Transactional Memory: Invyswell und BlueGene/Q TM

Karl Balzer

29. September 2015

Inhaltsverzeichnis

1 Einleitung

Transactional Memory (TM) ist ein Programmiermodell, das eine Alternative zur Programmierung mit *locks* darstellt. Letzteres liefert eine ähnlich gute oder schlechtere Skalierung, wenn die Anzahl der threads steigt, und die Programmierung ist insgesamt weniger robust gegenüber Fehlern. Zudem vereinfacht es die Komposition von Programmabschnitten, die durch *mutual exclusion* geschützt sind.
Bisher werden beim Programmieren von Systemen mit geteiltem Speicher häufig *locks* verwendet. Diese *locks* schützen die gemeinsam genutzen Speicherbereiche vor Lese- oder Schreibzugriffen, die zu einem inkonsistenten Datenbestand oder einem fehlerhaften Programmverhalten führen können. Wählt man die Granularität der *locks* zu grob, z. B. ein *lock* für alle geteilten Speicherbereiche, so schadet dies der Skalierbarkeit, da alle *threads* durch das *lock* serialisiert werden. Wird die Granularität sehr fein gewählt, steigert dies die Komplexität der Programmcodes und die Programmierung wird fehleranfälliger, so dass es zu *deadlock* oder *starvation* kommen kann. Programme, die *Transactional Memory* benutzen, skalieren ähnlich wie solche, die fein granulierte *locks* verwenden. Sie sind aber so einfach zu schreiben, wie Programme die grob granulierte *locks* verwenden. Im Programmcode werden hierzu die Programmabschnitte, die durch *mutual exclusion* geschützt werden müssen, gekennzeichnet. Sie werden dann als Transaktion ausgeführt und das *Transactional Memory System* ist für dessen korrekte Ausführung zuständig.
Eine Transaktion ist durch die *ACI*-Eigenschaften (*Atomicy*, *Consistency*, *Isolation*) definiert. [1]
Atomicity bedeutet, dass eine Transaktion nur ganz oder gar nicht ausgeführt werden kann. Entweder sie ist erfolgreich und alle von ihr gemachten Änderungen treten in Kraft oder sie war nicht erfolgreich und keine Änderung tritt in Kraft.
Consistency bedeutet, dass eine Transaktion das System nicht in einen inkonsistenten Zustand bringen darf. Wenn ein System in einem konsistenten Zustand ist, darf das Ausführen oder Abrrechen einer Transaktion nicht zu einem inkonsistenten Zustand führen.
Isolation bedeutet, dass eine laufende Transaktion keinen Einfluss auf andere laufende Transaktionen hat. Der Zustand des Systems, nachdem die Transaktionen erfolgreich waren, ist so, als wenn die Transaktionen serialisiert gelaufen wären.
Transaktionen werden meistens spekulativ ausgeführt und können in vielen TM-Systemen nebenläufig und parallel laufen. Das TM-System sorgt dafür, dass die *ACI*-Eigenschaften eingehalten, Konflikte zwischen laufenden Transaktionen detektiert und aufgelöst und Eigenschaften wie *forward progress*, *deadlock freedom* und *starvation freedom* gewährleistet

[1] Die bei Datenbanktransaktionen darüber hinaus erforderliche Eigenschaft *Durability* spielt bei TM keine Rolle.

werden.
Umsetzungen von *Transactional Memory* existieren als Software (STM)[DDS+10, DSS06, KGH+11, SMP08, DSS10], Hardware (HTM)[HWC+04, MBM+06a, YBM+07, AAK+06, RHL05] und als hybrider Ansatz aus Software und Hardware (HyTM)[CGS+14, WGW+12]. Die Softwarevarianten spielen hauptsächlich in der Wissenschaft eine Rolle [CBM+08], da Programme mit vielen kurzen Transaktionen, aufgrund der zusätzlichen Befehle für das STM-System, langsamer sind als Programme, die *locks* mit feiner Granularität verwenden. HTM-Systeme sind hinsichtlich Geschwindigkeit ähnlich schnell wie der Einsatz von *locks* mit feiner Granularität. Die meisten HTM-Systeme existieren aber nur in Prototyp-Prozessoren oder in Prozessor-Simulatoren. Die wenigen HTMs in Mainstream-Prozessoren, z. B. Intel Haswell[Int12] und IBM BlueGene/Q[WGW+12], sind *best-effort* HTMs. *Best-effort*-HTMs können zwar Transaktionen ausführen, gewährleisten aber nicht, dass jede Transaktion jemals erfolgreich abgeschlossen werden kann. Resourcenbeschränkungen oder Befehle, wie *I/O*, sind Gründe, warum Transaktion in so einem System scheitern können. Hybridsysteme verbinden die beiden Ansätze und kombinieren eine HTM, meistens eine *best effort* HTM, mit einer STM. Die HTM ist für die kurzen Transaktionen zuständig, so dass diese schnell ausgeführt werden können. Die STM übernimmt alle Transaktionen die nicht mit der Hardware ausgeführt werden konnten. Es wird also eine *best effort* HTM durch Ergänzung mit einer STM zu einem vollständigen TM-System erweitert. Zwei hybride Ansätze für *Transactional Memory*, die die oben genannten Mainstream-Prozessoren benutzen, werden in Kapitel 3 und 4 näher beschrieben: In Kapitel 3 wird die Funktionsweise von Invyswell[CGS+14], einem HyTM für Haswell, und in Kapitel 4 ein TM-System für BlueGene/Q[WGW+12] vorgestellt. Zunächst werden aber im Folgenden in Kapitel 2 die generelle Funktionsweise sowie besondere Eigenschaften von Transactional Memory erläutert.

2 Transactional Memory: Funktionsweise und Eigenschaften

Wenn *Transactional Memory* eingesetzt wird, werden Programmabschnitte als Transaktionen ausgeführt, die im Code als solche gekennzeichnet sind. Diese Kennzeichnung findet meist durch ein *begin* und *end* statt. Kommt ein Thread bei einem transaktionalen *begin* an, speichert er seinen aktuellen Zustand (*checkpointing*) für den Fall, dass die Transaktion fehlschlägt, und versucht diese dann auszuführen. Ist die Ausführung erfolgreich, dann ist die Transaktion *commited* und alle ihre Schreibvorgänge sind im System für alle anderen Threads sichtbar. Kommt es zu einem Konflikt, bricht die Ausführung ab. Der Thread kehrt dann zu seinem *checkpoint* zurück und versucht die Transaktion zu wiederholen. Konflikte können durch Validierungen während der Ausführung oder in der *commit*-Phase auftreten. Damit das TM-System Konflikte erkennen (*conflict detection*) und die geschriebenen Daten veröffentlichen oder verwerfen kann (*version managment*), führt das TM-System eine *read-* und ein *writeset* für jede Transaktion.
Jedes Mal, wenn eine Transaktion ein *read* ausführt, wird die Adresse und das Gelesene bzw. gewisse Metadaten, wie eine Versionsnummer, im *readset* eingetragen. Führt eine Transaktion eine Validierung aus, prüft sie, ob die Einträge im *readset* noch zum Speicher passen. Ist dies erfolgreich, dann kann die Transaktion fortfahren, schlägt die Validierung fehl, muss die Transaktion abbrechen und neu beginnen. Im *writeset* landen alle Adressen und zugehörigen Werte, an die eine Transaktion während ihrer Laufzeit schreibt. Sie werden dort spekulativ zwischengespeichert und bei erfolgreichem *commit* im Speicher veröffentlicht. Beim Lesen einer Adresse aus dem Speicher wird auch immer das *writeset* durchsucht, da die Transaktion schon vorher an dieselbe Adresse geschrieben haben könnte. In dem Fall wird der aktuellere Wert aus dem *writeset* benutzt.

2.1 Konflikterkennung

Conflict detection kann, je nach TM-System, mit einer Granularität von Speicheradressen, *cache lines*, *pages* oder Objekten stattfinden. Je gröber die Granularität ist, desto höher ist die Wahrscheinlichkeit von *false conflicts* bei der Erkennung. Wenn Konflikte erst in der *commit*-Phase erkannt und aufgelöst werden, spricht man von einem *lazy conflict managment*. Es gibt aber auch TM-Systeme, die schon früher Konflikte erkennen. Hierbei werden zum Beispiel beim Ausführen eines *reads* auch die *writesets* der anderen laufenden Transaktionen geprüfft und bei einem *write* die *read* und *write sets*. Tritt hierbei ein Konflikt auf, entscheidet meist ein *contention manager*, welche der beiden Transaktionen neustarten muss. Bei dieser frühen Konflikterkennung spricht man von

eager conflict detection.

2.2 Versionverwaltung

Auch beim *version managment* unterscheidet man zwischen *lazy* und *eager*. Speichert eine Transaktion alle ihre *writes* spekulativ zwischen und nimmt erst in der *commit*-Phase Änderungen am Speicher vor, dann verwendet das TM-System ein *lazy version managment*. Werden allerdings alle Änderungen am Speicher direkt beim *write* vollzogen und im *writeset* die alten Werte gespeichert, so wird ein *eager version managment* verwendet. Letzteres erfordert, dass Transaktionen die eine Adresse lesen wollen, den Wert erst lesen dürfen, wenn die Transaktion, die ihn geschrieben hat, ihren *commit* abgeschlossen hat. Falls solch eine Transaktion abbricht, müssen alle Werte im Speicher wieder durch ihre Originale aus dem *writeset* ersetzt werden.

2.3 Commit

Hat eine Transaktion alle ihre Programmbefehle ausgeführt, wechselt sie in die *commit*-Phase. In dieser validiert sie ihr *read set* und besorgt sich eine *commit*-Erlaubnis, zum Beispiel mit einem *commit token* oder durch das Sichern der entsprechenden Speicherbereiche. Ist die Validierung erfolgreich und eine *commit*-Erlaubnis erteilt, dann wird, wenn nötig, noch das *writeset* veröffentlicht, womit die Transaktion dann abgeschlossen (*commited*) ist. Ist ein TM-System mit *lazy conflict detection* und *lazy version managment* gegeben, dann könnte die *commit*-Phase mit drei Schritten wie folgt aussehen. Als erstes werden alle Speicheradressen aus *read-* und *writeset* für den alleinigen Zugriff gesichert. Dies kann durch *locks*, *ownership records* oder eine andere geeignete Zugriffsstruktur geschehen. Als zweiter Schritt wird nun das *readset* validiert. Kommt es zu einem Konflikt, werden die Speicheradressen wieder freigegeben und die Transaktion bricht ab und startet neu. War die Validatierung erfolgreich, hat die Transaktion die Erlaubnis für den *commit*. Im letzten Schritt werden nun noch die Einträge des *writesets* im Speicher veröffentlicht und die Speicheradressen wieder freigegeben.
Bei einem System mit *eager conflict detection* und *eager version managment* kann die *commit*-Phase deutlich kürzer ausfallen. Wenn ein Thread die *commit*-Phase erreicht, gab es keine Konflikte bzw. sie wurden zu Gunsten dieses Threads gelöst, es ist also keine weitere Validierung notwendig. Da auch die aktuellen Werte schon im Speicher stehen, ist die Transaktion mit dem Betreten der *commit*-Phase somit abgeschlossen. Bei dieser Variante ist der *commit* schnell, was vorteilhaft in Programmen mit wenig Konflikten ist. Das Neustarten einer Transaktion ist aber langsam, da die Werte aus dem *writeset* wieder hergestellt werden müssen, was zu Leistungseinbußen in Programmen mit häufigen Konflikten führen kann. Bei der *commit*-Strategie gibt es unterschiedliche Ansätze, wie zum Beispiel die oben benutzte *lock and validate*-Strategie, *2 phase commit* oder eine Strategie, die mit einer *compare and swap*-Anweisung auskommt, wie bei RingSTM[SMP08].

2.4 Fortschrittsgarantien

Das oben beschriebene *lazy* TM-System gewährleistet *progress*, da immer irgendeine Transaktion einen *commit* machen kann. Mit dieser Eigenschaft ist nur gemeint, dass es einen Fortschritt gibt, also irgendeine Transaktion erfolgreich beendet wird. Es ist hiermit nicht gemeint, dass jede Transaktion irgendwann erfolgreich abgeschlossen werden muss. Bei Systemen mit *eager conflict detection* muss die Auflösung von Konflikten so gestaltet sein, dass *progress* gewährleistet ist. Bei einer *requester wins*-Strategie könnten sich zwei Transaktionen dauerhaft gegenseitig beenden (*livelock*). Eine einfache Strategie zur Gewährleistung von *progress* ist, den Konflikt immer zu Gunsten der älteren Transaktion aufzulösen. Ein *contention manager* kann hier aber auch komplexere Strategien benutzten, die Prioritäten, bisherige Wiederholungen und andere Details miteinbeziehen können.
Forward progress ist durch das oben genannte *lazy* TM-System allerdings nicht gegeben. Eine Transaktion, die sehr lange brauchen würde, könnte immer wieder von kurzen Transaktionen abgebrochen und somit nie ausgeführt werden (*starvation*). Eine einfache Möglichkeit *forward progress* nachzurüsten, wäre zu Beginn der *commit*-Phase einen *contention manager* um *commit*-Erlaubnis zu fragen. Dieser könnte die Erlaubnis verweigern, wenn eine andere Transaktion, die einen Grenzwert an Wiederholungen überschritten hat, hierdurch abgebrochen werden würde. Der *contention manager* sorgt damit dafür, dass jede Transaktion irgendwann ausgeführt wird, somit also für *forward progress*. Auch in Systemen mit *eager conflict detection* kann ein *contention manager* für die Gewährleistung von *forward progress* eingesetzt werden.
Die Gewährleistung von *deadlock freedom* ist durch das Konzept von *transactional memory* gegeben, da bei Konflikten eine der beiden Transaktionen neugestartet wird. Werden in der *commit*-Phase *locks* verwendet, dann muss eine *locking*-Strategie verwendet werden, die *deadlocks* ausschließt, wie zum Beispiel das Ordnen der Adressen. Wird in einem System mit *eager conflict detection* beim Auflösen von Konflikten statt einer *abort*- eine *stall*-Strategie verwendet, muss auch hier sichergestellt sein, dass nicht zwei Transaktionen aufeinander warten. Dies kann durch ein *flagg* in jeder Transaktion gewährleistet werden, das immer gesetzt wird, wenn eine andere Transaktion auf diese warten muss. Kommt es zu einem potentiellen Konflikt, so darf eine Transaktion mit gesetztem *flagg* nicht auch warten. Entweder sie gewinnt den Konflikt oder sie muss abbrechen und von vorne beginnen.

2.5 Contention Manager

Ein *contention manager* kann in TM-Systemen für die Einhaltung der oben genannten Eigenschaften dienen. Seine ursprüngliche Aufgabe ist die Regelung des Transaktionsflusses. In Systemen mit *eager conflict detection* löst er Konflikte, die während der Ausführung zweier Transaktionen auftreten können, und versucht die Wahrscheinlichkeit für ein Auftreten zukünftiger Konflikte zu verringern. Ein Konflikt tritt zum Beispiel auf, wenn eine Transaktion T_1 einen Wert liest, den eine andere Transaktion T_2 schon

in ihrem *writeset* hat. Umgekehrt kommt es auch zum Konflikt, wenn T_1 an einem Ort schreiben will, der schon im *readset* von T_2 ist. Die Strategie mit der ein *contention manager* Konflikte auflöst, ist nicht umbedingt durch das TM-System vorgegeben, es können häufig unterschiedliche Strategien benutzt werden. Scherer et al. [IS05] haben gezeigt, dass die Wahl der Strategie einen Einfluß auf den Durchsatz von Transaktionen hat, und es nicht eine perfekte Strategie gibt, sondern diese vom *workload* abhängig sind.

2.6 Verschachtelung

Die meisten TM-Systeme unterstützen *nesting*, also die Möglichkeit Transaktionen aus einer Transaktion heraus aufzurufen. Diese Möglichkeit ist wichtig, wenn eine Transaktion als Komposition mehrer Transaktionen zusammengesetzt werden soll. Es gibt hierbei die Ansätze des *nesting by flattening* und des *nesting with partial abort*. *Nesting by flattening* bedeutet, dass die inneren Transaktionen 'flachgeklopft' werden, also in die äußere Transaktion integriert werden. Kommt eine Transaktion an einem *begin* einer inneren Transaktion an, so wird keine neue Transaktion gestartet, sondern der Inhalt der Transaktion von der Äußeren ausgeführt und das *nesting level* um einen erhöht. Das *end* dieser inneren Transaktion beendet nun keine Transaktion mehr, sondern dekrementiert nur das *nesting level*. Erreicht eine Transaktion ein transaktionales *end* und ist das *nesting level* gleich null, so ist dies das Ende der äußeren Transaktion und es kann ein *commit* stattfinden. Kommt es bei einer inneren Transaktion zu einem Konflikt, der in einem Abbruch resultiert, so bricht dies die gesammte Transaktion, also alle bis zur äußersten, ab. Bei *nesting with partial abort* wird bei einem Konflikt einer inneren Transaktion nicht die gesammte Transaktion abgebrochen, sondern nur die Innere. Dies wird dadurch möglich, dass die innere Transaktion nicht in die äußere integriert wird, sondern einen eigenen *checkpoint* sowie ein eigenes *read-* und *writeset* besitzt. Wenn die innere Transaktion mit ihrer Ausführung fertig ist, dann führt sie keinen *commit* aus, sondern die äußere Transaktion läuft weiter. Die Änderungen der inneren Transaktionen werden gemeinsam mit dem *commit* der äußersten Transaktion validiert und veröffentlicht. Die Nutzung von *nesting with partial abort* hat den Vorteil, dass der schon geleistete Rechenaufwand einer äußeren Transaktion nicht verschenkt wird, wenn eine innere abgebrochen wird. Ihr Nachteil ist aber, dass die Realisierung aufwendiger und der Durchsatz bei Abwesenheit von Konflikten geringer ist als bei *flattening*.
LogTM-NE[MBM+06b] bietet auch noch die Möglichkeit des *open nestings*. Hier darf eine innere Transaktion einen *commit* machen, bevor die äußere Transaktion abgeschlossen ist. Es müssen aber Kompensationsaktionen bei der äußeren Transaktion registriert werden, die ausgeführt werden, falls die äußere Transaktion abbricht. Der Vorteil dieses frühen *commits* ist, dass, wenn zum Beispiel ein *resource allocator* in der inneren Transaktion benutzt wird, dieser danach von anderen Transaktionen benutzt werden kann, bevor die äußere Transaktion fertig ist. Die Benutzung von *open nesting* ist aber gefährlich, da der Programmierer dafür verantwortlich ist, dass das Programm korrekt arbeitet.

2.7 Unumkehrbare Aktionen

Eine Eigenschaft von Transaktionen ist, dass sie bei einem Konflikt neustarten können, also ihre bisher getätigten Aktionen verwerfen oder rückgängig machen können. Es gibt aber Befehle und Aktionen, die sich nicht rückgängig machen lassen, wie zum Beispiel der Aufruf von *free* oder *I/O*-Operationen im Allgemeinen. Viele TM-Systeme unterstützen daher nur einen begrenzten Befehlssatz, der innerhalb von Transaktionen erlaubt ist. *I/O* und viele Systemfunktionen sind hierbei meist nicht erlaubt. TM-Systeme, die einen vollständigen Befehlssatz unterstützen und auch *I/O* erlauben, nutzen meist *irrevocable transactions*, Transaktionen, die nicht abgebrochen werden dürfen und daher jeden Konflikt gewinnen. Damit eine Transaktion auf jeden Fall erfolgreich abgeschlossen werden kann, darf immer nur eine Transaktion, die als *irrevocable* eingestuft ist, laufen. Würden zwei oder mehr *irrevocable transactions* laufen, könnte es zu einem Konflikt kommen, der mit dem Neustart einer der beiden Transaktionen gelöst werden müsste. Dies ist unerwünscht und kann zu Fehlverhalten oder Datenverlust führen. Parallel zu einer *irrevocable transaction* dürfen in einigen TM-Systemen andere Transaktionen laufen, wie *read only* Transaktionen oder *writer* Transaktionen, die keinen *irrevocable code* enthalten. Ist eine *nested transaction* als *irrevocable* eingestuft, so muss bei ihrem Start die äußere Transaktion in den Status *irrevocable* erhoben werden. Ist dies nicht möglich, da bereits eine andere nicht abbrechbare Transaktion T_i läuft, so muss mit dem Ausführen der *nested transaction* gewartet werden, bis T_i abgeschlossen ist.

2.8 Atomarität

Bei TM-Systemen unterscheidet man zwischen *strong* und *weak atomicy*. Damit ist der Einfluss von einer nicht transaktionalen Ausführung auf das Verhalten des Transaktionssystems gemeint. Unter *strong atomicity* versteht man, dass das Transaktionssystem davon Kenntnis nimmt, wenn geteilte Speicherbereiche außerhalb von Transaktionen verändert werden. Eine laufende Transaktion bricht in so einem System ab, wenn ein zuvor gelesener Wert von einer Nicht-Transaktion verändert wurde. Dies gewährleistet eine stets korrekte Ausführung von Transaktionen, erschwert aber das *debugging*. Systeme mit *weak atomicity* gewährleisten nur Atomarität zwischen den Transaktionen. Ändert eine Nicht-Transaktion einen Speicherbereich, der zuvor von einer Transaktion gelesen wurde, so führt dies nicht zum Konflikt. Solche Systeme sind meist einfacher zu konstruieren, das korrekte Zusammenspiel zwischen transaktionalem und nicht transaktionalem Code liegt dann aber in den Händen des Programmierers.

2.9 Opacity

Die *opacity*-Eigenschaft beschreibt den Umgang eines TM-Systems mit inkonsistenten Datenbeständen. Wenn Transaktionen unter keinen Umständen inkonsistente Daten lesen können, dann erfüllt das TM-System die *opacity*-Eigenschaft. Systeme, wie zum Beispiel solche mit *lazy conflict detection*, die nur zur *commit*-Zeit Validierungen

durchführen, erfüllen nicht die *opacity*-Eigenschaft. Bei diesen Systemen könnte eine Transaktion, deren *readset* nicht mehr konsistent zum Speicher ist, bis zum *commit* weiterlaufen und inkonsistente Daten lesen. Solche *doomed transactions* können zu *null-pointer exceptions* und unendlichen Schleifen führen.

2.10 Escape Actions

Escape actions sind dafür gedacht, Programmabschnitte innerhalb einer Transaktion nicht transaktional auszuführen. Programmabschnitte, die als *escape actions* gekennzeichnet sind, werden von der Transaktion ausgeführt, haben aber keinen Einfluss auf die *conflict detection* oder das *version management*. Sie sind dafür gedacht, in Abwesenheit von *irrevocable actions* oder als deren Ergänzung, die Benutzung von *legacy librarys* und Systemfunktionen zu ermöglichen. Die in Kapitel 3 und 4 beschriebenen TM-Systeme unterstützen keine *escape actions*, da die zugrundeliegenden *best effort* HTMs diese nicht unterstützen.

2.11 Validation

Für die Validierung des *read sets* bei der *conflict detection* gibt es mehrere Ansätze. Bei der Validierung über den Wert, *validation by value*, speichert das *readset* neben der Adresse den Wert zum Zeitpunkt des Lesens und dieser wird zum Zeitpunkt der Validierung gegen den Wert im Speicher geprüft. Es kann aber auch eine Validierung über Metadaten stattfinden, wie zum Beispiel eine Versionsnummer. Hierbei enthält das *readset* dann nicht den Wert, sondern die Metainformationen. Gemein haben diese beiden Vorgehensweisen, dass die Größe des *readsets* proportional zu der Anzahl der gelesenen Werte wächst. Deshalb gib es auch noch den Ansatz, für das *read-* und *writeset* Bloomfilter zu benutzen. Für die neuen oder alten zuschreibenden Werte wird trotzdem noch ein Speicher benötigt, um die Werte beim *commit* zu veröffentlichen oder beim *abort* zurück zuschreiben. Bloomfilter haben eine *bit array* konstanter Größe und arbeiten mit Signaturen aus Hashfunktionen, um die Mitgliedschaft eines Elements in einer Menge festzustellen. Aufgrund der Hashfunktionen kann es *false positives* geben, d. h., ein Element wird fälschlicherweise als Mitglied der Menge erkannt. *False negatives* gibt es aber nicht. Ein Bloomfilter bietet die Funktionen, Elemente dem Filter hinzuzufügen und auf Mitgliedschaft eines Elementes in dem Filter zu testen. Diese Operationen können jeweils in konstanter Zeit, $O(1)$, ausgeführt werden. Als Beispiel seien ein *bit array* der Länge n und k Hashfunktionen gegeben. Jede dieser Hashfunktionen muss ein Eingabeelement e auf ein anderes Bit des *array*s abbilden. Wird ein Element dem Filter hinzugefügt, so wird jedes Bit in dem *array* gesetzt, das ein Ergebnis der Hashfunktionen war. Soll auf die Mitgliedschaft eines Elements in dem Filter getestet werden, so werden die Hashfunktionen auf das Element angewendet und mit dem Filter verglichen. Nur wenn alle k-Bits, die als Ergebnis der Hashfunktionen rauskamen, in dem Filter gesetzt sind, dann ist das Element wahrscheinlich in der Menge. Es kann hierbei zu einem *false positive*

kommen, da die Bits auch von mehreren anderen Elementen stammen könnten. Die Möglichkeit, ein Element wieder aus einem Bloomfilter zu entfernen gibt es nicht, da es sonst *false negatives* gäbe. Der Schnitt und die Vereinigung von zwei Bloomfiltern ist durch das bitweise *AND* und *OR* auch in konstanter Zeit realisierbar. Bei einem TM-System mit *eager conflict detection* wird die Funktion zum Testen der Mitgliedschaft für die Erkennung von Konflikten benutzt. Möchte eine Transaktion an einer Addresse lesen, dann wird überprüft, ob diese Adresse in einem der *writeset*-Bloomfilter der anderen laufenden Transaktionen ist. Ist dies der Fall, kommt es zu einem Konflikt. *False positves* sind hierbei nicht ausgeschlossen. Wird keine Mitgliedschaft festgestellt, so kann die Adresse dem eigenen *readset*-Bloomfilter hinzugefügt und der Wert an der Adresse gelesen werden. Bei einem Schreibvorgang läuft es umgekehrt. Wird zusätzlich noch *eager version management* verwendet, dann wird bei einem Schreibvorgang die Adresse zusätzlich zu den *readset*-Bloomfiltern der anderen laufenden Transaktionen auch gegen deren *writeset*-Bloomfilter geprüft. Wird *lazy conflict detection* benutzt, so kann die Mitglidschaftsprüfung nicht zur Konflikterkennung genutzt werden, da zur *commit*-Zeit nur ein Bloomfilter der gelesenen Werte vorliegt und nicht die einzelnen Adressen. Hier kommt der Schnitt von Bloomfiltern zum Einsatz. Ist der Schnitt zweier Bloomfilter leer oder hat weniger Bits gesetzt, als die Anzahl der Hashfunktionen, so haben die ursprünglichen Mengen auch keine Überschneidung. Invyswell[CGS+14] nutzt diese Überprüfung der Schnittmengen zur Validierung vor dem *commit*, genauso wie, um nach dem *commit* einer Transaktion T_1 alle laufenden Transaktionen zu invalidieren, deren *read sets* eine Überschneidung mit dem *write set* von T_1 haben.

3 Invyswell: HyTM für Haswell

Invyswell[CGS+14] ist ein HyTM-System. Es erweitert Intels RTM (*Restricted Transactional Memory*) mit einer STM, die von InvalSTM[GVS10] abgeleitet ist, zu einem vollständigen TM-System. Invyswell gewährleistet *forward progress* und *opacity*. Im Folgenden wird kurz Intels RTM beschrieben und danach der Aufbau und die Funktionsweise von Invyswell erläutert.

3.1 RTM

RTM ist ein *best effort* HTM aus der TSX-Erweiterung[Int12], die mit Intels Haswell-CPU-Generation eingeführt wurde. Allerdings ist aufgrund eines Bugs die Erweiterung durch einen Patch in den Haswell-CPUs deaktiviert. Invyswell umgeht diesen Bug, um einen korrekten Ablauf des TM-System zu gewährleisten.
Soll ein Programmabschnitt transaktional ausgeführt werden, so muss er mit *XBEGIN* und *XEND* eingeklammert werden. Zusätzlich gibt es die Befehle *XABORT*, um eine Transaktion durch den Programmierer abzubrechen, und *XTEST*, um zu testen, ob der Prozessor eine Transaktion ausführt. Aufgrund der *best effort* Eigenschaft von RTM muss zu jeder Transaktion auch ein nicht-transaktionaler Programmabschnitt angegeben werden. Dieser wird ausgeführt, falls die Transaktion aus irgendeinem Grund abbricht, z. B. auf Grund eines Konflikts oder durch Ressourcenbeschränkungen. Innerhalb des Programmabschnitts, der als Transaktion ausgeführt werden soll, gibt es keine Beschänkungen bezüglich der Befehle, die verwendet werden dürfen. Einige Befehle, wie zum Beispiel I/O, garantieren aber, dass die Transaktion abbricht und der dazugehörige nicht-transaktionale Abschnitt ausgeführt wird. Anhand eines Fehlercodes kann festgestellt werden, warum eine Transaktion nicht erfolgreich ausgeführt wurde. Die Verschachtelung von Transaktionen (*nesting*) ist auch erlaubt und wird durch *flattening* realisiert. Es gibt aber eine von der Implementierung abhängige Verschachtelungstiefe, *MAX_RTM_NEST_COUNT*, bei deren Überschreitung die Transaktion abbricht.
Zu jeder Transaktion im RTM-System gehört ein eigenes *read* und *write set*. In letzterem werden die von der Transaktion getätigten Schreibvorgänge spekulativ gespeichert, um sie beim *commit* zu veröffentlichen (*lazy version management*). Beide *sets* zusammen werden für die Konflikterkennung eingesetzt. Diese erkennt Konflikte zwischen Transaktionen während deren Laufzeit und nicht erst beim *commit* (*eager conflict detection*). Die Konflikte werden mit einer Granularität von *cache lines* erkannt. RTM erfüllt die *strong-atomicity*-Eigenschaft, da auch Konflikte zwischen Transaktionen und nicht-transaktionalen Threads erkannt werden. Kommt es zu so einem Konflikt, wird die betroffene Transaktion abgebrochen. Auch *opacity* erfüllt RTM, da eine Transaktion

sofort abbricht, sobald sich ein Wert ändert, dessen Adresse sich im *read set* befindet.

3.2 Invyswell

Invyswell ist ein HyTM-System, das mehrere Hardware- und Softwaretransaktionen nebenläufig ausführen kann. Transaktionen sollen, wenn möglich, in Hardware ausgeführt werden. Hardwaretransaktionen sind schneller, da sie ohne oder mit wenig zusätzlicher Instrumentarisierung der *read*- und *write*-Befehle auskommen. Es können aber nicht alle Transaktionen in Hardware ausgeführt werden. Daher werden Transaktionen, deren Ausführung in Hardware scheitern, als Softwaretransaktionen ausgeführt.
Invyswell bietet fünf verschiedenen Typen, eine Transaktion auszuführen. Zwei davon werden in Hardware und die anderen drei in Software ausgeführt. Die beiden Hardwarevarianten sind: LiteHW, eine einfache Hardwaretransaktion, und BFHW, eine Bloomfilter basierte Hardwaretransaktion. Für die Ausführung in Software gibt es SpecSW, eine spekulativ ausgeführte Softwaretransaktion, IrrevocSW, eine nicht abbrechbare Softwaretransaktion, und SglSW, eine Softwaretransaktion mit nur einem globalen *lock*. Eine *scheduling strategy* entscheidet, welcher der fünf Typen für die Ausführung der Transaktion ausgewählt wird. Sie entscheidet auch, zu welchem Typ eine Transaktion wechselt, wenn es Probleme gibt oder ein Grenzwert an Wiederholungen überschritten wird. Abbildung 1 zeigt die Übergänge zwischen den einzelnen Transaktionstypen. Eine Transaktion wird zuerst als Hardwaretransaktion ausgeführt, entweder als LiteHW, wenn keine weiteren Softwaretransaktionen laufen, oder als BFHW, wenn es parallele Ausführungen von Softwaretransaktionen gibt. Wenn die Ausführung als Hardwaretransaktion scheitert, dann entscheidet der von RTM zurückgegebene Fehlercode über das weitere Vorgehen. Gibt der Fehlercode an, dass eine erfolgreiche Ausführung unwahrscheinlich ist, so wird die Transaktion als Softwaretransaktion ausgeführt. Ansonsten wird sie als Hardwaretransaktion wiederholt, sofern der Grenzwert für Wiederholungen nicht überschritten ist. Eine Transaktion, die in Software ausgeführt werden soll, kann als SpecSW oder SglSW ausgeführt werden. Kurze Transaktionen, die auf Grund von nicht unterstützten Befehlen als Hardwaretransaktion gescheitert sind, werden als SglSW ausgeführt. Alle anderen Transaktionen werden als SpecSW ausgeführt. Wird eine SpecSW-Transaktion abgebrochen, wird sie entweder wieder als SpecSW oder als IrrevocSW ausgeführt. Durch SglSWs und IrrevocSWs ist *forward progress* gewährleistet, da solche Transaktionen nicht abgebrochen werden können und immer erfolgreich sind.
Invyswell erfüllt nur die *weak atomicity*-Eigenschaft, da Softwaretransaktionen nicht mitbekommen, wenn nicht-transaktionale Ausführungen die Daten im Speicher verändern. Das in der Abbildung 1 zu sehende *fail-fast* ist eine Optimierung. Sie wird bei hoher *contention* aktiv, wenn zu viele Transaktionen abbrechen und hierdurch viel Rechenzeit verloren geht. Ist dies der Fall, werden nur noch LiteHW und SglSW verwendet.

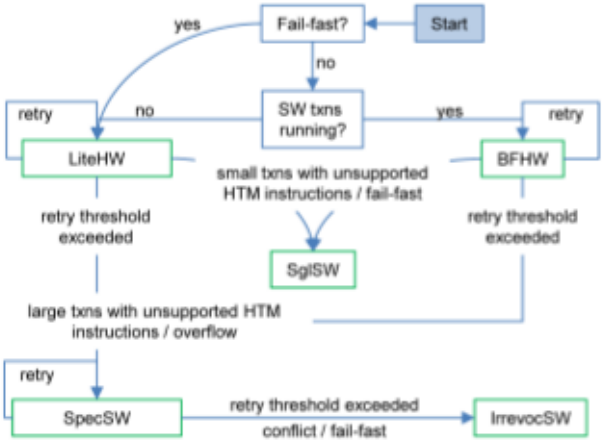

Abbildung 1: Invyswell's State Machine, die die Übergänge zwischen den Transaktionstypen beschreibt. Grafik stammt aus [CGS+14].

3.3 SpecSW

SpecSWs sind die typischen Softwaretransaktionen in Invyswell. Sie besitzen zwei Bloomfilter für *read-* und *writeset* und eine *hashtable* zum Speichern der spekulativen *write*-Werte. Zur *commit*-Zeit werden die spekulativen Werte aus der *hashtable* veröffentlicht und alle anderen Transaktionen, mit denen es einen Konflikt gibt, invalidiert. SpecSW-Transaktionen prüfen daher vor jedem *read*, *write* und *commit*, ob sie invalidiert wurden, und brechen ab, falls dies der Fall ist. Die Invalidierung findet durch das Setzen eines *invalid flags* statt. Dieses *flag* wird bei allen Transaktionen gesetzt, deren *read set* einen nicht leeren Schnitt mit dem *write set* der *commit*-Transaktion hat.
Damit ein STM-System korrekt arbeiten kann, müssen der Beginn und das Ende einer Transaktion und das Lesen und Schreiben von Werten mit extra Befehlen ausgestattet werden. Im Folgenden werden die Ereignisse zum Beginn einer Transaktion, beim Lesen und Schreiben von Werten und beim *commit* beschrieben.
Zu Beginn einer Transaktion wird der *sw_cnt*-Zähler inkrementiert. Er gibt an, wie viele SpecSW-Transaktionen momentan gestartet sind. Diese Information wird von LiteHW-Transaktionen benötigt, da diese keinen *commit* machen dürfen, wenn Softaretransaktionen laufen. Zusätzlich wird mit einem *spin-lock* geprüft, ob die *commit_sequenze* gerade ist. Denn, ist die *commit_sequenze* ungerade, so läuft eine SglSW-Transaktion, und die SpecSW-Transaktion darf nicht starten. Ist die *commit_sequenze* gerade, dann wird sie für spätere Validierungen zwischengespeichert und die Transaktion darf anfangen.
Möchte eine Transaktion einen Wert lesen, so wird zuerst in der *hashtable* überprüft, ob der Wert vorher schon geschrieben wurde. Ist dies der Fall, dann wird der aktuelle Wert aus der *hastable* genommen. Befindet sich der Wert nicht in der *hastable*, dann wird die Adresse dem Bloomfilter für das *read set* hinzugefügt. Danach wird der Wert an der Adresse geladen. Genutzt werden darf der Wert aber erst, nachdem eine Validation stattgefunden hat. Diese Validation ist notwendig, um *opacity* zu gewährleisten,

da ein Zeitfenster existiert, in dem inkonsistente Daten gelesen werden könnten. Dieses Zeitfenster entsteht, da beim *commit* die Invalidierung erst nach dem Veröffentlichen der *hashtable* stattfindet. Näheres zur Validation wird weiter unten beschrieben.
Soll ein Wert von einer Transaktion geschrieben werden, so wird zuerst geprüft, ob die Transaktion invaliert wurde. Ist dies nicht der Fall, wird die Adresse dem Bloomfilter für das *write set* und der zu schreibende Wert der *hashtable* hinzugefügt.
Kommt eine Transaktion am Ende ihrer Ausführung an, versucht sie einen *commit* zu machen. Handelt es sich um eine *read-only*-Transaktion, wird nur der *sw_cnt*-Zähler dekrementiert, womit die Transaktion abgeschlossen ist. Ansonsten versucht die Transaktion das *commit_lock* zu akquerieren. Bekommt die Transaktion das *commit_lock*, findet eine Validation statt. Ist diese erfolgreich, wird der *contention manager* (CM) um *commit*-Erlaubnis gefragt. Als *contention manager*-Strategie wird iBalanced[GHPS12] benutzt, die Prioritäten, Größe des *read* und *write set*s und andere Faktoren zur Entscheidungsfindung benutzt. Erteilt der CM die Erlaubnis, wird der *sw_cnt*-Zähler dekrementiert und die spekulativen Werte aus der *hashtable* werden veröffentlicht. Abschließend findet die Invalidierung statt und das *comit_lock* wird wieder freigegeben.
Die oben erwähnte Validierung beim Lesen und vor dem *commit* besteht aus vier Schritten. Der erste Schritt ist, zu prüfen, ob sich die *commit_sequenze* seit dem Start verändert hat, da eine Änderung der *commit_sequenze* bedeutet, dass eine SglSW am Laufen ist oder war. Da SglSW-Transaktionen kein *read* und *write set* führen, kann keine *conflict detection* stattfinden. SpecSW-Transaktionen müssen daher neustarten, falls sich die *commit_sequenze* geändert hat. Der zweite Schritt ist eine Konfliktprüfung mit der Transaktion, die grade das *commit_lock* hält. Kommt es zu einem Konflikt, muss die validierende Transaktion neugestartet werden. Ein Abbruch der Transaktion, die das *commit_lock* besitzt, ist nicht möglich, da es sich um eine IrrevocSW handeln könnte. Als drittes muss gewartet werden, dass der *hw_post_commit*-Zähler auf Null ist. Denn, ist dieser ungleich Null, dann gibt es noch BFHW-Transaktionen, die noch nicht mit ihrer Invalidierung fertig sind. Auf das Ende der Invalidierungen durch BFHW-Transaktionen muss gewartet werden, da sonst Konflikte übersehen werden könnten. Denn Schritt Zwei erkennt nur Konflikte mit Softwaretransaktionen, da Hardwaretransaktionen das *commit_lock* nicht akquerieren. Als vierter und letzter Schritt der Validierung wird geprüft, ob die Transaktion invalidiert wurde. Ist dies der Fall, muss die Transaktion neustarten. Ansonsten war die Validation erfolgreich und die Transaktion darf das Lesen oder den *commit* fortsetzen.

3.4 BFHW

BFHW-Transaktionen sind Hardwaretransaktionen, die RTM nutzen aber zusätzlich mit Bloomfiltern für *read-* und *writeset* versehen sind. Die Bloomfilter dienen zur Konflikterkennung mit Softwaretransaktionen und zu deren Invalidierung. Hardwaretransaktionen werden nie invalidiert, da RTMs strenge Isolationseigenschaft gewährleistet, dass eine Hardwaretransaktion abbricht, sobald ein von ihr gelesener oder spekulativ geschriebener Wert verändert wird.

Der Start einer BFHW-Transaktion ist ohne Veränderung einfach nur der *XBEGIN*-Befehl von RTM. Jedes Lesen oder Schreiben wird mit einem zusätzlichen Befehl versehen, der die Adresse dem jeweiligen Bloomfilter hinzufügt. Beim *commit* kommen ein paar mehr Aktionen zu dem *XEND*-Befehl von RTM hinzu. Die erste Aktion ist, das *commit_lock* zuprüfen. Es wird hierbei nicht akqueriert, sondern nur gelesen und damit dem Hardware-*readset* hinzugefügt. Ist das *commit_lock* frei, kann der *commit* vollzogen werden. Hierzu wird der *hw_post_commit*-Zähler inkrementiert und danach die spekulativen Werte aus dem Hardware-*write-set* mit dem *XEND*-Befehl veröffentlicht. Sollte es zwischen dem Inkrementieren und dem Veröffentlichen zu einem Konflikt kommen, muss der *hw_post_commit*-Zähler nicht dekrementiert werden, da er Teil des spekulativen *write set*s ist. War das *commit_lock* nicht frei, muss eine Konflikterkennung mit der Transaktion, die das *lock* hält, stattfinden. Bei einem Konflikt bricht die BFHW-Transaktion durch ein *XABORT* ab. Ansonsten wird, wie oben, der Zähler erhöht und das *write set* veröffentlicht. Abschließend findet die Invalidierung der Softwaretransaktionen statt und der *hw_post_commit*-Zähler wird wieder dekrementiert.
Der *hw_post_commit*-Zähler ist wichtig für die SpecSW-Transaktionen, um die *opacity*-Eigenschaft einzuhalten. Ansonsten könnte eine SpecSW-Transaktion einen inkonsistenten Datenbestand vorfinden, wenn die BFHW mit der Veröffentlichung des *write set*s fertig ist, aber noch nicht invalidiert hat.
Weiter ist noch zu erwähnen, das SpecSWs immer erst das *commit_lock* freigeben und dann erst ihr *read* und *write set* löschen. Hierduch brechen alle BFHWs ab, die das besetzte *commit_lock* gelesen haben. Würden die SpecSWs erst ihre *sets* löschen, bevor sie das *commit_lock* freigäben, könnte ein Konflikt unerkannt bleiben. Denn eine BFHW, die ein besetztes *commit_lock* vorgefunden hat, könnte einen Konflikt bei der Konflikterkennung übersehen, da die *sets* schon gelöscht wurden.

3.5 LiteHW

LiteHW-Transaktionen sind der zweite Typ von Hardwaretransaktionen in Invyswell. Es handelt sich bei ihnen um einfache RTM-Transaktionen mit einer Einschränkung beim *commit*. LiteHWs haben, im Gegensatz zu BFHWs, keine Bloomfilter für das *read* und *write set*. Sie sind dadurch schneller als BFHWs, da es keinen zusätzlichen *overhead* für die Bloomfilter gibt. Der Nachteil der fehlenden Bloomfilter ist aber, dass keine Konflikterkennung mit Softwaretransaktionen, bzw. deren Invalidierung, stattfinden kann. Dies führt zu der Einschränkung beim *commit*. Denn LiteHWs dürfen keinen *commit* machen, wenn Softwaretransaktionen im System laufen. Um dies zu gewährleisten, prüft eine LiteHW, bevor sie den *XEND*-Befehl ausführt, ob das *commit_lock* frei und der *sw_cnt*-Zähler gleich Null ist. Die Prüfug des *sw_cnt*-Zählers allein genügt nicht, da dieser nur die Anzahl der laufenden SpecSW-Transaktionen angibt. Um zu testen, ob eine der anderen beiden Softwaretransaktionstypen läuft, muss das *commit_lock* betrachtet werden, da beide dieses zu Beginn ihrer Ausführung einholen. Läuft keine Softwaretransaktion, darf die LiteHW den *commit* machen, ansonsten muss sie abbrechen.

3.6 IrrevocSW

IrrevocSW-Transaktionen sind Softwaretransaktionen, die nicht abgebrochen werden können. Sie werden benötigt, damit Invyswell die *forward progress*-Eigenschaft erfüllt. Denn sowohl SpecSWs als auch die beiden Hardwaretransaktionstypen gewährleisten nicht, dass eine Transaktion jemals erfolgreich abgeschlossen werden kann. Wird eine Transaktion zu häufig wiederholt und erreicht damit einen Grenzwert, wird sie zu einer IrrevocSW-Transaktion befördert. Auch Transaktionen, die Befehle enthalten, die nicht wiederholt werden können, wie z.B I/O, müssen als nicht-abbrechbare Transaktion ausgeführt werden. IrrevocSWs führen Bloomfilter für *read* und *write set*, um SpecSWs invalidieren zu können und eine Konflikterkennung mit BFHWs zu ermöglichen.
Zu Beginn einer IrrevocSW holt sich diese das *commit_lock*. Hiermit ist sichergestellt, dass keine anderen Transaktionen einen *commit* machen können, mit der Ausnahme von BFHWs, die bei einem vergebenen *commit_lock* Konflikterkennung betreiben. Das Lesen und Schreiben von Werten ist genauso wie bei einer nichttransaktionalen Ausführung, mit der Ausnahme, dass die Adressen dem jeweiligen Bloomfilter hinzugefügt werden. Änderungen finden direkt im Speicher statt und werden nicht spekulativ zwischengespeichert. Die *opacity*-Eigenschaft von SpecSWs ist hierdurch nicht gefährdet, da SpecSWs bei jedem Lesen eine Konflikterkennung mit der Transaktion machen, die das *commit_lock* hält. Kommt die IrrevocSW zum Ende ihrer Ausführung, besteht der *commit* daraus, dass eine Invalidierung über die Bloomfilter stattfindet und das *commit_lock* wieder freigegeben wird.
Nebenläufig zu einer IrrevocSW können SpecSWs, BFHWs und LiteHWs ausgeführt werden. Einen *commit* dürfen während der Ausführung einer IrrevocSW aber nur BFHWs machen.

3.7 SglSW

SglSW-Transaktionen sind auch Softwaretransaktionen, die nicht abgebrochen werden können. Sie sind leichtgewichtiger als IrrevocSW-Transaktionen, da sie keinen *overhead* durch Bloomfilter haben. Der Nachteil der fehlenden Bloomfilter ist, dass keine SpecSWs nebenläufig ausgeführt werden können und keine Transaktion einen *commit* machen darf, während einen SglSW ausgeführt wird. BFHWs und LiteHWs können aber nebenläufig ausgeführt werden. SglSWs eignen sich aufgrund dieser Ausführungs- und *commit*-Beschränkungen für kurze Transaktionen, die nicht in Hardware ausgeführt werden können.
Wenn eine SglSW-Transaktion beginnt, bewirbt sie sich um das *commit_lock*. Hat sie dieses bekommen, erhöht sie die *commit_sequenze* auf einen ungeraden Wert. Dieses beendet alle laufenden SpecSWs und hindert neue daran zu starten. Denn ohne die Bloomfilter können SpecSWs keine Konflikterkennung machen und auch nicht von der SglSW invalidiert werden. Die Ausführung des Inhalts der Transaktion ist wie bei einer nichttransaktionalen Ausführung, alle Änderungen finden direkt im Speicher statt. Zum Abschluss einer SglSW wird die *commit_sequenze* wieder auf einen geraden Wert erhöht

Typen	BFHW	LiteHW	SpecSW	IrrevocSW	SglSW
BFHW	ja	ja	ja	ja	ja
LiteHW	ja	ja	ja	ja	ja
SpecSW	ja	ja	ja	ja	nein
IrrevocSW	ja	ja	ja	nein	nein
SglSW	ja	ja	nein	nein	nein

Tabelle 1: Nebenläufige Ausführungsmatrix

und anschließend das *commit_lock* freigegeben.

3.8 Nebenläufige Ausführung und nebenläufiger commit

Tabelle 1 zeigt, welche Transaktionstypen nebenläufig ausgeführt werden können. Dies ist nicht zu verwechseln mit einem nebenläufigen *commit*. Zum Beispiel können LiteHW-Transaktionen nebenläufig zu allen Transaktionstypen ausgeführt werden, sie können aber nur nebenläufig zu LiteHWs und BFHWs einen *commit* machen. Generell können nur BFHW & BFHW, BFHW & LiteHW, BFHW & SpecSW, BFHW & IrrevocSW und LiteHW & LiteHW nebenläufig einen *commit* machen.
BFHW-Transaktionen können somit nebenläufig zu allen Transaktionstypen ausgeführt werden. Der Grund hierfür ist die strenge Isolationseigenschaft von RTM. Die Isolationseigenschaft gewährleistet, dass eine BFHW-Transaktion abgebrochen wird, sobald es eine Änderung im Speicher gibt, die die BFHW betrifft. Dies gewährleistet auch *opacity*. Einen *commit* können BFHWs nebenläufig zu BFHWs, LiteHWs, SpecSWs oder einer IrrevocSW machen. Für den nebenläufigen *commit* von BFHWs und LiteHWs sorgt RTMs Konflikterkennung. Bei SpecSWs oder einer IrrevocSW werden die Bloomfilter zur Erkennung von Konflikten benutzt. Existiert ein Konflikt, dann bricht die BFHW-Transaktion ab. Mit einer SglSW kann kein nebenläufiger *commit* stattfinden, da die SglSW keine Bloomfilter für die Konflikterkennung hat. Eine BFHW bricht ab, wenn sie die *commit*-Phase betritt und eine SglSW läuft.
LiteHW-Transaktionen können nebenläufig zu allen Transaktionstypen laufen. Dies liegt, wie bei BFHWs, an RTMs Isolationseigenschaft. LiteHWs können nur nebenläufig zu BFHWs und LiteHWs einen *commit* machen. Der Grund dafür ist, dass LiteHWs keine Bloomfilter besitzen. Diese werden aber zur Konflikterkennung mit Softwaretransaktionen benötigt. LiteHWs müssen daher abbrechen, wenn sie die *commit*-Phase erreichen und eine Softwaretransaktion am Laufen ist.
SpecSWs können nebenläufig zu allen Transaktionstypen, mit Ausnahme von SglSW, ausgeführt werden. Die Ausführung von SpecSWs neben LiteHWs ist möglich, da letztere keinen *commit* machen dürfen, solange Softwaretransaktionen laufen. Somit können LiteHWs die Ausführung von SpecSWs nicht stören. SpecSWs können nebenläufig zu BFHWs, SpecSWs oder einer IrrevocSW ausgeführt werden, da alle diese Transaktionstypen Bloomfilter zur Konflikterkennung besitzen. Wenn eine BFHW, SpecSW oder

IrrevocSW einen *commit* macht, dann werden alle SpecSW-Transaktionen mit denen es einen Konflikt gibt invalidiert. Um hierbei *opacity* für die SpecSW, die abgebrochen werden soll, zu gewährleisten, muss die in Kapitel 3.3 beschriebene Validation beim Lesen durchgeführt werden. Eine SpecSW-Transaktion kann nur nebenläufig zu BFHW-Transaktionen einen *commit* machen. Denn Softwaretransaktionen können keinen nebenläufigen *commit* machen, da immer nur eine Transaktion das *commit_lock* halten kann.
Neben einer IrrevocSW-Transaktion können BFHWs, LiteHWs oder SpecSWs ausgeführt werden. Dies liegt daran, dass eine IrrevocSW das *commit_lock* zu Beginn ihrer Ausführung akqueriert. Wenn das *commit_lock* vergeben ist, betreiben BFHWs eine Konflikterkennung vor dem *commit* und SpecSWs und LiteHWs dürfen keinen *commit* ausführen. Hierdurch ist garantiert, dass die Ausführung der IrrevocSW nicht gestört wird. Eine nebenläufige Ausführung einer anderen IrrevocSW oder SglSW ist nicht möglich, da immer nur eine unabbrechbare Transaktion im System laufen darf.
Neben einer SglSW-Transaktion können nur Hardwaretransaktionen ausgeführt werden. Einen *commit* dürfen diese aber nicht machen, da die SglSW davon nichts mitbekommen würde. SpecSW-Transaktionen können während der Ausführung einer SglSW nicht gestartet werden bzw. müssen abgebrochen werden, da keine Konflikterkennung mit der SglSW stattfinden kann.

3.9 TSX erratum

Die TSX-Erweiterung in den Haswell-Prozessoren ist mit einem Fehler behaftet. Unter sehr seltenen Konstellationen kann es dazu kommen, dass die Adresse eines indirekten Sprungs mit einem *XEND* oder Daten die, wie ein *XEND* aussehen, überschrieben wird. Dies führt zu einem verfrühten *commit*, ohne dass das *commit_lock* geprüft wird. Um dieses zu verhindern, haben die Entwickler von Invyswell vor jedem indirekten Sprung eine Überprüfung des *commit_locks* eingebaut. Dieses haben sie per Hand gemacht, da es keine Kompilerunterstützung dafür gibt.

3.10 Implementierung

Calciu et al.[CGS+14] haben Invyswell auf einem Core i7-4770 mit der STAMP Benchmark-Suite getestet. Die Auswahl der unterschiedlichen Codepfade für die Transaktionstypen wurde mit Makros aus der Benchmark-Suite implementiert. Die Bloomfilter waren 1024 Bits lang und verwendeten die *spooky-hash*-Funktion[Jen10]. Hardwaretransaktionen wurden zehn mal wiederholt, bevor zu einer Softwarevariante gewechselt wurde. SpecSW-Transaktionen wurden maximal 4 mal wiederholt, bevor sie zu SglSW-Transaktionen erhoben wurden. IrrevocSWs wurden nicht benutzt, da sie bei der Haswell-TSX-Implementierung weniger Durchsatz geliefert haben als SglSWs. In zukünftigen CPUs, die mehr Hardwaretransaktionen ausführen, könnten sich IrrevocSWs aber lohnen. Auch eine Kompilerintegration sollte sich positiv auf den Durchsatz auswirken.

4 TM für BlueGene/Q

BlueGene/Q-TM ist ein Hybrid-TM. Das System erweitert den *best effort* HTM des BlueGene/Q-Prozessors um Softwaretransaktionen, sodass ein vollwertiges TM-System entsteht. Das Software-System greift hierzu in den Kernel, den Kompiler und das Laufzeitsystem ein, um die Benutzung des TM-Programmiermodells einfach zu gestallten. Das Pragma *#pragma tm_atomic* {} umschließt einen Programmbereich, der als Transaktion ausgeführt werden soll. Das BlueGene/Q-TM-System versucht möglichst alle Transaktionen als Hardwaretransaktionen abzuwickeln, da diese nebenläufig ausgeführt werden können. Die restlichen Transaktionen, die nicht in Hardware abgeschlossen werden können oder zu häufig dort gescheitert sind, werden im *irrevocable mode* in der Software ausgeführt. Da *irrevocable transactions* nicht abgebrochen und wiederholt werden können, kann immer nur eine Transaktion zur Zeit ausgeführt werden. Er dürfen aber Hardwaretransaktionen nebenläufig zur Softwaretransaktion ausgeführt werden. Kommt es zu einem Konflikt zwischen der Software- und einer Hardwaretransaktion, bricht die Hardwaretransaktion ab.

4.1 BlueGene/Q HTM

BG/Q-HTM ist die *best effort* HTM-Unterstützung des BlueGene/Q-Prozessors. Ein Rechenchip des BG/Q hat 16 Prozessorkerne, die jeweils 4 Hardware-SMT-Threads (*Simultaneous-Multi-Threaded*) ausführen können. Jeder Rechenkern hat einen 16KByte großen L1-Cache, den sich die 4 SMT-Threads teilen. Die Größe der *cache line* beträgt hier 64 Bytes. Den 32MByte großen L2-Cache mit seinen 128 Byte langen *cache lines* teilen sich alle 16 Kerne.
Bei BG/Q-HTM kommt *lazy version managrment* zum Einsatz. Die Hardwaretransaktionen speichern ihre Werte spekulativ im L2-Cache, um sie bei einem *commit* zu veröffentlichen. Der L2-Cache ist 16fach *set-associative* und in der Lage, mehrere Versionen derselben physikalischen *cache line* zu halten.
Eine Buchführung über die Zugriffe auf den L2-Cache dient zur Konflikterkennung. Bei den Zugriffen wird gespeichert, ob es sich um lesende oder schreibende handelt, und, ob der Zugriff spekulativ ist. Handelt es sich um einen spekulativen Zugriff, dann wird zusätzlich noch die spekulative ID des Transaktionsthreads vermerkt. Von diesen einzigartigen spekulativen IDs gibt es 128 Stück. Sie verknüpfen die ThreadID der Transaktion mit den Speicherzugriffen im L2-Cache. Sind keine IDs mehr frei, müssen neue Transaktionen warten, bis wieder IDs zur Verfügung stehen. Ein Vorgang namens *scrubbing* sucht in einem festen Intervall (132 Cycles) den L2-Cache nach nicht mehr benötigten IDs ab und gibt diese frei. Der Intervall ist im Laufzeitsystem anpassbar.

Ein Konflikt wird von der Hardware zwischen Transaktionen erkannt, wenn es einen *read-write*, *write-read* oder *write-write* Zugriff von zwei unterschiedlichen Transaktionen gibt. Zu einem Konflikt mit einer nicht-transaktionalen Ausführung kommt es, wenn diese an einer Stelle schreibt, an der es einen vorherigen transaktionalen Zugriff gab. Kommt es zu einem Konflikt, wird ein *interrrupt* an alle betroffenen transaktionalen Threads geschickt. In einem zusätzlichen *speculative conflict register* werden Hardwareereignisse vermerkt, die die Transaktion zum Scheitern bringen können.
BG/Q bietet zwei Ausführungsmodi, um Transaktionen auszuführen. Es gibt den *short-running mode* und den *long-running mode*. Als Standard wird der *long-running mode* verwendet. Dies kann aber durch das Setzen einer Umgebungsvariable vor dem Programmstart geändert werden. Der große Unterschied zwischen den beiden Modi ist, wie sie spekulative Werte im L1-Cache von den anderen SMT-Threads geheim halten.
Der *short-running mode* umgeht den L1-Cache. Soll etwas spekulativ gespeichert werden, so wird es direkt in den L2-Cache geschrieben. Wird etwas geladen, dass zuvor spekulativ geschrieben wurde, dann wird es aus dem L2-Cache direkt in die Register des Threads geladen. Beim transaktionalen Laden von Daten, die sich schon im L1-Cache befinden, wird der L2-Cache mit einer L1-Notifikation über diesen Vorgang informiert.
Der *long-running mode* nutzt den L1-Cache, muss ihn aber zu Beginn der transaktionalen Ausführung invalidieren. Der L1-Cache kann durch TLB-Aliasing (*Translation Lookaside Buffer*) bis zu 5 verschiedene Versionen (vier spekulative und eine nichttransaktionale) derselben *cache line* speichern. Die *Memory Management Unit* verwendet einige Adressbits, um die unterschiedlichen Versionen vorzuhalten. Beim Übergang der Daten in den L2-Cache wird diese Alias-Illusion aufgehoben, da der L2-Cache sein eigenes Buchführungssystem hat. Die Invalidierung des L1-Caches beim Betreten des transaktionalen Bereichs ist notwendig, damit die L2-Buchführung über jegliche Zugriffe informiert wird. Bei jedem neuen Zugriff kommt es nun zu L1-*load-misses*, durch die der L2-Cache informiert wird.
Beide Modi haben jeweils einen Nachteil, wodurch sie sich für unterschiedliche Transaktionstypen eignen. Der *short-running mode* verliert den Vorteil der schnellen L1-Cache-Anbindung für *read-after-write* Zugriffsmuster und ist daher eher für kurz laufende Transaktionen geeignet. Der *long-running mode* verliert durch das Invalidieren den Vorteil der Wiederverwendung von Daten vor und nach der transaktionalen Ausführung. Dieser Modus ist daher besser für lang laufende Transaktionen geeignet.
Die erfolgreiche Ausführung von Transaktionen kann aus drei Gründen scheitern. Erstens durch einen Konflikt, der im L2-Cache entdeckt wird. Die Granularität der Konflikterkennung unterscheidet sich zwischen den beiden Ausführungsmodi. Bei dem *short-running mode* beträgt die Granularität 8 Bytes, wenn maximal zwei Threads auf dieselbe *cache line* zugriefen, und sonst 64 Byte. Und beim *long-running mode* beträgt sie immer 64 Byte. Der zweite Grund für das Scheitern ist ein Kapazitätsüberlauf, d. h., wenn das Limit des L2-Caches erreicht ist. Der dritte Grund ist eine *jail mode violation*. Damit Transaktionen keine unumkehrbaren Operationen ausführen, wie z. B. I/O, werden alle Transaktionen in einen *jail mode* gersperrt. Dieser *jail mode* ist eine Art Sandkasten. Versucht eine Transaktion aus diesem Sandkasten auszubrechen, quittiert der Kernel dies mit einem *jail mode interrupt*.

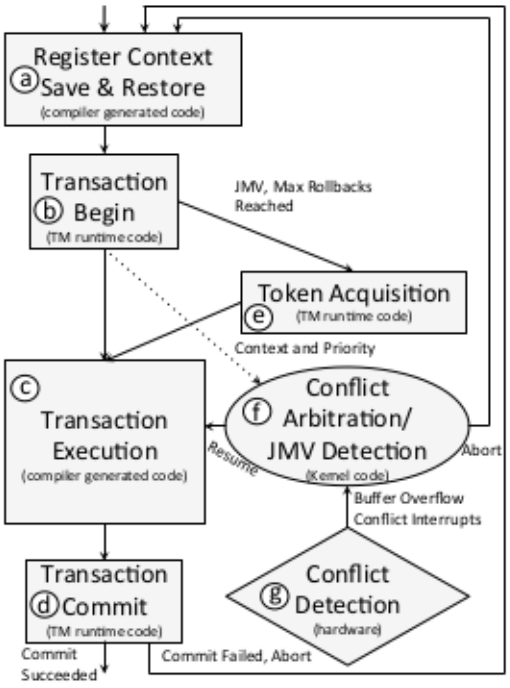

Abbildung 1: BlueGene/Q TM Ausführungsübersicht. Grafik stammt aus [WGW+12].

4.2 BlueGene/Q TM

Die Softwarelösung, die auf das HTM-System aufsetzt, ermöglicht es, Transaktionen beliebiger Länge und mit beliebigem Befehlssatz erfolgreich auszuführen. Das gesamte TM-System besteht aus einer Laufzeitumgebung sowie Erweiterungen am Kernel und Kompiler. Die Laufzeitumgebung regelt sowohl den Start als auch das Ende von Transaktionen und, dass solche, die nicht als Hardwaretransaktionen ausgeführt werden können, als *irrevocable transactions* in Software ausgeführt werden. Auch das Verschachteln von Transaktionen durch *nesting* ist durch das Laufzeitsystem möglich. Es wird durch *flattening* gelöst.
Abbildung 1 zeigt wie die Ausführung im TM-System aussieht. In ⓐ passiert das *checkpointing* und Wiederherstellen des *checkpoints*. Zu Beginn einer transaktionalen Ausführung müssen Register gesichert werden. Denn im Fall eines Abbruchs müssen dies Register wieder zurück geschrieben werden. Dies könnte man durch die Systemfunktionen *setjmp* und *longjmp* erledigen. Der *overhead* dieser beiden Funktionen ist bei kurzen Transaktionen aber groß. Daher generiert der Kompiler für jeden transaktionalen Bereich eigene Befehle zum Sichern und Zurückschreiben der jeweils relevanten Register. Jedes Mal werden der *stack pointer*, der *global offset table pointer* und ein *pointer* zu den Befehlen, die die Register zurückschreiben, gesichert. Die anderen Register, die gesichert werden müssen, bestimmt der Kompiler zur Kompilzeit.
Nach dem *checkpointing* entscheidet das Laufzeitsystem, wie es die Transaktion ausführt (ⓑ). Soll die Transaktion als Hardwaretransaktion ausgeführt werden, schreibt das Lauf-

zeitsystem in einem, vom Speicher verwalteten, I/O-Bereich. Kommt die Transaktion am Ende ihrer Ausführung an und möchte einen *commit* machen, schaltet sich wieder die Laufzeitumgebung in ⓓ ein. Diese versucht den *commit* der Transaktion auszuführen. Scheitert der Versuch, muss die Transaktion neustarten.
Die Konflikterkennung in ⓖ erkennt Konflikte durch Kapazitätsüberschreitungen und Zugriffskonflikte. Bei beiden wird ein *interrupt* an den Kernel gesendet. Dies ist die Standardeinstellung und entspricht einer *eager conflict detection*. Die Laufzeitumgebung kann die Hardware aber auch anweisen *interrupts* durch Zugriffskonflikte zu unterdrücken (*lazy conflict detection*). In diesem Fall ist *opacity* nicht gewährleistet, da Konflikte nur beim *commit* entdeckt werden. Bei einer Kapazitätsüberschreitung wird von der Hardware automatisch die spekulative ID invalidiert. Der hierbei an den Kernel gesendete *interrupt* dient dann nur noch, um die Wiederholung der Transaktion zu veranlassen. Erhällt der Kernel in ⓕ einen *interrupt* durch einen Zugriffskonflikt, entscheidet er anhand des Zeitstempels der Transaktionen, welche Transaktion die jüngere ist, und bricht diese ab.
In ⓕ findet auch die Überwachung der Transaktionen bzgl. *jail mode violations* statt. Im Falle einer *jail mode violation* invalidert der Kernel die Transaktion und startet sie neu.
Um *forward progress* zu gewährleisten, gibt es den *irrevocable mode*. In diesem Modus werden Transaktionen ausgeführt, die *jail mode violations* ausgelöst haben oder zu häufig durch andere Konflikte als Hardwaretransaktion gescheitert sind. Transaktionen, die in diesem Modus ausgeführt werden, werden nicht-spekulativ ausgeführt und können daher nicht rückgäng gemacht werden. Entscheidet das Laufzeitsystem in ⓑ, dass eine Transaktion in den *irrevocable mode* erhoben wird, so muss das Laufzeitsystem für diese Transaktion das *irrevocable token* akquerieren (ⓔ). Dieser *token* ist einmalig für alle *tm_atomic*-Blöcke und wirkt wie ein *lock* für diese. Ist die Transaktion im *irrevocable mode* abgeschlossen worden, dann wird der *token* wieder freigegeben. Danach kann die nächste Transaktion in diesem Modus ausgeführt werden. Eine nebenläufige Ausführung von Hardwaretransaktionen zu einer Transaktion in dem Modus ist möglich. Bei einem Konflikt muss aber immer die Hardwaretransaktion abbrechen.

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)

[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)

[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)

[GHPS12] Gottschlich, J. E. ; Herlihy, M. P. ; Pokam, G. A. ; Siek, J. G.: Visualizing transactional memory. In: *PACT '12 Proceedings of the 21st international conference on Parallel architectures and compilation techniques* (2012)

[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)

[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

[IS05] III, William N. S. ; Scott, Michael L.: Advanced Contention Management for Dynamic Software Transactional Memory. In: *PODC '05 Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing* (2005)

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

[KGH+11] Kestor, Gokcen ; Gioiosa, Roberto ; Harris, Tim ; Unsal, Osman S. ; Cristal, Adrian ; Hur, Ibrahim ; Valero, Mateo: STM^2: A Parallel STM for High Performance Simultaneous Multithreading Systems. In: *PACT '11 Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (2011)

[MBM+06a] 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)

[MBM+06b] Moravan, Michelle J. ; Bobba, Jayaram ; Moore, Kevin E. ; Yen, Luke ; Hill, Mark D. ; Liblit, Ben ; Swift, Michael M. ; Wood, David A.: Supporting Nested Transactional Memory in LogTM. In: *ASPLOS XII Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (2006)

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

[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)