein anderes Schema für die Sets von Vorteil sein.

- Der letzte Vorschlag für Verbesserungen in dieser Kategorie betrifft den Commit. Bei Parallbrid sind alle Softwaretransaktionen beim Commit serialisiert. Dieses ist nicht zwingend notwendig und könnte durch ein anderes Commit-Schema, welches einen parallelen Commit erlaubt, verbessert werden. Eine Commit-Strategie, wie

sie RingSTM [SMP08] vorschlägt, könnte hier weiterhelfen. Es muss aber hierbei getestet werden, ob der Overhead für einen parallelen Commit nicht zu einer Verschlechterung der Ausführungszeit führt.

Wenn in zukünftigen Prozessorgenerationen die Unterstützung für HTMs ausgebaut wird, dann unterstützen diese vielleicht auch *escape actions*. Dieses sind Aktionen, die eine Hardwaretransaktion ausführen kann, ohne dass die einen direkten Einfluss auf die transaktionale Ausführung dieser Transaktion hat. Sie können also zur Kommunikation zwischen Hardwaretransaktionen und dem STM-System benutzt werden.

- Wenn *escape actions* zur Verfügung stehen, dann kann die volle Parallelität von Invyswell benutzt werden, die auch einen Commit von Hardwaretransaktionen parallel zu SpecSW- und IrrevocSW-Transaktionen erlaubt. Denn mit diesen Aktionen kann der *opacity*-Fehler von Invyswell umgangen werden, da das Commitlock nicht mehr Teil der eigenen Sets ist, da es in einem nicht transaktionalen Kontext gelesen wird.

- Eine weitere Möglichkeit mit *escape actions* den Ablauf zu verbessern, wäre es, dass BFHW-Transaktionen einen Contention Manager nach einer Commiterlaubnis fragen, um so zu gewährleisten, dass Transaktionen, die schon viel Rechenzeit verbraucht haben, nicht abgebrochen werden.

- Der letzte Vorschlag betrifft die Verwendung des *lazy*-STM-Systems in Parallbrid. Denn wenn eine Kommunikation zwischen Hardware- und Softwaretransaktionen möglich ist, dann kann die *lazy*-STM-Implementierung der SpecSW-Transaktionen durch eine *egaer*-Implementierung wie bei GCC-TM ausgestauscht werden. Hardwaretransaktionen können dann mit den *escape actions* prüfen, ob sie auf einen Speicherbereich zugreifen dürfen.

Literaturverzeichnis

[AAK+06] Ananian, C. S. ; Asanović, Krste ; Kuszmaul, Bradley C. ; Leiserson, Charles E. ; Lie, Sean: Unbounded Transactional Memory. In: *ASPLOS XII Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (2006)

[AKW+08] Ansari, Mohammad ; Kotselidis, Christos ; Watson, Ian ; Kirkham, Chris ; Luj´an, Mikel ; Jarvis, Kim: Lee-TM: A Non-trivial Benchmark for Transactional Memory. In: *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing* (2008)

[ATSG12] Adl-Tabatabai, Ali-Reza ; Shpeisman, Tatiana ; Gottschlich, Justin: *Draft Specification of Transactional Language Constructs for C++ Version 1.1.* Website, Feb 2012. – https://sites.google.com/site/tmforcplusplus/C++TransactionalConstructs-1.1.pdf

[Bal16] Balzer, Karl: *Parallbrid Repository.* Website, Aug 2016. – https://github.com/karlcbalzer/Parallbrid.git

[CBM+08] Cascaval, C. ; Blundell, C. ; Michael, M. ; Cain, H. W. ; Wu, P. ; Chiras, S. ; Chatterjee, S.: Software transactional memory: Why is it only a research toy? In: *ACM Queue - The Concurrency Problem* (2008)

[CBO+11] Casper, Jared ; Bronson, Nathan G. ; Oguntebi, Tayo ; Hong, Sungpack ; Kozyrakis, Christos ; Olukotun, Kunle: Hardware Acceleration of Transactional Memory on Commodity Systems. In: *ASPLOS XVI Proceedings of the sixteenth international conference on Architectural support for programming languages and operating system* (2011)

[CGS+14] Calciu, Irina ; Gottschlich, Justin ; Shpeisman, Tatiana ; Pokam, Gilles ; Herlihy, Maurice: Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In: *PACT '14 Proceedings of the 23rd international conference on Parallel architectures and compilation* (2014)

[DDS+10] Dalessandro, L. ; Dice, D. ; Scott, M. ; Shavit, N. ; Spear, M.: Transactional Mutex Locks. In: *Euro-Par'10 Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II* (2010)

[DSS06] Dice, Dave ; Shalev, Ori ; Shavit, Nir: Transactional Locking II. In: *DISC'06 Proceedings of the 20th international conference on Distributed Computing* (2006)

[DSS10] Dalessandro, Luke ; Spear, Michael F. ; Scott, Michael L.: NOrec: Streamlining STM by Abolishing Ownership Records. In: *PPoPP '10 Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2010)

[GVS10] Gottschlich, Justin E. ; Vachharajani, Manish ; Siek, Jeremy G.: An efficient software transactional memory using commit-time invalidation. In: *CGO '10 Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (2010)

[GW] GCC-Wiki: *Installing GCC*. Website, . – https://gcc.gnu.org/wiki/InstallingGCC

[GW12] GCC-Wiki: *Transactional Memory in GCC*. Website, 2012. – https://gcc.gnu.org/wiki/TransactionalMemory

[HWC+04] Hammond, Lance ; Wong, Vicky ; Chen, Mike ; Carlstrom, Brian D. ; Davis, John D. ; Hertzberg, Ben: Transactional Memory Coherence and Consistency. In: *ISCA '04 Proceedings of the 31st annual international symposium on Computer Architecture* (2004)

[Int12] Intel: *Intel® Architecture Instruction Set Extensions Programming Reference*. Website, 2012. – https://software.intel.com/sites/default/files/m/9/2/3/41604

[Jen10] Jenkins, B.: *SpookyHash: a 128-bit non-cryptographic hash*. Website, 2010. – http://burtleburtle.net/bob/hash/spooky.html

[MBM+06] Moore, Kevin E. ; Bobba, Jayaram ; Moravan, Michelle J. ; Hill, Mark D. ; Wood, David A.: LogTM: Log-based Transactional Memory. In: *Proceedings of the Twelfth IEEE Symp. on High-Performance Computer Architecture* (2006)

[MCKO08] Minh, Chi C. ; Chung, JaeWoong ; Kozyrakis, Christos ; Olukotun, Kunle: STAMP: Stanford Transactional Applications for Multi-processing. In: *Proceedings of the IEEE International Symposium on Workload Characterization,Seattle, WA* (2008)

[RHL05] Rajwar, Ravi ; Herlihy, Maurice ; Lai, Konrad: Virtualizing Transactional Memory. In: *ISCA '05 Proceedings of the 32nd annual international symposium on Computer Architecture* (2005)

[RLS14] Ruan, Wenjia ; Liu, Yujie ; ; Spear, Michael: STAMP Need Not Be Considered Harmful. In: *9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2014), Salt Lake City, UT* (2014)

[SMP08] Spear, Michael F. ; Michael, Maged M. ; Praun, Christoph von: RingSTM: Scalable Transactions with a Single Atomic Instruction. In: *SPAA '08 Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures* (2008)

[WGW+12] Wang, Amy ; Gaudet, Matthew ; Wu, Peng ; Amaral, José N. ; Ohmacht, Martin ; Barton, Christopher ; Silvera, Raul ; Michael, Maged: Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In: *PACT '12 Proceedings of the 21st international conference on Parallel architectures and compilation techniques* (2012)

[YBM+07] Yen, Luke ; Bobba, Jayaram ; Marty, Michael R. ; Moore, Kevin E. ; Volos, Haris ; Hill, Mark D. ; Swift, Michael M. ; Wood, David A.: LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In: *HPCA '07 Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (2007)