1. Verwendete Implementationen

- Monitor Facade (*Monitor*)
- Feingranularen Monitor (*SmallLock*)
- Compare and set Immuable collections und Klassen (*Cas*)
- Striped compare and set für geringere contention (*Accumulator*)
- Software Transactional Memory (Clojure) (*Ref*)

2. Unterschiede der Implementationen

Monitor Facade:

Die Bank besitzt alle Accounts. Die Bank ist voll Synchronisiert. Die beiden Eigenschaften führen dazu das auf die Objekte innerhalb der Bank immer nur ein Thread zugreifen kann. Es sind wenige Arbeitsschritte notwendig um diese Art der synchronisierung von einer Thread unsicheren implementation abzuleiten.
Skalierbarkeit: sehr schlecht weil mehr threads keinen vorteil bringt.
Performanz: am schlechtesten **Rechentier-score: 680.954,57 ops/s**
Lebendigkeit: Es kann nicht zu Deadlocks oder Livelocks kommen, da bei einer Ressource kein wechselseitiger Ausschluss auftreten kann. Durch zentralen Lock liegt die Verantwortung der Fairness beim Scheduler.
Fortschritt: Immer genau 1 Thread macht Fortschritt.

Feingranulare Monitor:

Jeder Account ist allein voll Synchronisiert. Es gibt ein Read Write lock für Transfers um Deadlocks und sichtbarkeit von inkonsistenten Zuständen zu vermeiden.

Skalierbarkeit: Der fein granulare Monitor wird durch erhöhung der Threads immer besser.

Benchmark	Threads	Score	Score Error (99,9%)	Unit
BenchBank.mixed	5	4.817.074	6.435.218	ops/s
BenchBank.mixed	10	5.394.063	11.428.480	ops/s
BenchBank.mixed	20	6.195.198	8.743.315	ops/s
BenchBank.mixed	35	8.040.882	8.727.390	ops/s
BenchBank.mixed	65	11.356.426	6.422.838	ops/s

Performanz: Bei niedriger Threadzahl gute Performanz. Bei hoher Threadzahl die beste Performanz. **Rechentier-score: 6.050.105,75 ops/s**
Lebendigkeit: Die Verantwortung für Deadlocks, Livelocks und Starvation liegt beim Entwickler. In diesem Konstrukt kann es schnell zu Liveness Problemen kommen.
Fortschritt: Pro Account kann ein Thread Fortschritt machen. Es kann nur eine transfer aktion gleichzeitig passieren. Fortschritt ist hier besser als bei der Monitor Facade, jedoch schwerer zu durchblicken.

Compare and Set:

Es existiert eine atomare Referenz auf alle Accounts. Dies wird durch Immutable Persistent Collections(Menge an Accounts, die Accounts selbst, die Menge der Einträge pro Account und die Einträge selbst) ermöglicht. Wenn ein einzelner Account verändert werden soll, muss die gesamte Menge an Accounts durch eine neue geänderte Menge an Accounts ersetzt werden.
Skalierbarkeit: Konstante Geschwindigkeit bei steigender Anzahl an Threads.
Performanz: Bei niedriger Threadzahl die beste Performanz. Bei hoher Threadzahl gut Performanz.
Lebendigkeit: Keine Deadlocks, da Lockfrei programmiert wurde. Keine Livelocks weil nur eine zentrale Resource besteht. Starvation sollte im Regelfall nicht auftreten, es sei denn ein Thread hat dauerhaft “Pech” und muss dauerhaft neu versuchen seine Transaktion abzuschließen.
Fortschritt: Es kann mindestens 1 Thread Fortschritt machen.

Striped Compare and Set:

Inspiriert vom LongAccumulator. Hier besteht statt einer Atomaren Referenz, idealerweise eine Referenz pro Thread. Das Ziel dieser Implementation ist die Anzahl der “retries” zu verringern. Um den Gesamtzustand der Bank zu bestimmen müssen die Zustände der einzelnen Referenzen aggregiert werden.
Diese implementierung ist bei schreibenden zugriff schneller aber bei lesenden Zugriffen deutlich schlechter als normales Compare and Set
Skalierbarkeit: Mit zunehmender Anzhl an Threads, abnehmende Geschwindigkeit.

Benchmark	Threads	Score	Score Error (99,9%)	Unit
BenchBank.mixed	5	1.696.327	539.368	ops/s
BenchBank.mixed	10	925.781	147.349	ops/s
BenchBank.mixed	20	1.034.798	456.702	ops/s
BenchBank.mixed	35	1.001.881	472.596	ops/s

BenchBank.mixed	65	680.426	1.793.923	ops/s

Performanz: Nicht sehr gut, jedoch immer noch besser als die grob granulare Monitor Implementation. **Rechentier-score: 799.974,96 ops/s.**
Lebendigkeit: Gleiche Lebendigkeit wie bei normalem Compare and Set, jedoch geringere Starvation Gefahr, da bei einem neuen Versuch eine andere atomare Referenz verwendet wird.
Fortschritt: Es können mehrere Threads Fortschritt erzeugen.

Software Transactional Memory (Clojure)

Eine Atomare Reference auf eine Map. Die Map enthält eine maping von Account Number zu Clojure Ref. Jede Ref zeigt auf einen Imutable Account (auch eine clojure map). Ref bieten die möglichkeit transaktionen über mehrere refs hinweg zu bauen. Der erwartete Vorteil war das mehrere Account gleichzeitig geändert werden können, aber trotzdem ein Atomarer transfer durchgeführt werden kann.
Skalierbarkeit: Die Geschwindigkeit wird mit zunehmender Anzahl an Threads langsamer
Performanz: Mittel, **Rechentier-Score: 1.655.881 ops/s**
Lebendigkeit: Es können keine Deadlocks auftreten, weil Lockfrei programmiert wurde. Kein offensichtliches Risiko für Livelocks. Noch geringere Starvation Gefahr, als bei Striped Compare and Set.
Fortschritt: Es können viele Threads Fortschritt machen da einzahlen und auszahlen in einem Account nicht im Konflikt mit ein- und auszahlen in einem anderen Account steht. Ein Transfer ändert zwei Accounts und steht im Konflikt mit änderungen in diesen beiden. Generell bester Fortschritt, weil pro Account mindestens ein Thread Fortschritt machen könnte

3. Messdaten

Gemessen wurde mit 100 Accounts, 40 Threads. Es wurden 10 warm up Iterationen und 8 Iterationen zur Messung durchgeführt. Der Benchmark “mixed” zeigt die Summe aller getesteten Funktionen (deposit, withdraw, getEntries, balance, transfer)

Benchmark	Score	Score Error (99,9%)	Unit	Param: implName
BenchBank.mixed	680.955	380.214	ops/s	Monitor
BenchBank.mixed	6.050.106	5.213.915	ops/s	SmallLock
BenchBank.mixed	4.466.297	571.027	ops/s	Cas
BenchBank.mixed	799.975	150.057	ops/s	Accumulator
BenchBank.mixed	1.655.881	411.560	ops/s	Ref

Weitere Messdaten können im Ornder java/doc/two/rechentier gefunden werden
Ergebnisse von einem 2 Kern Laptop mit hyperthreading, 6 Threads, 100 Accounts:

Benchmark	Score	Score Error (99,9%)	Unit	Param: implName
BenchBank.mixed	304.686	925.509	ops/s	Monitor
BenchBank.mixed	7.083.767	4.392.010	ops/s	SmallLock
BenchBank.mixed	6.001.634	1.542.565	ops/s	Cas
BenchBank.mixed	1.644.967	660.686	ops/s	Accumulator
BenchBank.mixed	3.319.848	885.424	ops/s	Ref

Die Datei java/doc/two/jmh-results-mixed-laptop.ods zeigt die Zusammensetzung des Wert “mixed”.

4. Messdaten Skalierbarkeit

Benchmark	Threads	Score	Score Error (99,9%)	Unit	Param: implName
BenchBank.mixed	5	660.458,01	524.459,80	ops/s	Monitor
BenchBank.mixed	5	4.817.074,26	6.435.218,35	ops/s	SmallLock
BenchBank.mixed	5	4.817.358,47	3.068.726,86	ops/s	Cas
BenchBank.mixed	5	1.696.326,56	539.368,16	ops/s	Accumulator
BenchBank.mixed	5	1.195.346,92	517.572,95	ops/s	Ref
BenchBank.mixed	10	503.986,94	736.734,09	ops/s	Monitor
BenchBank.mixed	10	5.394.062,77	11.428.479,83	ops/s	SmallLock
BenchBank.mixed	10	3.953.324,19	849.379,82	ops/s	Cas
BenchBank.mixed	10	925.780,70	147.349,03	ops/s	Accumulator
BenchBank.mixed	10	859.516,23	510.914,77	ops/s	Ref
BenchBank.mixed	20	528.646,31	771.943,97	ops/s	Monitor
BenchBank.mixed	20	6.195.198,07	8.743.314,56	ops/s	SmallLock

BenchBank.mixed	20	5.311.270,57	558.854,65	ops/s	Cas
BenchBank.mixed	20	1.034.797,58	456.702,21	ops/s	Accumulator
BenchBank.mixed	20	792.526,91	465.865,97	ops/s	Ref
BenchBank.mixed	35	504.346,97	626.766,98	ops/s	Monitor
BenchBank.mixed	35	8.040.881,81	8.727.389,60	ops/s	SmallLock
BenchBank.mixed	35	4.182.181,52	672.472,00	ops/s	Cas
BenchBank.mixed	35	1.001.881,37	472.595,69	ops/s	Accumulator
BenchBank.mixed	35	660.375,77	294.823,32	ops/s	Ref
BenchBank.mixed	65	666.592,16	803.167,71	ops/s	Monitor
BenchBank.mixed	65	11.356.426,37	6.422.837,92	ops/s	SmallLock
BenchBank.mixed	65	4.461.625,84	606.539,31	ops/s	Cas
BenchBank.mixed	65	680.426,23	1.793.922,81	ops/s	Accumulator
BenchBank.mixed	65	787.465,59	282.529,16	ops/s	Ref

6. ConcurrentLinkedList VS. VectorQueue

Beide Implementationen sind in etwa gleich schnell, da beide voll synchronisiert sind. Es treten lediglich Differenzen bei den einzelnen Funktionen auf. Zu finden ist dieser Benchmark in der Klasse “BenchList”

Benchmark	Threads	Samples	Score	Score Error (99,9%)	Unit	Param: implName
main	6	6	3.936.518	249.026	ops/s	ConcurrentLinked
main:add	6	6	1.548.280	91.200	ops/s	ConcurrentLinked
main:poll	6	6	2.388.238	175.754	ops/s	ConcurrentLinked
main	6	6	4.129.632	149.330	ops/s	Vector
main:add	6	6	2.154.593	81.028	ops/s	Vector
main:poll	6	6	1.975.039	76.991	ops/s	Vector
main	6	6	1.773.628	575.856	ops/s	ConcurrentLinked
main:add	6	6	657.453	481.669	ops/s	ConcurrentLinked
main:poll	6	6	1.116.175	245.330	ops/s	ConcurrentLinked
main	6	6	1.627.191	234.995	ops/s	Vector
main:add	6	6	862.138	163.234	ops/s	Vector
main:poll	6	6	765.053	73.794	ops/s	Vector
main	32	6	1.947.403	230.599	ops/s	ConcurrentLinked
main:add	32	6	307.368	11.511	ops/s	ConcurrentLinked
main:poll	32	6	1.640.035	234.600	ops/s	ConcurrentLinked
main	32	6	1.743.761	120.579	ops/s	Vector
main:add	32	6	897.620	91.332	ops/s	Vector
main:poll	32	6	846.142	42.326	ops/s	Vector