Methodology

Sun, IBM, BEA en anderen hebben de laatste jaren veel aandacht besteed aan het optimaliseren van lock beheer en synchronisatie in hun Java 6 virtuele machines. Functies als biased locking (locking met voorkeur), lock coarsening (ruimer locken), lock elision by escape analysis (weglaten van locks na ontsnappingsanalyse) zijn ontworpen om multi-threading efficiënter te maken. Zo geavanceerd en interessant als elk van deze functies ook zijn, de vraag is of zij die belofte van versnelling werkelijk waarmaken? Ik ga deze functies onderzoeken en een poging wagen om antwoord te geven op deze performance vraag met behulp van een single-threaded benchmark.

Java 6 threading optimalisaties:

Fictie of werkelijkheid?

Het lockingmodel van Java is pessimistisch. Al is er maar een zeer kleine kans dat twee threads gegevens delen waarbij ze elkaar kunnen verstoren, zijn we gedwongen tot de draconische oplossing om locking via synchronisatie te gebruiken om dit te voorkomen. Dit terwijl praktijkonderzoek heeft aangetoond dat de kans eigenlijk klein is dat meerdere threads tegelijkertijd een lock willen bezitten. Sterker nog, in de praktijk worden de meeste locks slechts door één thread verworven. Met andere woorden: als een thread om een lock vraagt, hoeft hij zelden te wachten om hem te krijgen. Dit terwijl het verwerven van de lock een serie acties initiëert die resulteert in serieuze overhead en daarmee een vertraging van de applicatie veroorzaakt.

We hebben wel een keuze. Denk bijvoorbeeld aan het gebruik van de thread-safe StringBuffer. Wanneer heb je voor het laatst een StringBuffer object vanuit meerdere threads benaderd? Wat is de reden dat je niet StringBuilder gebruikt?

Enkel de wetenschap dat de meeste locks slechts door één thread worden benaderd helpt niet veel. Als er ook maar de kleinste kans is dat twee threads dezelfde gegevens tegelijkertijd kunnen benaderen, zijn we genoodzaakt om de toegang tot deze gegevens te synchroniseren middels locking om corruptie van gegevens te voorkomen. Alleen als we de lock in zijn runtime omgeving bekijken kunnen we de vraag beantwoorden: hebben we de lock echt nodig? Om een antwoord op deze vraag te krijgen zijn de JVM ontwikkelaars en onderzoekers aan universiteiten en instituten gaan expirimenteren met HotSpot en de JIT. Als resultaat hiervan hebben we onder andere biased locking en twee vormen van lock eliminatie onder de namen lock coarsening en lock elision. Ik zal voordat we gaan benchmarken eerst deze optimalisaties nader beschouwen zodat we begrijpen hoe ze werken.

Lock elision na escape analysis

Escape analysis of ontsnappingsanalyse behelst het beoordelen van alle object referenties in een lopende applicatie. Deze analyse is een normaal onderdeel van het werk van de runtime HotSpot profiler. Als HotSpot via ontsnappingsanalyse kan vaststellen dat referenties naar een object beperkt zijn tot een lokale scope en dat er geen referentie kan 'ontsnappen' naar een bredere scope, dan kan HotSpot de JIT instrueren om een aantal runtime optimalisaties toe te passen. Eén zo'n optimalisatie staat bekend als lock elision. Als referenties naar een lock object beperkt zijn tot een lokale scope, betekent dat, dat de thread die de lock creëerde de enige thread is die de lock ooit zal benaderen. Onder deze omstandigheden zullen de gegevens die beschermd zijn door de lock nooit corrupt raken als die lock weggelaten wordt. En dat is precies wat de JIT als optimalisa-

Jeroen Borgers
is senior consultant bij Xebia - IT Architects.

tie uitvoert: de lock wordt verwijderd als er geen referenties naar de lock kunnen ontsnappen. Zie de volgende voorbeeld methode:

```
public String concatBuffer(String s1, String s2,
String s3) {
StringBuffer sb = new StringBuffer();
sb.append(s1);
sb.append(s2);
sb.append(s3);
return sb.toString();
}
```

Figuur 1. String concatenatie met een lokale StringBuffer

Van variabele *sb* kunnen we vaststellen dat hij alleen leeft binnen de grenzen van de concatBuffer methode. Bovendien 'ontsnappen' er geen referenties naar de StringBuffer uit de scope waarin sb is gedeclareerd. Er is dus geen manier waarop een andere thead ooit toegang zou kunnen krijgen tot deze StringBuffer. Dit beseffende weten we dat de locks die deze thread-safe StringBuffer beschermen, weggelaten kunnen worden. Het lijkt zo te zijn dat lock elision ons toestaat om thread-safe code te schrijven zonder de overhead van locking in de gevallen dat het feitelijk niet nodig blijkt om thread-safe te zijn. Of dit werkelijk zo is, is een vraag die we zullen onderzoeken.

Lock coarsening

Een andere threading optimalisatie heet lock coarsening ook wel lock merging: het verruimen ofwel samenvoegen van locks. Dit vindt plaats als achtereenvolgende gesynchroniseerde stukken code samengevoegd kunnen worden in één gesynchroniseerd stuk code. Een variatie op dit thema is om achtereenvolgende gesynchroniseerde methoden samen te voegen tot één. Deze optimalisatie kan worden toegepast als hetzelfde lock object wordt gebruikt door al die methoden. Zie het voorbeeld in Figuur 2:

```
public static String concatToBuffer(StringBuffer sb,
String s1, String s2, String s3) {
sb.append(s1);
sb.append(s2);
sb.append(s3);
return sb.toString();
}
```

Figuur 2. String concatenatie met een niet-locale StringBuffer

In dit geval heeft de StringBuffer een niet-lokale scope en kan hij dus benaderd worden door meerdere threads. Derhalve zal escape analysis vaststellen dat een referentie naar de StringBuffer kan ontsnappen en dat zijn lock dus niet weggelaten kan worden. Interessant is dat de beslissing om coarsening toe te passen onafhankelijk van het aantal threads dat de lock wil verwerven, genomen kan worden. In dit voorbeeld wordt er vier keer een lock verworven: drie keer voor de append en één keer voor de toString. De eerste stap in de optimalisatie is om de methodes te in-linen, ofwel, de methode aanroep te vervangen door zijn body. Daarna kunnen alle vier de aanroepen om de lock te verwerven vervangen worden door slechts één die de hele concatToBuffer methode omvat.

Het netto effect is dat we een langer gesynchroniseerd stuk code krijgen. Als dit gesynchroniseerde stuk substantieel langer duurt, kan dit resulteren in het blokkeren van andere threads en verminderde verwerkingssnelheid. Vanwege dit risico wordt lock coarsening beperkt in de mate van verruiming van locks.

Biased locking

Biased locking is afgeleid van de observatie in de praktijk dat de meeste locks nooit verworven worden door meer dan één thread gedurende hun levenscyclus. In het zeldzame geval dat meerdere threads wel dezelfde lock verwerven en gegevens delen, is de benadering van die gegevens zelden door meerdere threads tegelijkertijd. Om het voordeel van biased locking te begrijpen in het kader van deze observatie, is het nodig dat we eerst beschrijven hoe locks worden verworven.

Het verwerven van een lock is te zien als twee stappen. Eerst verwerf je het recht op de lock. Als je dat recht hebt verworven, kun je de lock daadwerkelijk pakken en weet je ook dat je de enige bent. Om het recht op de lock te verwerven is het nodig dat de thread een relatief dure atomaire instructie uitvoert. Bij het vrijgeven van de lock wordt traditioneel ook het recht erop opgegeven.

In het geval dat een thread itereert over een gesynchroniseerd stuk code, lijkt lock coarsening op het eerste gezicht een ideale optimalisatie. In het volgende voorbeeld, te zien in Figuur 3, kunnen we de lock buiten de for-loop halen.

```
public static String concatMultipleToBuffer(String
Buffer sb, String s1, int num) {
for (int i = 0; i < num; i++) {
sb.append(s1);
}
return sb.toString();
}
```

Figuur 3. Meerdere String concatenaties met een niet-locale StringBuffer

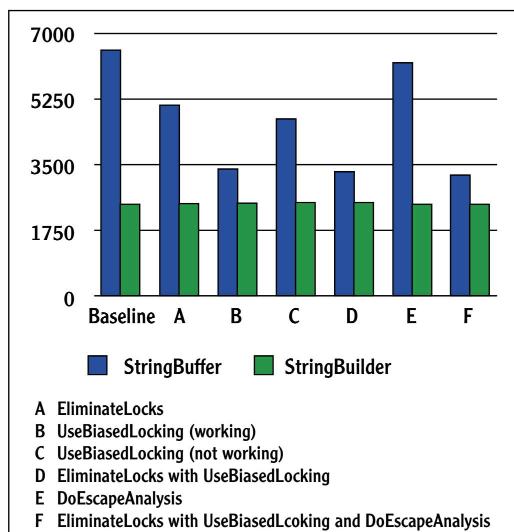
Hiermee hoeft de lock maar eenmalig verworven te worden in plaats van num keer. Dit is echter niet altijd een goede oplossing aangezien het andere threads kan uitsluiten van een eerlijke toegang tot de gedeelde gegevens. Een verstandiger oplossing in het kader van onze observatie is om de lock voorkeur (bias) te geven voor de thread die de iteraties uitvoert.

Een lock voorkeur geven voor een thread houdt in dat de thread niet meer het recht op de lock

opgeeft bij het vrijgeven van de lock. Daarmee worden achtereenvolgende verwervingen van de lock veel minder duur. Het recht op de lock wordt pas opgegeven door de voorkeursthread als een andere thread de lock probeert te verwerven. Die eerste thread geeft dan het recht op, waarna de andere thread het recht weer kan verwerven. Dit zijn relatief dure operaties die echter gezien onze observatie weinig voorkomen. Wel is het zo dat de effectiviteit van biased locking en de beslissing om het toe te passen daarmee afhankelijk is van het aantal threads dat de lock wil verwerven, in tegenstelling tot lock coarsening waarbij dit niet geldt. De Java 6 HotSpot/JIT implementatie optimaliseert default voor biased locking.

StringBuffer tegenover StringBuilder benchmark

Hoe ik het beste het effect van al deze geavanceerde optimalisaties kon meten, vond ik niet zo simpel te bepalen. Hoe kon ik de benchmark maken? Om deze vraag te beantwoorden besloot ik te kijken naar enkele van de gebruikelijke constructies die mensen de hele tijd toepassen in hun code. Strings worden door vrijwel iedereen veel gebruikt. Hier speelt de klassieke vraag: wat is de snelheidswinst van het gebruik van StringBuffer in plaats van String?
Het bekende advies is: gebruik StringBuffer in plaats van String als je hem gaat muteren. De reden voor dit advies is duidelijk: String is immutable en als er mutaties nodig zijn, is StringBuffer het goedkopere alternatief. Interessant is echter dat dit advies nalaat om StringBuffer's nieuwe ongesynchroniseerde neef, StringBuilder te erkennen. Aangezien het enige verschil tussen StringBuilder en StringBuffer de synchronisatie is, lijkt een benchmark die de performance verschillen tussen de twee meet de kosten van syn-



Figuur 4. Benchmark resultaten: tijd in ms. voor de zeven gevallen.

```
public class LockTest {
private static final int MAX = 20000000; // 20 mil-
lion

public  static  void  main(String[]  args)  throws
InterruptedException {
    // warm up the method cache
    for (int i = 0; i < MAX; i++) {
        concatBuffer("Josh", "James", "Duke");
        concatBuilder("Josh", "James", "Duke");
    }

    System.gc();
    Thread.sleep(1000);

    System.out.println("Starting test");
    long start = System.currentTimeMillis();
    for (int i = 0; i < MAX; i++) {
        concatBuffer("Josh", "James", "Duke");
    }
     long bufferCost = System.currentTimeMillis() -
start;
    System.out.println("StringBuffer: " + bufferCost
+ " ms.");

    System.gc();
    Thread.sleep(1000);

    start = System.currentTimeMillis();
    for (int i = 0; i < MAX; i++) {
        concatBuilder("Josh", "James", "Duke");
    }
    long builderCost = System.currentTimeMillis() -
start;
    System.out.println("StringBuilder: " + builder-
Cost + " ms.");
     System.out.println("Thread safety overhead of
StringBuffer: "
     + ((bufferCost * 10000 / (builderCost * 100))
- 100) + "%\n");

}

public static String concatBuffer(String s1, String
s2, String s3) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    sb.append(s3);
    return sb.toString();
}

public static String concatBuilder(String s1, String
s2, String s3) {
    StringBuilder sb = new StringBuilder();
    sb.append(s1);
    sb.append(s2);
    sb.append(s3);
    return sb.toString();
}

}
```

Listing 1

chronisatie bloot te leggen. Het speurwerk begint met de vraag: wat zijn de kosten van locking met slechts één enkele thread?
De benchmark is te zien in listing 1. De essentie ervan is om een handvol Strings te pakken en deze aan elkaar te concateneren. De initiële capaciteit van de onderliggende buffer is groot genoeg om de drie samen te voegen strings te bevatten. Dit staat ons toe om de hoeveelheid werk in het gesynchroniseerde deel te minimaliseren en de kosten van synchronisatie duidelijk te kunnen onderscheiden.

Wat is de snelheidswinst van het gebruik van StringBuffer in plaats van String?

Benchmark resultaten

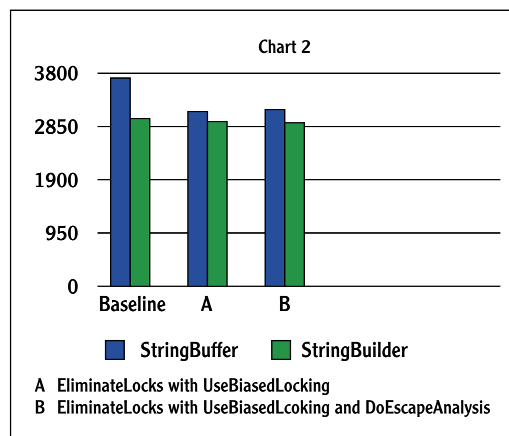
In figuur 4 staan de resultaten van zinvolle combinaties van de drie opties: EliminateLocks, UseBiasedLocking, en DoEscapeAnalysis.
Het doel van het gebruik van de ongesynchroniseerde StringBuilder was om een baseline voor de performance meting te leveren. Ik wilde bovendien zien of de optimalisaties enig effect zouden hebben op de performance van StringBuilder. Zoals je kunt zien, blijft deze constant gedurende de hele benchmark. Aangezien deze vlaggen direct gericht zijn op het optimaliseren van het gebruik van locks, is dit resultaat zoals ik verwachtte. Aan de andere kant van het performance spectrum kunnen we zien dat het gebruik van de gesynchroniseerde StringBuffer zonder de optimalisaties ongeveer drie keer trager is. Dat vind ik een groot verschil.
Van links naar rechts laten de resultaten van Figuur 4 een flinke toename in performance zien die kan worden toegeschreven aan EliminateLocks. Echter, die toename valt in het niet vergeleken bij de toename door biased locking. In feite krijgt elke run met biased locking aan, een even grote performance toename, met uitzondering van kolom C. Wat is er met kolom C?
Tijdens het verwerken van de ruwe data viel me op dat één op de zes runs met alleen UseBiasedLocking significant langer duurde. Het verschil was groot genoeg dat het leek alsof de bench de resultaten rapporteerde van twee compleet verschillende optimalisaties. Na enig nadenken besloot ik om de hogere en lagere waarden apart te rapporteren (runs B & C.) Bij gebrek aan een dieper onderzoek, kan ik alleen maar speculeren dat er twee mogelijke optimalisaties kunnen worden toegepast en dat er één of andere race conditie bestaat binnen Hotspot waarbij biased locking meestal wint, maar niet altijd. Als de andere optimalisatie wint, wordt biased locking niet, of vertraagd toegepast.
Het opmerkelijke resultaat wordt geproduceerd door Escape Analysis. Gegeven het single threaded karakter van dit experiment, verwachtte ik helemaal dat Escape Analysis de lock zou verwijderen en daarmee StringBuffer net zo snel zou maken als StringBuilder. Niets was echter minder waar: het is niet gebeurd. Nog een probleem: metingen op mijn machine verschilden flink van run tot run. Bovendien had ik meerdere collega's gevraagd om tests uit te voeren op hun systemen en de resultaten daarvan verschilden in enkele gevallen veel van de mijne. Bij sommigen leverde alle optimalisaties veel minder op en was het verschil tussen StringBuffer en StringBuilder ook veel kleiner.

Nader onderzoek noodzakelijk

Omdat er zoveel verschillende resultaten waren, kwam ik tot de conclusie dat ik niet alles begreep en ik iets miste. Nader onderzoek was dus noodzakelijk. Toen ik de resultaten nog eens goed bestudeerde, bleek dat er twee typen resultaten waren: resultaten als de mijne op Intel dual core CPU's, en hele andere op single-core CPU's.

Single-core resultaten

Op de oudere single core machines bleek het benchmark resultaat voor StringBuffer maar weinig te verschillen van die van StringBuilder. Doordoor is de winst door de optimalisaties dus ook minder. Om erachter te komen of het echt aan het aantal cores lag, heb ik één van de twee cores van mijn Intel Core 2 Duo uitgeschakeld in de BIOS. Hiermee was er maar één core actief in mijn systeem. De resultaten van deze configuratie zijn te zien in het volgende figuur.



Figuur 5. Benchmark resultaten voor single-core; tijd in ms. voor de drie gevallen.

Net als bij de multi-core resultaten geldt de baseline voor de situatie zonder optimalisaties. Nu bleef StringBuilder performance ook weer gelijk voor alle drie de gevallen. Opvallend is nu echter dat StringBuffer slechts een beetje trager is dan StringBuilder en dat de EliminateLocks met UseBiasedLocking optimalisaties dit verschil nog verder marginaliseren. Ook in deze test heeft escape analysis geen effect. Deze test levert de informatie die een beeld oplevert van wat we nu werkelijk meten met deze benchmark.
In een multi-core wereld ziet het delen van gegevens tussen threads er heel anders uit. Alle moderne CPU's moeten een lokale geheugen cache benutten om de tijd die het kost om instructies en gegevens uit het geheugen op te halen te minimaliseren. Als we een lock gebruiken, introduceren we een zogenaamde 'memory barrier' instructie. Dit is een signaal voor de CPU om te gaan coördineren met de andere CPU's om de meest up-to-date waarde te gaan lezen. Hiervoor communiceren de CPU's met elkaar. Dit zorgt ervoor dat elke processor de actieve applicatie threads een tijdje zal laten wachten.

De resultaten waren zo verschillend dat er nader onderzoek nodig is.

Hoe lang dat wachten duurt, is afhankelijk van het memory model van de CPU. Conservatievere memory modellen zullen meer thread-safe zijn, maar zullen meer tijd nodig hebben om te coördineren tussen de cores. Als je met de Core 2 Duo de tweede core aanzet, neemt de tijd van de StringBuffer baseline benchmark toe van 3731 ms. naar 6574 ms. De benchmark wordt dus een factor 176% trager! Het is duidelijk dat alle hulp die Hotspot kan bieden om dit te optimaliseren welkom is. De optimalisaties kunnen de performance van onze applicaties op multi-core CPU's duidelijk verbeteren.

En escape analysis?

De optimalisatie die niet bleek te werken in onze benchmark was lock elision na escape analysis. Er wordt al jaren over geschreven, maar dit is een techniek die pas recentelijk is geïmplementeerd in Hotspot. Navraag bij de performance lead van HotSpot leerde mij dat het wel werkt voor bepaalde eenvoudige gevallen. Een eenvoudig geval is bijvoorbeeld beschreven in: http://blog.nirav.name/2007_02_01_archive.html. Ook kan ik bevestigen dat het werkt zoals beloofd in de Monte Carlo SciMark2 benchmark (zie http://math.nist.gov/scimark2/index.html.) Omdat de implementie nog niet perfect werkt, wordt lock elision slechts conservatief, in de meest eenvoudige gevallen, toegepast. In komende versies of subversies van Java kunnen we hier verbetering verwachten.

Conclusies

Toen ik aan deze oefening begon om de effectiviteit van Hotspot's lock optimalisaties te beoordelen, dacht ik dat het een paar uur werk zou zijn. Ik zou er dan een leuke blog over kunnen schrijven. Maar zoals vaker met benchmarks, het checken en variëren van de benchmark zelf en het interpreteren van de resultaten duurden veel langer. Dankzij de medewerking van ander experts op dit gebied ben ik tot deze opvallende resultaten, conclusies en dit artikel gekomen.

Als je performance van belang vindt en je wilt een multi-threaded applicatie op een multi-core machine draaien (en wie wil dat niet?), raad ik aan dat je continue meegaat naar de meest recente versie van de JDK. Veel optimalisaties komen ook beschikbaar in sub releases van vroegere Java versies, maar niet allemaal en niet by default. Zo zijn UseBiasedLocking en EliminateLocks default actief in Java 6 en zijn sinds update 6 ook beschikbaar in JDK 1.5, alleen staan ze dan default uit.

Als je een applicatie draait op een multi-core machine die voornamelijk single-threaded werkt, kan het zo zijn dat deze applicatie sneller wordt als je de tweede core, en de eventueel meer aanwezige cores, uitschakeld in het BIOS. De meerdere cores zitten dan eigenlijk alleen maar in de weg.

Op het lagere niveau is de locking overhead veel lager op een single-core dan op een multi-core machine. Inter-core coördinatie, namelijk memory barrier semantiek, heeft op multi-core duidelijk zijn overhead. Dit hebben we aangetoond met de benchmark op mijn systeem met de standaard configuratie tegenover een configuratie met slechts één core actief. Het is duidelijk dat we de threading optimalisaties hard nodig hebben om deze multi-core overhead te reduceren. Ik hoopte dat escape analysis de overhead helemaal kon wegnemen, maar zover is het helaas nog niet.

Concluderend, de beste benchmark voor jou is jouw applicatie op jouw hardware. Het beste wat dit artikel kan bieden is inzicht in wat er gebeurt op het gebied van locking, en handvatten om mee te werken als jouw multi- of single-threaded performance niet is zoals je het graag zou zien.

Uitvoeren van de benchmark

Ik heb de benchmark uitgevoerd op mijn 32 bit Windows Vista laptop met een Intel Core 2 Duo en Java 1.6.0_07. De optimalisaties zijn beschikbaar in de server VM. Dit is niet de default VM op het 32 bit Windows platform. De server VM is zelfs niet beschikbaar vanuit de JRE, slechts vanuit de JDK. Om de server VM te gebruiken, geef ik –server mee op de command line. De drie gebruikte opties zijn:

-XX:+DoEscapeAnalysis, off by default
-XX:+UseBiasedLocking, on by default
-XX:+EliminateLocks, on by default

Om de benchmark uit te voeren: compileer de broncode en gebruik een command als:
java -server -XX:+DoEscapeAnalysis LockTest. «

Referenties

- Dr. Cliff Click, voormalig lead architect van Sun's server VM en momenteel werkzaam bij Azul Systems.
- Kirk Pepperdine, Java performance autoriteit.
- David Dagastine, Sun JVM performance team lead.
- Java concurrency in practice, Brian Goetz et all.
- Java theory and practice: Synchronization optimizations in Mustang, http://www.ibm.com/developerworks/java/library/j-jtp10185/
- Did escape analysis escape from Java 6, http://blog.xebia.com/2007/12/21/did-escape-analysis-escape-from-java-6
- Dave Dice's Weblog, http://blogs.sun.com/dave/entry/biased_locking_in_hotspot
- Java SE 6 Performance White Paper http://java.sun.com/performance/reference/whitepapers/6_performance.html