Reduktion von False-Sharing in Software-Transactional-Memory

Stefan Kempf, Ronald Veldema und Michael Philippsen

Department Informatik, Lehrstuhl für Informatik 2 (Programmiersysteme)
Universität Erlangen-Nürnberg, Martensstr. 3, 91058 Erlangen
{stefan.kempf, veldema, philippsen}@cs.fau.de

Abstract: Software-Transactional-Memory (STM) erleichtert das parallele Programmieren, jedoch hat STM noch einen zu hohen Laufzeitaufwand, da gegenseitiger Ausschluss beim Zugriff auf gemeinsame Daten meist mittels einer Lock-Tabelle fester Größe realisiert wird. Für Programme mit wenigen konkurrierenden Zugriffen und überwiegend Lesezugriffen ist diese Tabelle größer als notwendig, so dass beim Commit einer Transaktion mehr Locks zur Konsistenzprüfung zu inspizieren sind als nötig. Für große Datenmengen ist die Tabelle zu klein. Dann begrenzt False-Sharing (unterschiedliche Adressen werden auf das gleiche Lock abgebildet) die Parallelität, da sogar unabhängige Transaktionen sich gegenseitig ausschließen. Diese Arbeit beschreibt eine Technik, die die Lock-Tabelle bei False-Sharing vergrößert. Zusätzlich kann ein Programmierer mit Annotationen unterschiedliche Lock-Tabellen für voneinander unabhängige Daten verlangen, was die Möglichkeit von False-Sharing und den Speicherbedarf für die Locks weiter verringert In Benchmarks erreichen wir einen maximalen Speedup von 10.3 gegenüber TL2, wobei die Lock-Tabelle bis zu 1024 mal kleiner ist.

1 Einleitung

Beim Programmierkonzept Software-Transactional-Memory (STM) markieren Entwickler kritische Abschnitte in parallelem Code als atomare Blöcke, anstatt diese manuell mit Locks zu synchronisieren. Atomare Blöcke werden zur Laufzeit als Transaktionen ausgeführt. Es können keine Deadlocks aufgrund einer falschen Verwendung von Locks auftreten. Das STM übernimmt die Implementierung des gegenseitigen Ausschlusses, typischerweise unter Verwendung einer Lock-Tabelle mit fester Größe, die aber mit wachsenden Datenmengen zum Flaschenhals wird, da die Wahrscheinlichkeit von False-Sharing steigt (verschiedene Adressen werden auf das gleiche Lock abgebildet). Dies verringert die Parallelität, da sich dadurch sogar unabhängige Transaktionen gegenseitig ausschließen. Um dem entgegenzuwirken, stellt diese Arbeit nach einer Diskussion der technischen Grundlagen von STM ein System vor, bei dem die Größe der Lock-Tabelle dynamisch mit der Datenmenge des Programms skaliert. Zudem kann der Programmierer mit Hilfe von Annotationen unabhängige Speicherregionen/Datenstrukturen mit getrennten Lock-Tabellen versehen. Ein Vergleich unseres STMs mit TL2 [DSS06] zeigt, dass dynamische Lock-Tabellen bis zu einem Speedup von 10.3 gegenüber TL2 führen, wobei die Tabelle für einige Benchmarks im Vergleich zu TL2 bis zu 1024 mal kleiner ist.

2 Grundlagen von Software-Transaktionen

Unser STM verwendet das Design von TL2 [DSS06], das hier skizziert wird. Ein globaler Zähler *GC* und ein Array fester Größe von n versionierten Locks bilden die zentralen Datenstrukturen des STM. Eine startende Transaktion T kopiert den aktuellen Wert von *GC* in die für T private Variable *rv*. Ein Schreibzugriff auf eine Variable v vermerkt den zu schreibenden Wert zunächst nur in einem Schreibprotokoll. Am Ende von T (Commit) erwirbt T zuerst Locks für alle zu schreibenden Variablen v (eine Hash-Funktion bildet Adressen/Variablen auf Locks ab). Anschließend wird *GC* atomar inkrementiert und mittels der Einträge aus dem Schreibprotokoll werden alle Variablen v aktualisiert. Abschließend gibt T alle Locks frei, indem sie die Versionen in den Locks auf den neuen Wert von *GC* setzt. Die Lock-Version wird bei einem Lesezugriff verwendet, um Änderungen einer Variable durch konkurrierende Transaktionen aufzudecken. Beim Lesen einer Variable v prüft T zuerst, ob das Schreibprotokoll einen Eintrag für v enthält, und liefert ggf. den Wert aus dem Protokoll zurück. Anderenfalls wird v auf ein Lock l abgebildet. T speichert nun den Status von l (Versionsnummer und das Bit, das angibt, ob l gesperrt ist) in s_1. Daraufhin liest T den Wert von v. Schließlich wird der Lock-Status erneut gelesen und in s_2 gespeichert. Gilt $s_1 \neq s_2$ oder war das Lock gesperrt, dann muss eine andere Transaktion v geändert haben, und T startet von vorne (bricht ab). Ist die Version in s_1 größer als *rv*, dann hat eine andere Transaktion v seit Beginn der Ausführung von T geändert, und T bricht ebenfalls ab. In allen anderen Fällen beobachtet T einen konsistenten Wert von v und fügt das Lock l einem Leseprotokoll hinzu. Nachdem T beim Commit alle Locks erworben hat, wird vor dem Zurückschreiben des Schreibprotokolls sichergestellt, dass alle gelesenen Variablen noch den ursprünglichen Wert haben, indem für alle Locks im Leseprotokoll geprüft wird, dass deren Versionen kleiner oder gleich *rv* sind. Dies gewährleistet, dass T noch immer konsistente Werte sieht und die Variablen im Schreibprotokoll zurückgeschrieben werden dürfen. Anderenfalls bricht T ab.

3 Erkennung von False-Sharing und Vergrößerung der Lock-Tabelle

Durch Kollisionen bei der Abbildung von Variablen auf Locks entsteht False-Sharing, was zum Abbruch von Transaktionen führen kann. Eine Transaktion T starte und kopiere den Wert $GC = 3$ nach *rv*. Eine andere Transaktion U betrete ihre Commit-Phase, inkrementiere *GC* auf 4, erwerbe ein Lock l für eine Variable v, aktualisiere v und setze die Version von l auf 4. Nun lese T eine Variable w, die ebenfalls auf l abgebildet wird. Der Lesezugriff sieht, dass die Version in l den Wert $4 > (rv = 3)$ enthält und bricht ab.

False-Sharing tritt auf, wenn die Größe der Lock-Tabelle nicht mit der Datenmenge der Anwendung skaliert. Enthalten Programme wenige konkurrierende Zugriffe, reicht eine kleine Lock-Tabelle aus. Dadurch enthält das Leseprotokoll wenige Einträge, womit dessen Konsistenzprüfung beim Commit schnell durchzuführen ist. Zusätzlich wird der Daten-Cache besser ausgenutzt, da die Locks weniger Cache-Zeilen eines Kerns belegen, die beim Erwerben/Freigeben von Locks zusätzlich invalidiert werden. Bei großen Datenmengen ist eine größere Lock-Tabelle nötig, um die Wahrscheinlichkeit von False-Sharing

gering zu halten, da False-Sharing schwerer als die oben genannten Vorteile wiegt. Daher wird ein Verfahren vorgestellt, das False-Sharing erkennt und mittels einer Heuristik die Größe der Lock-Tabelle entsprechend skaliert.

Erkennung von False-Sharing. Erwirbt eine Commit-Operation ein Lock l für eine Variable v, dann schreibt unser STM in das Lock zusätzlich zum Lock-Status die Adresse von v, d.h. es entsteht somit eine Abbildung von Locks auf Variablen. Ein Lesezugriff auf v stelle beim Inspizieren des Lock-Status von l nun fest, dass die Transaktion abzubrechen ist. Es handelt sich um einen Abbruch aufgrund von False-Sharing, falls die Adresse in l nicht mit v übereinstimmt. Durch die Erweiterung um ein Adressfeld wächst die Größe eines Locks von 8 auf 16 Byte an, wodurch zum Erwerben des Locks auch eine Doppelwort-CAS-Instruktion nötig ist. Sie ist in etwa so teuer wie ein einfaches CAS, da Prozessoren die Speicherkonsistenz generell auf Cache-Zeilenebene implementieren.

Entscheidungsheuristik. Um zu erkennen, dass False-Sharing zu häufig auftritt, verwaltet jeder Kern c zwei Zähler $C_{c,1}$ und $C_{c,2}$ in einem *Deskriptor* für die Lock-Tabelle. $C_{c,1}$ wird bei jedem Abbruch einer Transaktion auf Kern c erhöht. $C_{c,2}$ wird inkrementiert, wenn False-Sharing wie oben beschrieben erkannt wird. Bei insgesamt K Kernen ist $R = (\sum_{c=1}^{K} C_{c,2})/(\sum_{c=1}^{K} C_{c,1})$ der Anteil an aufgrund von False-Sharing abgebrochenen Transaktionen. Die Lock-Tabelle wird vergrößert, sobald $R \geq 0.1$ gilt. Experimentell hat sich gezeigt, dass ein False-Sharing-Anteil R von unter 10% vernachlässigbar ist.

Vergrößerungsoperation. Betritt eine Transaktion T ihre Commit-Phase, berechnet sie R. Falls $R \geq 0.1$ gilt, startet T die atomare, in die Phasen *EXTEND* und *XCHG* aufgeteilte Vergrößerungsoperation. Mittels einer Lock-Variable wird garantiert, dass genau eine sich im Commit befindliche Transaktion die Vergrößerung anstößt. Alle anderen Transaktionen fahren mit ihrem Commit einfach fort. Die erste Phase alloziert eine größere Lock-Tabelle, die zweite Phase ersetzt die alte Tabelle durch die neue. Damit alle Transaktionen zu jeder Zeit die selbe Tabelle verwenden, wartet die *XCHG*-Phase auf Beendigung aller Transaktionen, die noch die alte Tabelle verwenden. Danach macht sie die neue Tabelle sichtbar, setzt alle Zähler $C_{c,1}$ und $C_{c,2}$ zurück und gibt den Speicher der alten Tabelle frei.

Bei der Festlegung der neuen Tabellengröße wird angenommen, dass eine Größenverdoppelung den False-Sharing-Anteil halbiert. Demnach fällt der Anteil durch wenige Verdoppelungen unter 10%, wodurch exzessive Vergrößerungen vermieden werden. Jede Lock-Tabelle besitzt einen Header, der die Anzahl der Locks in der Tabelle angibt. Dieser Wert wird von der Hash-Funktion zur Abbildung von Adressen auf Locks verwendet.

Die *XCHG*-Phase macht die neue Tabelle sichtbar, sobald alle laufenden Transaktionen ihre Lesezugriffe bzw. Commits abgeschlossen haben. Während dieser Phase müssen alle anderen Transaktionen sicherstellen, dass eine mehrmals gelesene Variable in jeweiligen Lesezugriffen immer auf das selbe Lock abbildet abgebildet wird. Die Read- und Commit-Operationen in Abb. 1 lesen dazu zuerst die Variable *phase* und starten die Transaktion neu, falls eine andere gerade laufende Transaktion die *XCHG*-Phase gestartet hat, d.h. falls demnächst eine neue Lock-Tabelle verfügbar wird. Falls sich die Tabellengröße zwischen zwei Leseoperationen geändert hat, wird ebenfalls ein Neustart der Transaktion ausgelöst, da die Operationen in diesem Fall jeweils unterschiedliche Tabellen nutzen würden. Dazu vergleichen Read-Operationen die Größe der aktuellen Tabelle mit der Größe, die in einem

```
class LockDesc {
  int abortcounters[CORES][2];
  int sizes[CORES];
  int committers[CORES];
  int phase;
  LockTab tab;
};
LockDesc globdesc;
LockDesc readers[CORES];

int read(int *addr) {
  int c = get_core_index();
  readers[c] = globdesc;
  if (globdesc.phase == XCHG) {
    readers[c] = NULL;
    restart();
  } else if (globdesc.sizes[c] !=
      globdesc.tab.size) {
    globdesc.sizes[c] =
        globdesc.tab.size;
    readers[c] = NULL;
    restart();
  }
  ... /* Actual read operation */
  readers[c] = NULL;
}
```

```
void commit() {
  /* Lock acquisition phase */
  int c = get_core_index();
  entry last = null;
  foreach entry e in write-set {
    globdesc.committers[c]++;
    if (phase == XCHG) {
      // counted one too far
      globdesc.committers[c]--;
      goto unlock;
    }
    lock(hash(e.addr));
    last = e;
  }
  ... /* Writeback phase */
  /* Lock release phase. If commit
   * was successful, set a new
   * version, otherwise just clear
   * the lock bit. */
unlock: foreach entry e in write-set {
    globdesc.committers[c]--;
    unlock(hash(e.addr));
    if (e == last) { break; }
  }
  restart_if_commit_failed();
}
```

Abbildung 1: Synchronisation zum Austausch der Lock-Tabelle.

vorherigen Read ermittelt wurde. Die alte Größe wird im Array *sizes* gespeichert.

Gibt es lesende bzw. sich im Commit befindliche Transaktionen, muss die Transaktion, die die Lock-Tabelle ersetzt, auf deren Beendigung warten. Der Code benutzt dazu die Arrays *readers* und *committers* und aktives Warten. Zu Beginn/Ende einer Leseoperation trägt sich eine auf Kern c laufende Transaktion in $readers[c]$ ein/aus. Eine auf Kern c sich im Commit befindende Transaktion inkrementiert/dekrementiert $commiters[c]$ für jedes Lock das erworben/freigegeben wird. Der aktiv wartende Abschnitt terminiert, wenn alle $readers[i]$ *NULL* und $commiters[i]$ 0 sind. Danach ist die neue Lock-Tabelle verwendbar.

4 Eigenständige Lock-Tabellen zur Reduktion von False-Sharing

Die obige Technik reduziert False-Sharing, jedoch sind noch bessere Resultate bei einer gleichzeitig geringeren Anzahl von Locks möglich. Dazu werden kleine Lock-Tabellen für Speicherregionen mit wenigen konkurrierenden Zugriffen und passend größere Tabellen für Regionen mit vielen konkurrierenden Zugriffen benutzt. Wir verwenden dazu unsere experimentelle Sprache TransC, in der man mittels Annotationen eigenständige Lock-Tabellen spezifiziert. Wir erläutern nun, wie eigenständige Tabellen im STM vergrößert werden und danach, wie die Annotationen verwendet und implementiert werden.

Implementierung eigenständiger Lock-Tabellen. Die Implementierung verwendet einen *Lock-Tabellen-Deskriptor* pro Lock-Tabelle. Der vom Compiler erzeugte Code übergibt

```
/* Original */
//@ independent
class Parent {
  private Child c;
  Parent(Child c) { this.c = c; }
  void par_inc() {
    c.val++;
} }
class Child { int val; }

void inc(Parent p) {
  atomic { p.par_inc(); }
}

void main() {
  Child c1 = new Child();
  Child c2 = new Child();
  c1.val = c2.val = 1;
  Parent[2] p = new Parent[2];
  p[0] = new Parent(c1);
  p[1] = new Parent(c2);
  parfor i = 0 to 1 {
    increment(p[i]);
} }
```

```
/* Generated */
class Parent {
  /* ... */

  void par_inc(Txn t) {
A:  LockDesc orig = t.table;
    t.table = this.d;
    // c.val++;
B:  if (c.d == NULL) {
      if (CAS(c.d, NULL, t.table)) {
        refcount(c.d);
    } }
    int v = t.read(&c.val, c.d);
    t.write(&c.val, v + 1, c.d);
C:  t.table = orig;
  } }

  void inc(Parent p) {
    // atomic {
    Txn t = new Txn();
D:  t.table = globdesc;
    p.par_inc(t); // p.par_inc();
    t.commit(); // }
  }
```

Abbildung 2: Implementierung der Annotationen.

jeder transaktionalen Operation einen Zeiger auf den zu benutzenden Deskriptor, um Adressen auf Locks abzubilden. Eine Schreiboperation auf Variable v speichert nun sowohl den Deskriptor als auch den zu schreibenden Wert für v in einem Eintrag im Schreibprotokoll. In der Commit-Phase können dadurch Variablen auf Locks in den entsprechenden Tabellen abgebildet werden. Die Zähler $C_{c,1}$ und $C_{c,2}$ werden nun pro Tabelle verwaltet. Eine Vergrößerungsoperation betrifft nun jeweils eine einzelne Tabelle. Vor der Ersetzung einer Tabelle T prüft der aktiv wartende Abschnitt, dass es keine Leseoperation gibt, die auf Locks in T zugreift. Dazu wartet der Code, bis alle Einträge in *readers* ungleich *NULL* sind. Ansonsten bleibt der in Abschnitt 3 beschriebene Mechanismus unverändert.

Implementierung von Annotationen. Anhand des linken Code-Fragments in Abb. 2 wird beschrieben, wie TransC Code für annotierte Objekte erzeugt (rechtes Code-Fragment). Der Programmierer spezifiziert eigenständige Lock-Tabellen für *Parent*-Objekte, indem er der Klasse *Parent* die Annotation *independent* voranstellt. Bei der Erzeugung eines *Parent*-Objekts wird jeweils eine eigenständige Tabelle alloziert und dem Objekt zugewiesen. Allen anderen Objekten ohne Annotation wird zur Laufzeit einmalig und permanent eine beliebige Lock-Tabelle zugewiesen. Angenommen, ein Objekt o verwendet eine eigenständige Lock-Tabelle. Die Auswahlheuristik für unannotierte Objekte ist, jedem transitiv von o aus erreichbaren Objekt die von o verwendete Tabelle zuzuweisen. Diese Heuristik ist für Objekte o sinnvoll, die Container-Datenstrukturen bilden, bei denen also jedes transitiv von o erreichbare Objekt auch nur von o aus erreichbar ist.

Jeder Objekt-Header besitzt ein Feld d, das auf den Deskriptor der ausgewählten Tabelle zeigt und transaktionalen Zugriffsoperationen übergeben wird. Beim Erzeugen eines annotierten Objekts wird eine neue Lock-Tabelle reserviert und d zugewiesen. In allen

anderen Objekten ist d initial *NULL*. Im Beispiel besitzt jedes *Parent*-Objekt eine eigene Lock-Tabelle. Beim Zugriff auf ein *Child* wählt die Transaktion einmalig eine Tabelle aus und setzt d entsprechend. Um die oben beschriebene Heuristik zu implementieren, verwaltet jede Transaktion eine zu verwendende Lock-Tabelle, die anfangs der globalen Tabelle entspricht (Zeile D). Beim Aufruf einer Methode eines annotierten Objekts o wird die zu verwendende Tabelle auf die Tabelle von o gesetzt (Zeile A), beim Verlassen der Methode wird die vorherige zu verwendende Tabelle wiederhergestellt (Zeile C). Beim Zugriff auf ein Objekt ohne Tabelle wird atomar die aktuell zu benutzende Tabelle zugewiesen (Zeile B). Mittels eines Referenzzählers wird eine eigenständige (aber niemals die globale) Lock-Tabelle freigegeben, sobald kein Objekt mehr diese Tabelle benutzt.

Durch die atomare Zuweisung von Tabellen an Objekte führen fehlerhafte Annotationen nicht zu inkorrektem Verhalten. Verwendet ein Programmierer unbeabsichtigt die selbe Tabelle für unabhängige Objekte, so verhält sich das Programm im Wesentlichen so, als ob die Objekte die globale Lock-Tabelle verwenden. Ein unannotiertes und ein falsch annotiertes Programm werden somit ähnliche Laufzeiten besitzen.

5 Auswertung

Für die Auswertung wurden mehrere Programme der STAMP-Benchmark-Suite [CCKO08], sowie drei selbst geschriebene Benchmarks verwendet.

STAMP. Aus STAMP werden die Benchmarks *Genome*, *Intruder*, *Kmeans*, *Labyrinth* und *Yada* verwendet. *Ssca2* bzw. *Bayes* sind auf unserem prototypischen STM bzw. TL2 nicht ausführbar. Wir verwenden eine vereinfachte Version von *Vacation* als eines unserer eigenen Benchmarks. Eigenständige Lock-Tabellen wurden in *Genome* für eine Hash-Tabelle und eine Liste von Gensegmenten benutzt, in *Intruder* für eine Liste von Netzwerkpaketen, in *Labyrinth* für eine Liste von Vektoren von Pfaden und in *Yada* für einen Heap.

Library (Lib) simuliert Leute, die Bücher ausleihen. Es gibt $8\,192$ Regale mit einem Array von jeweils 512 Büchern. Ein atomarer Block wählt ein zufälliges Buch aus dem zugehörigen Regal und zählt, wie oft das Buch ausgeliehen wurde. Da sich ein Buch immer im selben Regal befindet, wird für jedes Regal eine eigenständige Tabelle verwendet.

Bank simuliert 64 Banken pro Kern und $4\,096$ Konten pro Bank, wobei Beträge zufällig zwischen Konten überwiesen werden. Nach jedem Transfer prüft ein atomarer Block, dass der Gesamtbetrag über alle Konten unverändert ist. Da Überweisungen nicht über Banken hinweg stattfinden, kann jede Bank mit einer eigenen Lock-Tabelle arbeiten.

Vacation simuliert Urlauber, die Trips in vier verschiedene Länder buchen. Pro Land verwaltet eine Hash-Tabelle mit 64 Buckets Hotels, Mietautos und Flüge. Eine Reservierung wird atomar durch Prüfen der Verfügbarkeit eines Hotels, Autos und Flugs, sowie der Buchung selbst vorgenommen. Es gibt eine eigenständige Lock-Tabelle pro Bucket.

Die Benchmarks wurden auf einem Rechner mit 24 GB RAM und Acht-Kern-CPU (X5550, 2.66 GHz) ausgeführt. Abb. 3 und 4 zeigen auch Werte für TL2, da unser STM auf dessen Design aufbaut. Wenn unser STM eine Lock-Tabelle fester Größe benutzt, sind die Lauf-

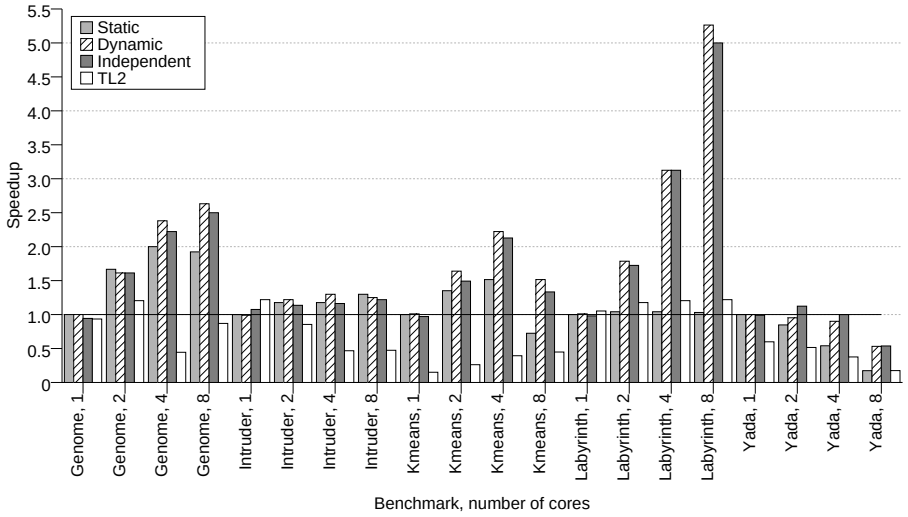

Abbildung 3: Ergebnisse für STAMP.

zeiten bis auf Abweichungen wegen unterschiedlicher Implementierungsdetails ähnlich. Bessere Laufzeiten sind also auf den Vergrößerungsmechanismus zurückführbar.

Für die Messungen mit fester Tabellengröße wurde diese jeweils auf 1024 Einträge gesetzt. Dadurch lässt sich sowohl die Wirksamkeit der Vergrößerungsmechanismen zeigen, als auch, wie groß die Lock-Tabelle pro Benchmark tatsächlich sein muss. Wir messen mit unserem STM pro Benchmark zunächst die Laufzeiten auf einem Kern mit einer Tabelle fester Größe. Diese Laufzeit ist der Referenzwert pro Benchmark. Alle anderen Messungen werden als Speedup relativ zum Referenzwert angegeben. Abb. 3 und 4 zeigen die Speedups bei Verwendung einer Tabelle fester/dynamischer Größe (*Static/Dynamic*) und eigenständigen Lock-Tabellen (*Independent*). Der Speedup zwischen zwei Versionen bei Ausführung auf c Kernen ergibt sich durch den Vergleich der entsprechenden Balken. Tabelle 1 enthält Statistiken für die Ausführung auf acht Kernen.

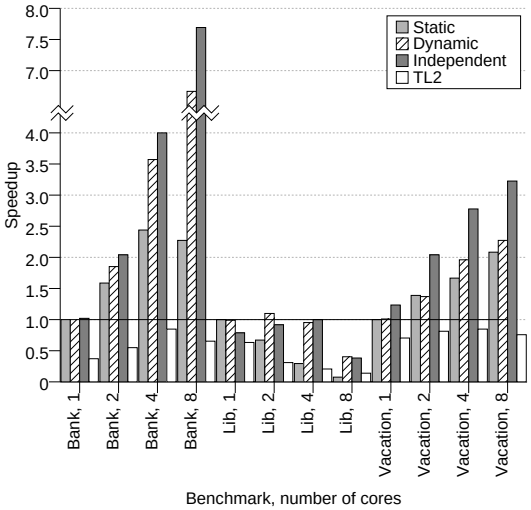

Abbildung 4: Ergebnisse für unsere Benchmarks.

Für *Intruder*, *Kmeans* und *Labyrinth* genügt die in TL2 voreingestellte Zahl von Locks (2^{20}), da die *Dynamic*-Version ma-

Tabelle 1: Statistiken: False-Sharing-Anteile R in Prozent, Anzahl Vergrößerungen N, Anzahl an Locks L und Gesamtzeit T der Vergrößerungsoperationen in ms. Die Indizes s, d und i bezeichnen jeweils die *Static*-, *Dynamic*- und *Independent*-Version.

Benchmark	R_s	R_d	R_i	N_d	N_i	L_d	L_i	T_d	T_r
Genome	98.4	0.5	1.5	11	2635	$2 \cdot 2^{20}$	$8 \cdot 2^{20} + 69710$	0.3	0.3
Intruder	37.7	3.7	2.7	3	0	$32 \cdot 2^{10}$	$1 \cdot 2^{10} + 1$	0.1	0.0
Kmeans	62.3	6.0	10.8	2	2	$8 \cdot 2^{10}$	$4 \cdot 2^{10} + 0$	0.1	0.1
Labyrinth	99.6	15.1	13.0	5	10	$1 \cdot 2^{20}$	$1 \cdot 2^{10} + 1048586$	0.2	0.2
Yada	98.0	1.0	1.2	12	15	$16 \cdot 2^{20}$	$16 \cdot 2^{20} + 32$	1.2	1.2
Bank	82.1	14.3	0.0	11	0	$2 \cdot 2^{20}$	$1 \cdot 2^{10} + 64$	0.2	0.2
Lib	76.9	0.0	9.0	8	4042	$4 \cdot 2^{20}$	$1 \cdot 2^{10} + 139520$	0.1	0.1
Vacation	81.9	3.1	6.1	11	13932	$16 \cdot 2^{20}$	$1 \cdot 2^{10} + 5140776$	0.7	0.6

ximal 2^{20} Locks benutzt. Die anderen Benchmarks brauchen bis zu 16 mal mehr Locks, was zeigt, dass für gute Laufzeiten die Anzahl an Locks mit der Datenmenge skalieren muss. Dieser Ergebnisse entsprechen auch der in [PG11] gemachten Beobachtung, dass 2^{22} Locks im Durchschnitt zu akzeptablen Laufzeiten führen. Der in der *Dynamic*-Version maximale False-Sharing-Anteil von 15% ist deutlich kleiner als in der *Static*-Version. Da es sich um den über die Laufzeit des Programms akkumulierten Anteil handelt, ist dieser größer als die angestrebten 10%. In der *Independent*-Version ist der False-Sharing-Anteil teilweise größer als in der *Dynamic*-Version, da der akkumulierte Anteil über alle Lock-Tabellen gemittelt wurde.

Die Tabelle erreicht nach wenigen Vergrößerungen ihre endgültige Größe. *Yada* und *Vacation* verwenden die selbe finale Anzahl an Locks, die aber nach einer unterschiedlichen Anzahl an Vergrößerungen erreicht wird, da die Tabellen in unterschiedlichen Schrittweiten wachsen. Die Tabellen werden also abhängig vom aktuellen False-Sharing-Anteil vergrößert. Eigenständige Lock-Tabellen verursachen insgesamt mehrere Vergrößerungen, aber die Anzahl an Vergrößerungen pro Tabelle ist klein. Von den Benchmarks, die eigenständige Tabellen benutzen, brauchen *Intruder*, *Bank*, *Library*, und *Vacation* weniger Locks als bei Verwendung einer Tabelle ($L_i \ll L_d$), was Speicher spart. Nur in *Genome* enthalten die eigenständigen Tabellen signifikant mehr Locks als bei Verwendung einer globalen Tabelle. Die Vergrößerungsoperationen sind mit höchstens $1.2ms$ (0.2% der Gesamtlaufzeit von *Yada*, dem Benchmark mit der kürzesten Laufzeit) vernachlässigbar.

Tabelle 2 zeigt die Speedups der *Dynamic*- gegenüber der *Static*-Version bzw. TL2.[1] Generell werden gute Speedups erzielt, da der Vergrößerungsmechanismus den False-Sharing-Anteil so stark reduziert, dass dadurch weniger Transaktionen abgebrochen werden müssen.

Das Vergrößern der Tabelle ist nur für *Genome* und *Intruder* ohne Effekt. *Genome* hat in der *Static*-Version einen False-Sharing-Anteil von 98%, aber kaum konkurrierende Zugriffe. Daher gibt es wenig abbrechende Transaktionen, d.h. ein Vergrößern der Tabelle reduziert die Anzahl der Abbrüche kaum. *Intruder* hat viele konkurrierende Zugriffe, aber in der *Static*-Version einen relativ kleinen False-Sharing-Anteil von 38%. Da False-Sharing

[1] Wir vergleichen *Dynamic* mit *Static*/TL2 auf c Kernen und nehmen den besten Speedup.

Tabelle 2: Speedups der *Dynamic*-Version gegenüber *Static* und TL2.

	Genome	Intruder	Kmeans	Labyrinth	Yada	Bank	Lib	Vacation
Static	1.3	1.1	2.2	3.0	3.0	3.0	4.6	1.6
TL2	5.4	2.8	6.7	5.0	3.0	10.3	5.3	2.8

kaum vorkommt, hat ein Vergrößern der Tabelle keinen Effekt, wodurch die Kosten der Vergrößerung nicht amortisiert werden. Generell sind die Speedups jedoch beachtlich.

Eigenständige Lock-Tabellen verbessern die Zeiten für *Yada*, *Bank* und *Vacation* weiter. In *Yada* wird getrennt von anderen atomaren Blöcken auf einem Heap gearbeitet, so dass für diesen eine eigene Tabelle sinnvoll ist. Da in *Bank* eine Bank exklusiv von einem Kern benutzt wird, gibt es bei Verwendung einer Tabelle pro Bank kein False-Sharing. Die Verwendung einer Tabelle pro Bucket in einer Hash-Tabelle in *Vacation* eliminiert False-Sharing zwischen Einträgen in unterschiedlichen Buckets. Die Kosten für die atomare Zuweisung von Lock-Tabellen an Objekte erhöht die Laufzeiten in den anderen Benchmarks leicht. Aufgrund der wenigen konkurrierenden Zugriffe in *Genome* können, wie bei der dynamischen Tabelle, eigenständige Tabellen die Laufzeit nicht verbessern. In *Intruder* und *Labyrinth* wird jeweils eine eigenständige Lock-Tabelle für eine globale Liste benutzt. Es gibt keine Verbesserungen der Laufzeit, da Zugriffe einseitig über die Datenstrukturen verteilt sind, so dass Adressen hauptsächlich auf die Locks einer Tabelle abgebildet werden. Da *Library* viele eigenständige Tabellen benutzt, sind mehrere Vergrößerungen wahrscheinlich, so dass andere Transaktionen warten, bis die neuen Tabellen verfügbar sind, was die Ausführungszeiten verschlechtert. Bei richtiger Anwendung führen eigenständige Lock-Tabellen aber zu besseren Laufzeiten bei gleichzeitig geringerer Anzahl von Locks.

6 Verwandte Arbeiten

TinySTM [FFR08] misst sekündlich den Transaktionsdurchsatz und verdoppelt oder halbiert die Tabellengröße zufällig. Falls TinySTM diese Konfiguration noch nicht verwendet hat, ersetzt es die aktuelle Tabelle. Verringert sich nun der Transaktionsdurchsatz, wird die alte Konfiguration wiederhergestellt. Unser Ansatz prüft bei jedem Commit auf False-Sharing und reagiert schneller als innerhalb einer Sekunde. Die Wahl der neuen Tabellengröße ist bei uns nicht zufällig, sondern abhängig vom False-Sharing-Anteil.

Ein weiterer Punkt, der die Leistung von STMs beeinflusst ist, wie viele Bytes an Daten ein Lock schützt. Gängig ist, dass ein Lock ein Wort oder eine Cache-Zeile schützt. Da bei Cache-Zeilen-Granularität ebenfalls False-Sharing möglich ist, nutzt [SEK11] eine statische Analyse, die die für ein Programm geeignetere Granularität bestimmt. Unser Ansatz verringert False-Sharing stattdessen durch ein Vergrößern der Lock-Tabelle zur Laufzeit.

Contention-Management [SDMS09] sorgt dafür, den Durchsatz von Transaktionen unter hoher Contention zu erhöhen, z.B. indem bei einem Abbruch von Transaktionen diejenige Transaktion neu gestartet wird, die weniger Arbeit verrichtet hat. Die Arbeit ist orthogo-

nal zu unserem Ansatz, da Transaktionen, die aufgrund von False-Sharing abgebrochen werden, auch bei Verwendung eines Contention-Managers weiterhin neu zu starten sind.

Adaptive Locks werden in [UBES09] verwendet, um atomare Blöcke mit Mutexen zu annotieren. Abhängig vom Laufzeitverhalten wechselt dieses System zwischen Synchronisierung mit Transaktionen und Mutexen. Adaptive Locks reduzieren wie unsere Arbeit den Aufwand von Transaktionen, jedoch wird in unserem Ansatz annotiert, welche Datenstrukturen mit eigenständigen Lock-Tabellen geschützt werden sollen, wohingegen bei adaptiven Locks die von atomaren Blöcken zu verwendenden Mutexe angegeben werden.

7 Zusammenfassung

In dieser Arbeit wurde die Lock-Tabelle fester Größe in STMs als Flaschenhals erkannt und ein Verfahren vorgestellt, das die Tabellengröße der Datenmenge anpasst und auch für mehrere Tabellen funktioniert, die mittels Annotationen spezifiziert werden. Weitere Arbeiten könnten einen Verkleinerungsmechanismus untersuchen. Experimente haben die Wirksamkeit des Verfahrens belegt, das einen maximalen Speedup von 10.3 erreicht und bis zu 1024 mal weniger Locks als TL2 benötigt.

Literatur

[CCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis und Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC'08: Proc. Symp. Workload Characterization*, Seiten 35–46, Seattle, WA, Sep. 2008.

[DSS06] Dave Dice, Ori Shalev und Nir Shavit. Transactional Locking II. In *DISC'06: Proc. 20th Intl. Symp. Distributed Computing*, Seiten 194–208, Stockholm, Sweden, Sep. 2006.

[FFR08] Pascal Felber, Christof Fetzer und Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proc. Symp. Principles and Practice of Parallel Prog.*, Seiten 237–246, Salt Lake City, UT, Feb. 2008.

[PG11] Mathias Payer und Thomas R. Gross. Performance evaluation of adaptivity in software transactional memory. In *ISPASS '11: Proc. Intl. Symp. on Performance Analysis of Systems and Software*, ISPASS '11, Seiten 165–174, Austin, TX, Apr. 2011.

[SDMS09] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe und Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. *SIGPLAN Not.*, 44(4):141–150, 2009.

[SEK11] Martin Schindewolf, Alexander Esselson und Wolfgang Karl. Compiler-assisted selection of a software transactional memory system. In *ARCS '11: Proc. Intl. Conf. on Archit. Computing Systems*, ARCS'11, Seiten 147–157, Como, Italy, Feb. 2011.

[UBES09] Takayuki Usui, Reimer Behrends, Jacob Evans und Yannis Smaragdakis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *PACT'09: Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques*, Seiten 3–14, Raleigh, NC, Sep. 2009.