* Sync-preserving deadlocks sind eine Unterklasse von allen vorhersagbaren Deadlocks
* Sync-preserving deadlocks sind Deadlocks, die durch das Verändern der Ausführungsreihenfolge bei gleichzeitiger Erhaltung der Reihenfolge von kritischen und konfliktbehafteten Abschnitten vorhergesagt werden können
* Abstract deadlock patterns sollen Cluster von Deadlock patterns mit derselben Signatur sein; Die Signatur zeichnet sich durch die beteiligten Threads und Locks aus. Sind diese bei zwei deadlock patterns gleich gehören sie dem gleichen Cluster an
* Ein Algorithmus der Sound ist muss herausfinden ob es zu einem Deadlock über einen Zeugen kommt
* Ein Zeuge ist eine Umordnung p von (einem Teil von) dem Trace, welcher aber wieder ein valider Trace ist
* Wann ist ein deadlock pattern ein sync-preserving deadlock? Es muss in einem Sync-Preserving Reordering pSP eines Trace beobachtet werden
* Was ist ein Correct Reordering?
  + Wenn es ein Teil von dem Trace ist
  + Wenn die Ordnung der Operationen innerhalb eines Threads nicht verändert werden
  + Jeder Writer für jeden Reader beibehalten (also nicht weggelassen wird)
  + Diese Regeln versichern eine wichtige Eigenschaft: Jedes Programm was den ursprünglichen Trace generiert hat, kann auch das Reordering generieren, somit dient das Reordering als echter Zeuge eines Bugs
* Ein Reordering eines Trace ist dann Sync-Preserving wenn
  + Der Kontrollfluss der gleiche ist
  + Die gemeinsame Ordnung zweier kritischer Sektionen (auf denselben Lock) beibehalten wird
* Nach Abarbeiten des Reorderings wird der Rest der Operationen in den Threads ausgeführt (hier kann dann der Sync-Preserving Deadlock festgestellt werden)
* Sync-preserving Deadlocks können im Gegensatz zu beliebigen Deadlocks effizient erkannt werden
* Die Experimente später zeigen, dass die meisten Deadlocks sync-preserving Deadlocks sind -> Schnelle Detektion kommt mit sehr geringem Precision Loss in der Praxis

Beispiel:

Ein Bild, das Text, Screenshot, Diagramm, Reihe enthält.

Automatisch generierte Beschreibung

Algorithmus:

* Correct Reordering finden (Umordnung finden, die vom Programm ebenso generiert werden kann)
* Checken, ob Correct Reordering auch Sync-Preserving ist (Sicherstellen, dass die Acq-Events auf einen bestimmten Lock in der gleichen Reihenfolge wie beim Ursprungs-Trace ausgeführt werden)
* Wenn dem so ist schauen, ob nach der Ausführung des Correct Reordering ein Deadlock Pattern existiert (Können die restlichen Events des Trace nach der Ausführung des Correct Reordering so angeordnet werden, dass ein Deadlock auftritt)

Was ist ein Correct Reordering?

* Wenn es ein Teil von dem Trace ist
* Wenn die Ordnung der Operationen innerhalb eines Threads nicht verändert werden
* Jeder Writer für jeden Reader beibehalten (also nicht weggelassen wird)
* Diese Regeln versichern eine wichtige Eigenschaft: Jedes Programm was den ursprünglichen Trace generiert hat, kann auch das Reordering generieren, somit dient das Reordering als echter Zeuge eines Bugs

Aufgabe: Finden eines Sync-Preserving Deadlock in 𝜎2

p3 = e1e2e3e8e9e12..e15e16e17

Prüfen der Bedingungen für ein Correct Reordering:

* Teil von 𝜎2? Ja, weil alle Events auch in 𝜎2 existieren
* Ordnung der Events innerhalb des Threads beibehalten?
  + T1: e1e2e12..e15
  + T2: e3
  + T3: e8e9e16e17
  + T4: nicht vorhanden
  + Schlussfolgerung: Ordnung ist beibehalten, da die Nummerierung der Events für jeden Thread weiterhin aufsteigend ist
* Jeder Writer für jeden Reader beibehalten?
  + e17 ist Reader von x, Writer von x ist e13 und ist ebenfalls im Reordering
  + e14 ist Reader von y, Writer von y ist e9 und ist ebenfalls im Reordering
  + Schlussfolgerung: Jeder Writer wurde für jeden Reader beibehalten

Schlussfolgerung für p3: p3 ist ein Correct Reordering

Es muss nun geprüft werden, ob p3 auch Sync-Preserving ist:

Ein Reordering eines Trace ist dann Sync-Preserving wenn

* Die gemeinsame Ordnung zweier Acq-Events (auf denselben Lock) beibehalten wird

Prüfen ob p3 sync-preserving ist:

* Acq-Events auf L1 in p3: e1, e8
* Acq-Events auf L2 in p3: e3
* Acq-Events auf L3 in p3: e12, e16

Schlussfolgerung: Da e1 < e8 und e12 < e16 sowohl in 𝜎2 als auch p3 gilt ist p3 sync-preserving

Prüfen, ob ein Deadlock Pattern existiert:

* T2 hält L2 und möchte in e4 L3
* T3 hält L3 und möchte in e18 L2
* Die aufeinander folgende Ausführung von e4 und e18 führt zu einem Deadlock, damit haben wir ein Deadlock Pattern D=<e4, e18> gefunden
* D=<e4, e18> ist folglich ein Sync-Preserving Deadlock

„Synchronization-preserving Deadlocks“ (auch „Sync-preserving) Deadlocks sind eine Teilmenge aller vorhersagbaren Deadlocks. Sie stellen auch die Deadlocks dar, die in der Praxis am häufigsten vorkommen. Durch den SPD-Algorithmus ist es möglich, sie ohne die Gefahr von False-Positives und darüber hinaus effizient sowohl in einem Offline- als auch in einem Online-Szenario anhand eines Beispiel-Trace zu erkennen. Mit Offline- und Online-Szenario ist hierbei gemeint, dass der SPD-Algorithmus sowohl auf Traces einer vergangenen Ausführung als auch auf Traces zur Laufzeit angewendet werden kann.

Um den SPD-Algorithmus nachvollziehen zu können, muss zuerst die Idee hinter Sync-preserving Deadlocks erläutert werden. Ein Sync-preserving Deadlock ist ein Deadlock-Pattern, welches der SPD-Algorithmus versucht innerhalb eines Trace zu erkennen. Dafür wird im ersten Schritt versucht, aus dem Trace ein sog. „Sync-preserving Reordering“ zu bilden. Um festzustellen, ob es ein Sync-preserving Reordering von einem Trace gibt, muss zuerst ein „Correct Reordering“ gefunden werden. Ein Correct Reordering, also eine korrekte Umordnung eines Trace, kann durch die Einhaltung folgender Regeln erzeugt werden:

* Es enthält Teile des originalen Trace
* Die Reihenfolge der Operationen innerhalb eines Threads wurde nicht verändert
* Für jede Leseoperation auf einer bestimmten Variable existiert in der Umordnung eine dazugehörige Schreiboperation

Diese Regeln stellen sicher, dass die resultierende(n) Umordnung(en) eines Trace genauso auch von dem Programm generiert werden könnten, welches den ursprünglichen Trace ausgegeben hat. Dieser Schritt generiert also viele mögliche Ausführungen eines Programms.

Der Trace Delta in Listing 12 enthält einen Sync-preserving Deadlock und damit auch ein Correct Reordering roh = e1e2e3e8e9e12..e15e16e17. Durch Überprüfung der oben genannten Regeln ist einfach zu erkennen, dass $\rho$ ein Correct Reordering ist. $\rho$ ist Teil von $\delta$, da alle Events aus $\delta$ stammen. Weiterhin wird die Ordnung der Operationen innerhalb der Threads beibehalten. Dies ist einfach anhand der durchgehend aufsteigenden Nummerierung der Events in $\rho$ zu erkennen. Schlussendlich existiert auch für jede Leseopertation eine dazugehörige Schreiboperation. Die einzigen Leseoperationen in $\rho$ sind $e\_{17}$, bei dem x gelesen wird, und $e\_{14}$, bei dem y gelesen wird. Zugehörig zu $e\_{17}$ ist die Schreiboperation $e\_{13}$, sowie $e\_9$ zu $e\_{14}$. Damit wären alle drei Kriterien für ein Correct Reordering erfüllt.

Im nächsten Schritt muss rho darauf geprüft werden, ob es auch ein Sync-preserving Reordering ist. Ein Correct Reordering ist genau dann sync-preserving, wenn auch die Acquire-Events auf denselben Lock in der gleichen Reihenfolge wie in delta beibehalten werden. Beispielsweise würde diese Regel durch eine Umordnung verletzt werden, die e7 vor e1 ausführt, da in delta die Acquire-Operationen auf L1 in der Reihenfolge e1,e7 ausgeführt werden.