Asynchronous Assertions

Witold Antkiewicz

Oryginalny artykuł:

Edward E. Aftandilian

Samuel Z. Guyer

Martin Vechev

Eran Yahav

Warszawa, 2012

Plan

- Idea
- Koncepcja działania
- Snapshoty
- Implementacja
- Wyniki
- Podsumowanie

Idea

- Assercje – wygodne i proste narzędzie do monitorowania i wykrywania błędów
- Stosowane najczęściej w prostych przypadkach: *assert(p!=null)*
- Rzadko uruchamiane dla kosztownych operacji: *assert(redblack_invariants(my_tree))*

Idea

- *Asynchronous Assertions* – metoda wydajnego uruchamiania złożonych asercji
- Poszczególne asercje uruchamiane w oddzielnych wątkach
- Praca na snapshotach w celu zachowania stanu pamięci z momentu wywołania asercji

Koncepcja działania

- Wyniki asercji identyczne jak w przypadku synchronicznego liczenia
- Dwie metod obsługi błędów:
  - Całkowicie asynchroniczna
  - Synchronizowalna
- Druga metoda umożliwia wymuszenie oczekiwania na zakończenie asercji

Koncepcja działania

- Optymalizacja pamięci wykorzystywanej przez snapshoty:
  - Kopiowanie obiektów dopiero, gdy jest to niezbędne
  - Współdzielenie obiektów przez snapshoty
  - Obiekty powstałe po starcie asercji nie są kopiowane

Koncepcja działania

- Przykładowy kod do sprawdzenia, czy lista jest posortowana

```
private boolean isOrdered() {
        Node n = head;
        while (n != null && n.next != null) {
                if (n.data > n.next.data)
                        return false;
                n = n.next;
        }
        return true;
}
```

Koncepcja działania

- Interface StrobeTask

```
public interface StrobeTask {
        public boolean compute();
}


private class InvTask implements StrobeTask {
        @ConcurrentCheck
        public boolean compute() {
                return isOrdered();
        }
}
```

Koncepcja działania

- Uruchamianie asynchroniczne

```
StrobeAssertion sassertion =
            new StrobeAssertion(new InvTask());
sassertion.go();
```

- Uruchamianie synchronizowalne

```
StrobeFuture sfuture =
            new StrobeFuture(new InvTask());
sfuture.go();
...
assert(sfuture.get());
```

Snapshoty

- Program może modyfikować pamięć po wywołaniu asercji
- Niezbędne jest przechowywanie starszych wersji modyfikowanych obiektów
- Obiekty, które nie ulegają zmianie, nie muszą być przechowywane
- Kopiowanie odbywa się tuż przed próbą modyfikacji obiektu

Snapshoty

Pojedynczy snapshot

- Flaga *active*
- *Current(o)* – aktualna wersja
- *Preserved(o)* – wersja przechowywana
- *Modified(o)* – bool czy obiekt był modyfikowany

Snapshoty

Pojedynczy snapshot

$$
\begin{array}{l}
\texttt{\textbf{if}}\ active\ \texttt{\textbf{and}}\ !modified(o) \\
\quad preserved(o) := current(o) = o_i \\
\quad modified(o) := \texttt{\textbf{true}} \\
\quad \texttt{write to}\ o\ \texttt{,}\ current(o) := o_{i+1}
\end{array}
$$

$$
snapshot(o) = \begin{cases} preserved(o), & modified(o) \\ current(o), & \text{otherwise.} \end{cases}
$$

Snapshoty

Pojedynczy snapshot

- Odczyt oryginalnego obiektu lub jego zachowanej wersji
- Aktualizacje wykonywane atomowo
- Zachowana tożsamość obiektów bez konieczności aktualizacji referencji

Snapshoty

Wiele snapshotów

- Informacje, które obiekty trzeba zachować dla których asercji
- Licznik asercji
- Flaga *active* dla każdej asercji
- *modifiedAt(o)* – informacja, kiedy obiekt został ostatnio zmodyfikowany

Snapshoty

Wiele snapshotów

for each assertion E_t

  if $active(E_t)$ **and** $modifiedAt(o) < E_t$

    then $preserved(o, E_t) := current(o) = o_i$

  $modifiedAt(o) := E$

  write to o , $current(o) := o_{i+1}$

$$snapshot(o, E_t) = \begin{cases} preserved(o, E_t), & modifiedAt(o) \geq E_t \\ current(o), & \text{otherwise.} \end{cases}$$

Snapshoty

Optymalizacje

- Pomijanie nowych obiektów
  - *modifiedAt(o)* inicjalizowane na E
- Współdzielenie obiektów
  - Jeśli *modifiedAt(o)* $< E$ to przynajmniej jedna asercja potrzebuje zachować obiekt

Implementacja

- Implementacja nazwana STROBE
- Wykorzystuje Jikes RVM 3.1.1

Implementacja

- Trzy najważniejsze komponenty:
  - Copying write barrier – aktywowana gdy działają asercje, odpowiada za tworzenie kopii obiektów, gdy jest to konieczne
  - Checker thread pool – Odpowiada z wykonywanie asercji, blokuje wykonywanie głównego wątku, gdy jest za dużo asercji
  - Snapshot read barrier – aktywowana, gdy wątek asercji chce odczytać dane z pamięci, odpowiada za przekazanie wersji obiektu właściwej dla danej asercji

Implementacja

- Znacznik czasu dodany do nagłówka obiektu
- Stała liczba T wątków realizujących asercje
- Maksymalnie T snapshotów i T kopi obiektów
- *Forwarding array* – przechowuje informacje o kopiach obiektu dla poszczególnych wątków
- Strażnik „*being copied*” gwarantuje, że operacje aktualizacji znacznika czasu, kopiowania obiektu i zapisu do obiektu odbywają się atomowo.

Implementacja

- Bariera zapisu

```
void writeBarrier(Object src, Object target,
                  Offset offset)
{
  int epoch = Snapshot.epoch;
  if (Header.isCopyNeeded(src, epoch)) {
    // -- Needs to be copied, we are the copier
    //    timestamp(src) == BEING_COPIED
    snapshotObject(src);
    // -- Done; update timestamp to current epoch
    Header.setTimestamp(src, epoch);
  }
  // -- Do the write (omitted: GC write barrier)
  src.toAddress().plus(offset).store(target);
}
```

Implementacja

- Bariera zapisu

```
boolean isCopyNeeded(Object obj, int epoch)
{
  int timestamp;
  do {
    // -- Atomic read of current timestamp
    timestamp = obj.prepareWord(EPOCH_POS);
    // -- If in current epoch, nothing to do
    if (timestamp == epoch)
      return false;
    // -- If someone else is copying, wait
    if (timestamp == BEING_COPIED)
      continue;
    // -- ...until CAS BEING_COPIED succeeds
  } while (!obj.attempt(timestamp, BEING_COPIED,
                        EPOCH_POS));
  return true;
}
```

Implementacja

- Bariera zapisu

```
void snapshotObject(Object obj)
{
  // -- Get forwarding array; create if needed
  Object[] forwardArr =
                Header.getForwardingArray(obj);
  if (forwardArr == null) {
    forwardArr = new Object[NUM_CHECK_THREADS];
    Header.setForwardingArray(obj, forwardArr);
  }
  // -- Copy object
  Object copy = MemoryManager.copyObject(obj);
  // -- Provide copy to each active checker
  //    that has not already copied it
  for (int t=0; t < NUM_CHECK_THREADS; t++) {
    if (isActiveCheck(t) &&
        forwardArr[t] == null)
      forwardArr[t] = copy;
  }
}
```

Implementacja

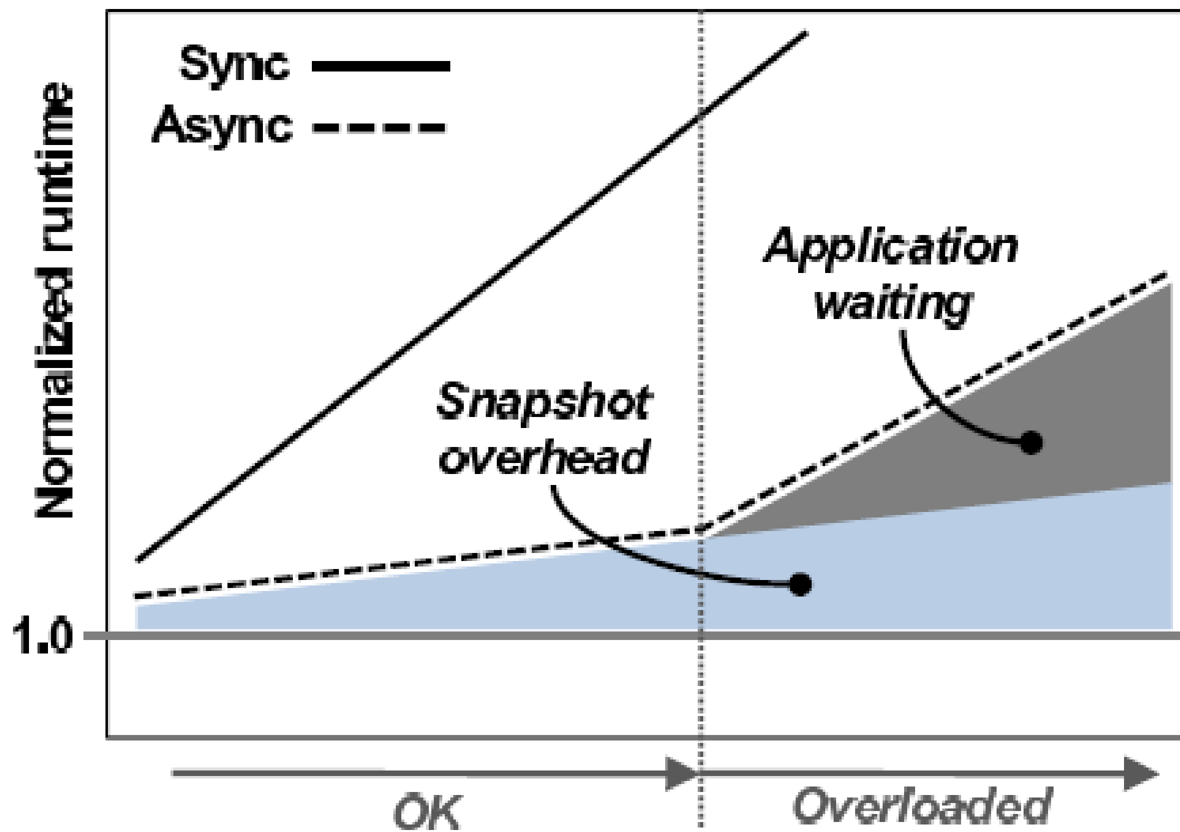
- Bariera odczytu

```
// -- Returns the object to read from,
//    either the original or the copy
Object readBarrier(Object obj)
{
  // -- Get forwarding array
  Object[] forwardArr =
             Header.getForwardingArray(obj)
  // -- No forwarding array? return original
  if (forwardArr == null) return obj;
  else {
    // -- Else load copy from forwarding array,
    //    indexing by checking thread ID
    Object copy =
             forwardArray[thisThread.checkerId];
    // -- No copy of this object? return original
    if (copy == null) return obj;
    else {
      // -- ...otherwise return copy (snapshot)
      return copy;
    }
  }
}
```

Wyniki

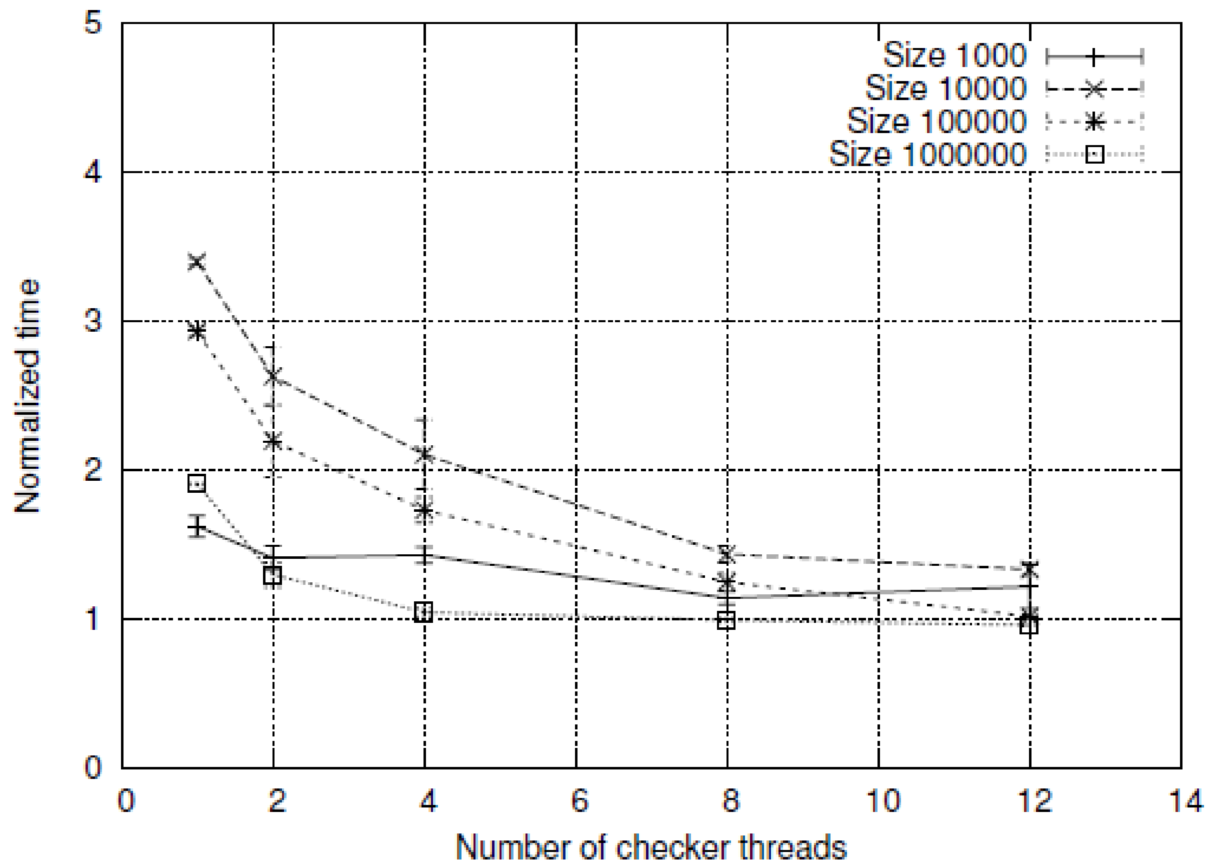
- W większości przypadków asynchroniczne asercje sprawdzają się lepiej niż synchroniczne
- Jedynie w przypadku bardzo prostych asercji, dodatkowy koszt przewyższa zyski
- Gdy liczba wątków wystarczająca, spowolnienie 10 do 60%

Wyniki



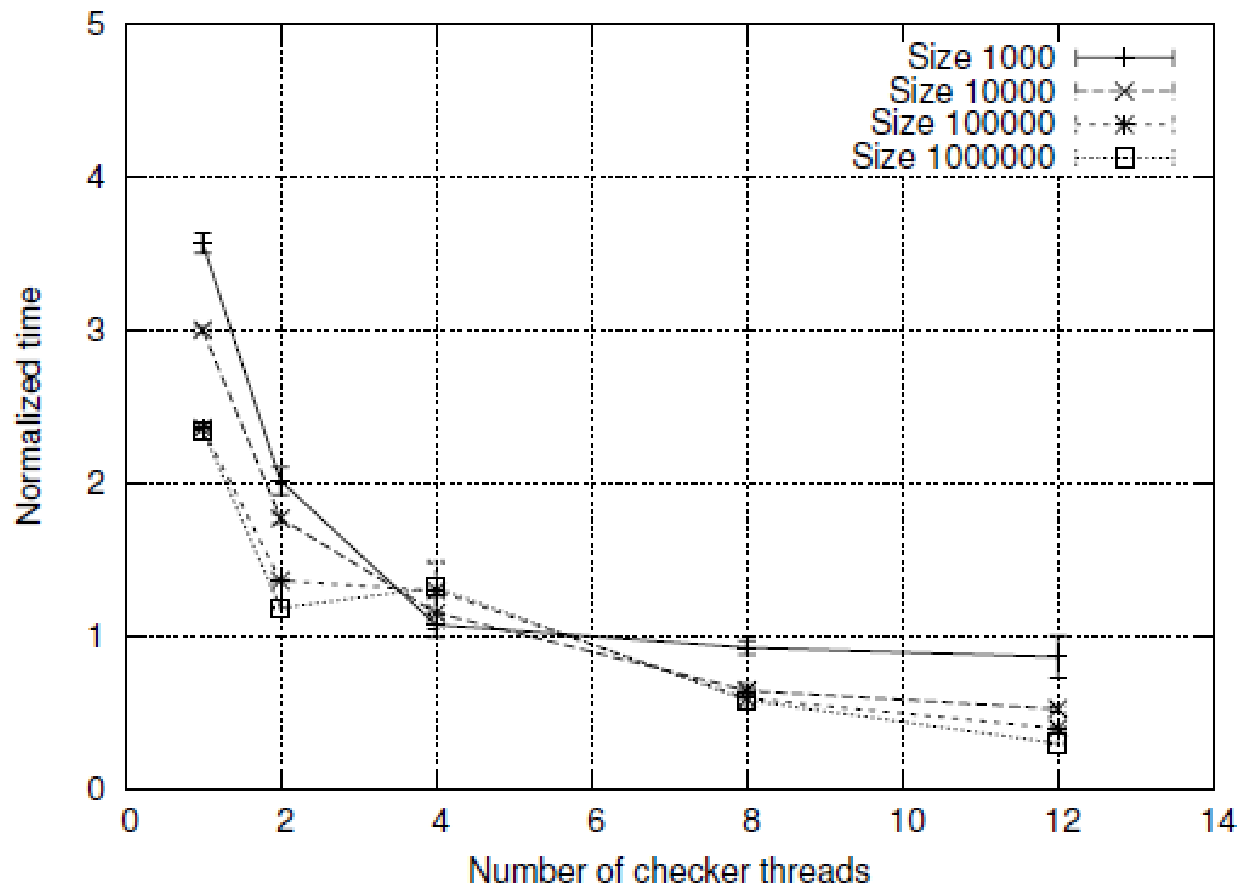
Wyniki

- Microbenchmark – ordered linked-list



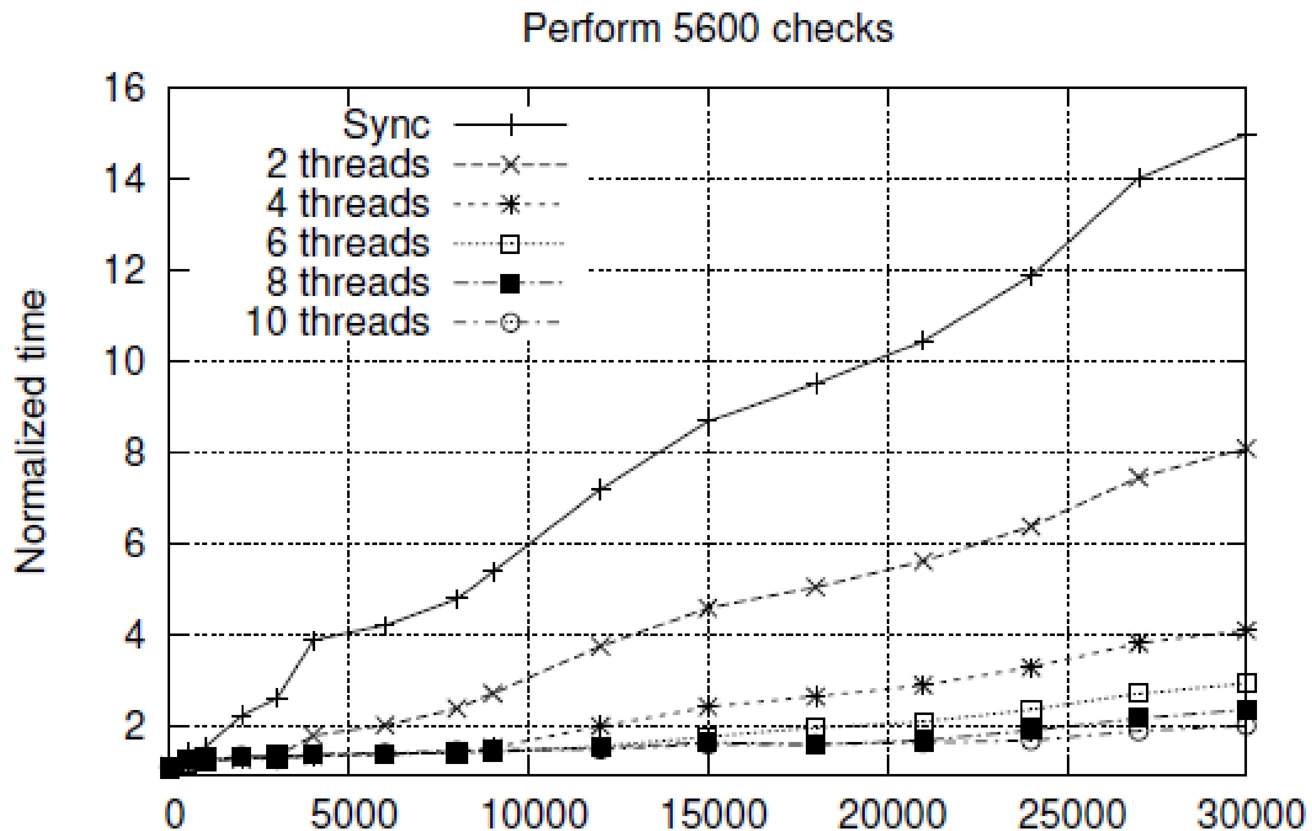
Wyniki

- Microbenchmark – red-black tree



Wyniki

- Syntetyczny benchmark



Wnioski

- Metoda asynchronicznych asercji dla skomplikowanych warunków i przy odpowiedniej liczbie wątków umożliwiwa uruchomienie pogramu w czasie znacznie krótszym niż w przypadku synchronicznego uruchomienia. W przypadku prostych asercji lub zbyt małej liczby wątków, metoda nie sprawdza się.

Dziękuję za uwagę