Zur Performanz der Überwachung von Methodenaufrufen mit der Java Platform Debugger Architecture (JPDA)

Katharina Mehner

Kurzfassung
Die Java Platform Debugger Architecture (JPDA) *bietet komfortable Schnittstellen für die Überwachung von laufenden Java-Programmen. Mit der JPDA können z.B. Debugger, Tracing- und Monitoring-Werkzeuge implementiert werden. Für den Einsatz der JPDA ist häufig ihre Performanz entscheidend. Daher wurde der* Laufzeit-Overhead *für die Überwachung von* Methodenaufrufen *gemessen, wie sie in Tracing- und Monitoring-Werkzeugen vorkommt. Der Laufzeit-Overhead ist nicht unerheblich ist, aber er kann durch geschickte Wahl der Schnittstellen und weiterer Parameter reduziert werden.*

1 Einleitung

Die *Java Platform Debugger Architecture (JPDA)* ist seit dem SDK 1.3 fester Bestandteil der Java 2 Plattform [1,2]. Sie bietet komfortable Schnittstellen für die Überwachung von laufenden Java-Programmen, auch von nebenläufigen und verteilten Java-Programmen. Über die JPDA können Java Virtual Machines verschiedener Versionen und verschiedener Hersteller auf verschiedenen Plattformen zusammen arbeiten. In erster Linie ist die JPDA für die Entwicklung von Debuggern gedacht. Die meisten Debugger innerhalb kommerzieller Entwicklungsumgebungen verwenden diese Architektur, z.B. JBuilder [3].

Bisher weniger verbreitet ist die Verwendung der JPDA für andere Formen der Überwachung, wie z.B. Tracing oder Monitoring. Beim Tracing wird die Ausführung eines Programms in einer Datei protokolliert und bei einigen Werkzeugen anschließend visualisiert, z.B. Jinsight [4]. Beim Monitoring wird ein Programm zur Laufzeit analysiert, z.B. auf Fehler, und gegebenenfalls wird darauf reagiert, z.B. mit Abbruch des Programms. Häufig werden beide Ansätze auch kombiniert, z.B. JaVis [5,6].

Während der Ausführung eines Java-Programms führt die JPDA zusätzlichen Code aus, um Informationen über die Programmausführung zu sammeln und bereitzustellen. Daher werden die überwachten Programme durch den Einsatz der JPDA langsamer. Es ist wichtig, sich mit dieser Tatsache auseinanderzusetzen, denn die Verlangsamung kann nicht immer hingenommen werden und wird oft zum Problem. Je nach Anwendungsgebiet werden verschiedene Anforderungen gestellt, wie sehr die Ausführung eines Programms verlangsamt werden darf. Bei interaktiven Programmen muss darauf geachtet werden, dass die Antwortzeiten für Benutzer akzeptabel bleiben. Manchmal werden Programme über einen längeren Zeitraum hinweg überwacht oder es werden viele Testläufe überwacht. Dann muss die Verlangsamung eingeplant werden.

Nach unserem Wissen gibt es seitens der Hersteller von Java-SDKs keine allgemein zugänglichen Laufzeitmessungen zur JPDA. Im Folgenden wird daher untersucht, in welchem Maße die JPDA die Laufzeit der überwachten Programme verlängert. Dies wird am Beispiel der Überwachung von Methodenaufrufen untersucht, die sowohl bei der JPDA wie auch beim Tracing und Monitoring eine zentrale Rolle spielen. Zu diesem Zweck werden mit der JPDA verschiedene Testapplikationen implementiert, mit denen Beispielprogramme überwacht werden. Es wird gemessen, um wie viel die Beispielprogramme durch die Überwachung

langsamer werden. Dabei werden die SDK Versionen 1.3 und 1.4 von Sun Microsystems [1] auf den Betriebssystemen Windows XP Professional und Linux 2.4 (SuSE 7.1) untersucht. Die Messungen sollen neben den Größenordnungen der Verlangsamung auch zeigen, welche Parameter die Verlangsamung beeinflussen.

Der Beitrag ist wie folgt aufgebaut. Abschnitt 2 gibt einen Überblick über die JPDA. In Abschnitt 3 beschreiben wir den Einsatz der JPDA für die Überwachung von Methodenaufrufen. In Abschnitt 4 beschreiben wir die Testapplikationen, deren Einfluss auf Beispiele wir messen wollen. Diese Beispiele beschreiben wir in Abschnitt 5. In Abschnitt 6 folgen die Messergebnisse. Eine Auswertung erfolgt in Abschnitt 7. In Abschnitt 8 geben wir ein kurzes Fazit.

2 Die JPDA

Über die JPDA kann ein bereits laufendes Java-Programm überwacht, d.h. beobachtet, untersucht und gesteuert werden. Die JPDA ermöglicht dies durch Interaktion mit der Java Virtual Machine (JVM), auf der das überwachte Programm läuft. Um ein Programm mit der JPDA zu überwachen, muss weder sein Quellcode noch sein Bytecode verändert werden. Lediglich beim Start müssen besondere Optionen gesetzt werden. Die JPDA besteht aus zwei Interfaces und einem Protokoll [2], siehe Abbildung 1:

1. Das *Java Virtual Machine Debug Interface (JVMDI)* ist ein Interface, das von jeder JVM implementiert wird. Das JVMDI definiert Mechanismen zur Beobachtung eines laufenden Programmes, zum Zugriff auf den Zustand eines Programms und zur Steuerung des Programms, das von der JVM ausgeführt wird. Dieses Interface wird in der Regel in der Programmiersprache angeboten, in der die JVM implementiert wurde, weshalb es als natives Interface bezeichnet wird. Im SDK 1.3 und 1.4 von Sun Microsystems ist das JVMDI ein C-Interface.
2. Das Java Debug Wire Protocol (JDWP) ist ein standardisiertes Protokoll, um das JVMDI von einem anderen Rechner aus (remote) zu benutzen. Der Teil der JVM, der für die Kommunikation gemäß JDWP verantwortlich ist, wird als Backend bezeichnet. Das Backend leitet Anfragen an das JVMDI weiter und sendet die Antworten des JVMDI zurück.
3. Das Java Debug Interface (JDI) ist ein Java-API zur Überwachung eines laufenden Programms. Das JDI ist in Java implementiert und kommuniziert über das JDWP mit dem JVMDI der JVM des überwachten Programms. Die Implementierung des JDI wird als Frontend bezeichnet. Das Programm, welches das JDI benutzt, und das Frontend laufen auf einer anderen JVM als das überwachte Programm. Das JDWP kann über Shared Memory oder über eine TCP-Verbindung genutzt werden. Der Transport-Mechanismus kann über das JDI ausgewählt werden.

Das JVMDI und das JDI bieten im Wesentlichen die gleiche Funktionalität an. Zum einen bieten sie die Möglichkeit an, auszuwählen, was beobachtet werden soll. Dazu definieren sie eine Menge von Ereignissen, die in einem laufenden Programm auftreten, über die das überwachende Programm informiert werden kann. Zum anderen definieren sie Möglichkeiten, wie das überwachte Programm gesteuert werden kann. Im Folgenden gehen wir auf einige der Ereignisse und Steuerungsmöglichkeiten näher ein.

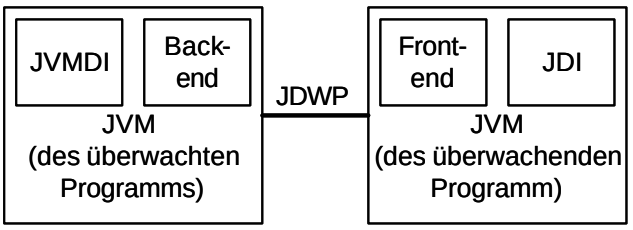


Abbildung 1: Java Platform Debugger Architecture

3 Überwachung von Methodenaufrufen

Mit den Interfaces der JPDA können Applikationen geschrieben werden, um Programme zu überwachen. Entsprechend der gerade vorgestellten Funktionalität der JPDA muss für das überwachende Programm festgelegt werden, welche Ereignisse überwacht werden und wie das Programm gesteuert wird. In diesem Abschnitt beschreiben wir die Vorgehensweise, die für JVMDI und JDI im Wesentlichen gleich ist.

3.1 Die Ereignisse Methodeneintritt und -austritt

Bei vielen Tracing- oder Monitoring-Anwendungen ist es wichtig, das funktionale Verhalten zu überwachen. Bei einem objektorientierten Programm versteht man darunter die Interaktion zwischen den Objekten in Form von Nachrichten bzw. Methodenaufrufen. Für die Beobachtung eines Methodenaufrufs brauchen wir Informationen über das aufrufende Objekt, das aufgerufene Objekt und über den Aufruf selber. Zu dieser Anforderung müssen die passenden Ereignisse der JPDA ausgewählt werden.

Zu den Ereignissen, die mittels der JPDA in einem laufenden Programm überwacht werden können, gehören Methodeneintritt und -austritt, die zusammen einen Methodenaufruf ergeben. Zusammen mit einem Ereignis wird zusätzliche Information über das Ereignis übermittelt. Als nächstes ist zu untersuchen, ob diese Informationen ausreichen, um Interaktion zwischen Objekten, wie oben gefordert, zu beschreiben. Mit dem Ereignis Methodeneintritt werden folgende Informationen übermittelt:

1. Name der aufgerufenen Methode,
2. Formal- und Aktualparameter,
3. Java-Klasse, aus deren Schnittstelle die Methode aufgerufen wurde,
4. Referenz auf den nebenläufiger Thread, der den Methodenaufruf ausführt, und seine Java-Klasse

Mit dem Ereignis Methodenaustritt werden dieselben Informationen übermittelt, lediglich die aufgerufene Java-Klasse fehlt. Diese Information lässt sich aber aus dem zugehörigen vorangegangenen Methodeneintritt ermitteln. Mehr ins Gewicht fällt die Tatsache, dass zwar die Klasse des aufgerufenen Objekts übermittelt wird, aber nicht die Identität, z.B. in Form einer Referenz auf das aufgerufene Objekt. Die Referenz auf das aufgerufene Objekt ist aber nötig, um die Interaktion von Objekten vollständig zu beschreiben. Auch die Referenz auf das Objekt, von dem aus der Aufruf ausgeht, fehlt. Diese entspricht aber der Referenz auf das aufgerufene Objekt vom vorhergehenden Methodeneintritt. Der nächste Unterabschnitt beschreibt, wie man diese Information bestimmen kann.

3.2 Steuerung

Nach der Festlegung, was überwacht werden muss, ist zu entscheiden, wie die Überwachung abläuft. Die überwachende Applikation soll über alle Methodenein- und -austritte in dem überwachten Programm informiert werden. Über die Interfaces der JPDA kann sich die

überwachende Applikation für diese Ereignisse registrieren. Tritt ein registriertes Ereignis auf, wird die überwachende Applikation aufgerufen, um das Ereignis zu behandeln. Darüber hinaus kann die überwachende Applikation entscheiden, ob das überwachte Programm bei Auftreten eines Ereignisses weiterläuft oder anhält (Suspend-Modus). Nur im Suspend-Modus können weitere Informationen als die mit dem Ereignis mitgelieferten bestimmt werden. Dazu bieten die Interfaces Methoden, um den Programmzustand abzufragen. So kann z.B. die oben noch fehlende Referenz auf das aufgerufene Objekt bestimmt werden. Danach kann die Applikation das Programm weiterlaufen lassen. Diese Technik ermöglicht es auch, das Programm schrittweise auszuführen oder Breakpoints zu setzen, und wird daher für interaktives Debuggen eingesetzt. Die Verwendung des Suspend-Modus verlangsamt das überwachte Programm natürlich zusätzlich.

3.3 Verwendung von Filtern

Registriert sich die überwachende Applikation für Methodenaufrufe, so wird sie auch über die Methodenaufrufe in den von Java importierten Paketen informiert. Diese Aufrufe sind unverständlich, da der Quellcode nicht vorliegt. Außerdem erhöhen sie die Anzahl der Ereignisse, auf die die überwachende Applikation reagieren muss, auch wenn diese dann entscheidet, sie zu ignorieren. Damit die überwachende Applikation diese Ereignisse gar nicht erst behandeln muss, bieten JVMDI und JDI die Möglichkeit, Methodenaufrufe aus beliebigen Paketen und Klassen herauszufiltern und damit die Anzahl der vorkommenden Ereignisse zu reduzieren. Der Einsatz von Filtern kann die Überwachung von Programmen zusätzlich langsamer machen, da durch Filter in der JPDA eine Fallunterscheidung nötig wird.

4 Die Testapplikationen

Bisher haben wir die Mechanismen der JPDA beschrieben, um Methodenaufrufe zu überwachen. Sie gelten sowohl für das JVMDI wie für das JDI. Für beide Interfaces werden verschiedene Testapplikationen entwickelt, deren Einfluss auf ein überwachtes Programm gemessen wird. Ziel der Messung ist es, den Laufzeit-Overhead eines durch die JPDA überwachten Methodeneintritts oder –austritts gegenüber einem Methodenaufruf in einem nicht überwachten Programm festzustellen. Dabei kann der Overhead nicht nur von der Wahl des Interfaces abhängen, sondern auch noch vom Vorhandensein von Filtern und von der Wahl der Steuerung. Die Testapplikationen berücksichtigen alle möglichen Kombinationen von Interfaces, Filtern und Steuerung.

Abgesehen von den gerade beschriebenen Varianten sind die Testapplikationen gleich aufgebaut. Sie registrieren sich für die Ereignisse Methodeneintritte und –austritt. Bei Auftreten eines Ereignisses wird lediglich die mitgelieferte Information über das Ereignis abfragt. Die Anzahl der Ereignisse wird zählt. Es werden keine Analysen durchgeführt und keine Informationen in Dateien ausgegeben. Die Testapplikationen sind somit minimal ausgelegt, denn es soll nur der Overhead der JPDA gemessen werden, nicht aber zuätzlicher Code einer Testapplikation. Das Zählen der Ereignisse erlaubt es, aus einer Messung des gesamten Overheads den Overhead pro Methode abzuschätzen.

Als Beispiel für einen Filter wird die Herausfilterung aller Java-Pakete gewählt. Jede Testapplikation läuft einmal mit Filter und einmal ohne. Bei der Steuerung haben wir eine Einschränkung bezüglich des JVMDI vorgenommen. Normalerweise erfordert die Verwendung des JVMDI keine eigenständige Testapplikation. Die Überwachung wird in demselben Prozess ausgeführt wie die JVM des überwachten Java-Programms. Eine Verwendung des Suspend-Modus würde das Erzeugen zusätzlicher Prozesse oder Threads erfordern. Um eine allgemeine Aussage treffen zu können, hätten wir wiederum verschiedene Varianten implementieren müssen. Darauf haben wir im Rahmen dieser Messungen zunächst verzichtet und daher für das JVMDI keinen Suspend-Modus gemessen.

Beim JDI ist die Testapplikation eigenständig, so dass wir ohne Aufwand den Suspend-Modus verwenden können. Beim JDI muss aber bezüglich des JDWP eine Realisierung gewählt werden. Hier haben wir uns auf Ports beschränkt, weil sie ein universelles Kommunikationsmittel darstellen. Sie erlauben es, im Gegensatz zu Shared Memory, auf sehr einfache Weise sowohl lokale wie verteilte Programme zu überwachen.

Wir zeigen hier keinen Quellcode. Eine Testapplikation umfasst zwischen 300 und 600 Zeilen Code. Ingesamt werden sechs Applikation für die Messungen implementiert, unter Berücksichtigung der gerade beschriebenen Varianten:

1. JVMDI ohne Suspend
2. JVMDI ohne Suspend mit Filter
3. JDI ohne Suspend
4. JDI ohne Suspend mit Filter
5. JDI mit Suspend
6. JDI mit Suspend mit Filter

5 Auswahl von Beispielprogrammen

In diesem Abschnitt erläutern wir die gewählten Beispielprogrammen, deren Überwachung gemessen wird. Es ist interessant, ob und in wieweit sich die JPDA unterschiedlich auf sequentielle und nebenläufige Programme auswirkt. Daher messen wir die Auswirkung der sechs Testapplikationen auf ein sequentielles und auf ein nebenläufiges Beispielprogramm.

Die Zeitmessung erfolgt jeweils in dem überwachten Programm selbst. Die Laufzeit wird gemessen, indem im Code die Systemzeit abgefragt und am Ende der Messung ausgegeben wird. Die Zeitmessungen werden für einen Ausschnitt aus einem Java-Programm vorgenommen, in dem keine Methodenaufrufe in Java-Paketen vorkommen.

5.1 Das sequentielle Beispiel

Das sequentielle Beispiel berechnet, ob eine natürliche Zahl n teilbar ist oder nicht. Geprüft werden alle natürlichen Zahlen kleiner gleich n, um viele Methodenaufrufe zu erzeugen. Die Zeitmessung beginnt nach der Initialisierung der Variablen und endet direkt nach der Berechnung, siehe die Kommentare in Listing 1. Das Programm erzeugt bei der Überwachung mit der JPDA im gemessenen Zeitraum 20020 Ereignisse, und zwar 10010 Methodeneintritte und 10010 Methodenaustritte. Das ergibt 10010 Methodenaufrufe.

```
class Main {
  static int events;

  public static void main( String args[] ){
    int n = 10010;
    int d = 0;
    boolean dividable = false;
    long starttime, endtime;
    events = 0;
    starttime = new Date().getTime();    //Begin Zeitmessung
    for( int k = 1; k <= n; k++ ){
      doDiv(n, k, d);
      if( d == 0 )dividable = true;
    }
    endtime =  new Date().getTime();     //Ende Zeitmessung
    System.out.println("Runtime:" +(endtime - starttime)+
                       " Events:" + events );
  }
  static void doDiv(int a, int b, int d){
      events++;//MethodEntryEvent
      d = a/b;
```

```
      events++;//MethodExitEvent
    }
}
```

Listing 1: Sequentielles Beispielprogramm

5.2 Das nebenläufige Beispiel

Das zweite Beispiel ist die Abwandlung des sequentiellen Beispiels in ein nebenläufiges Programm. Es berechnet ebenfalls die Teilbarkeit einer natürlichen Zahl. Zehn Threads testen gleichzeitig alle Zahlen kleiner gleich n. Der Code hierfür ist analog zum sequentiellen Beispiel aufgebaut. Alle Threads bekommen die gleiche Aufgabe, damit sie gleichlange arbeiten und während der Messung die Anzahl der Threads nicht ständig schwankt. Im nebenläufigen Programm beginnt die Zeitmessung nach der Erzeugung aller Threads, bevor sie gestartet werden. Sie endet, nachdem festgestellt wird, dass alle Threads beendet sind. Die Zahl n wurde so gewählt, dass die Anzahl der von der JPDA überwachten Ereignisse genauso groß ist wie im sequentiellen Beispiel, d.h. 10010 Methodeneintritte und 10100 Methodenaustritte.

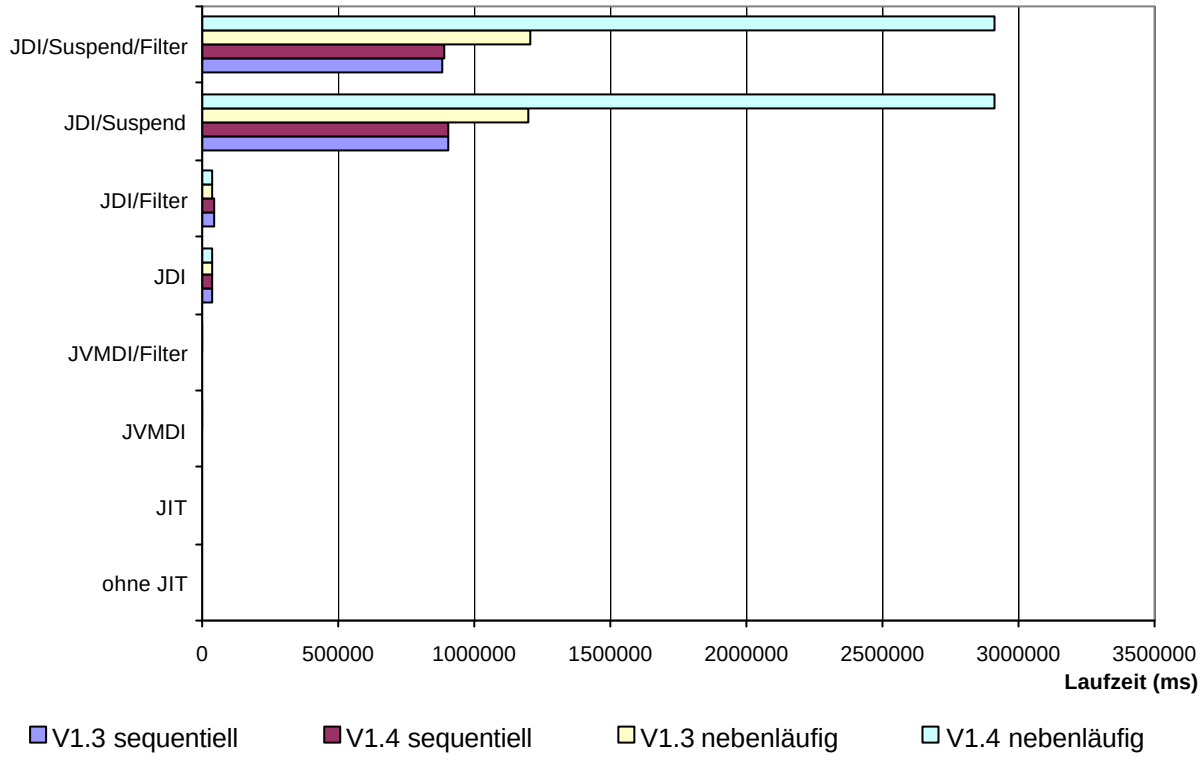
6 Messungen und Ergebnisse

Zur Überwachungen der Beispiele sollen die zuvor beschriebenen sechs Testapplikationen unter den SDKs 1.3 und 1.4 von Sun Microsystems ausgeführt werden, in Verbindung mit den Betriebssystemen Windows XP Professional Linux 2.4 (SuSE 7.1). Alle Messungen werden auf einem Pentium III mit 500 MHz und mit 192 MB RAM durchgeführt.

Die Laufzeit wird auch für das nicht überwachte Beispiel gemessen. Da bei Verwendung der JPDA zur Beobachtung von Methodeneintritten und –austritten der Just-In-Time (JIT) Compiler abgeschaltet wird, wird die Überwachung nicht nur mit der originalen Laufzeit verglichen, sondern auch mit der Laufzeit des Programms ohne JIT Compiler. Die Anzahl der Methodenaufrufe ist in den überwachten Programmen genauso groß wie in den nicht überwachten, was uns einen Vergleich ermöglichen soll.

Die Beispiele laufen unter der gleichen Umgebung wie die Testapplikation. Die Beispiele wurden jeweils 3 Mal mit der gleichen überwachenden Testapplikation ausgeführt und gemessen. Die Messergebnisse wurden arithmetisch gemittelt. Dasselbe gilt für die Messungen ohne jegliche Testapplikation und auch ohne JIT.

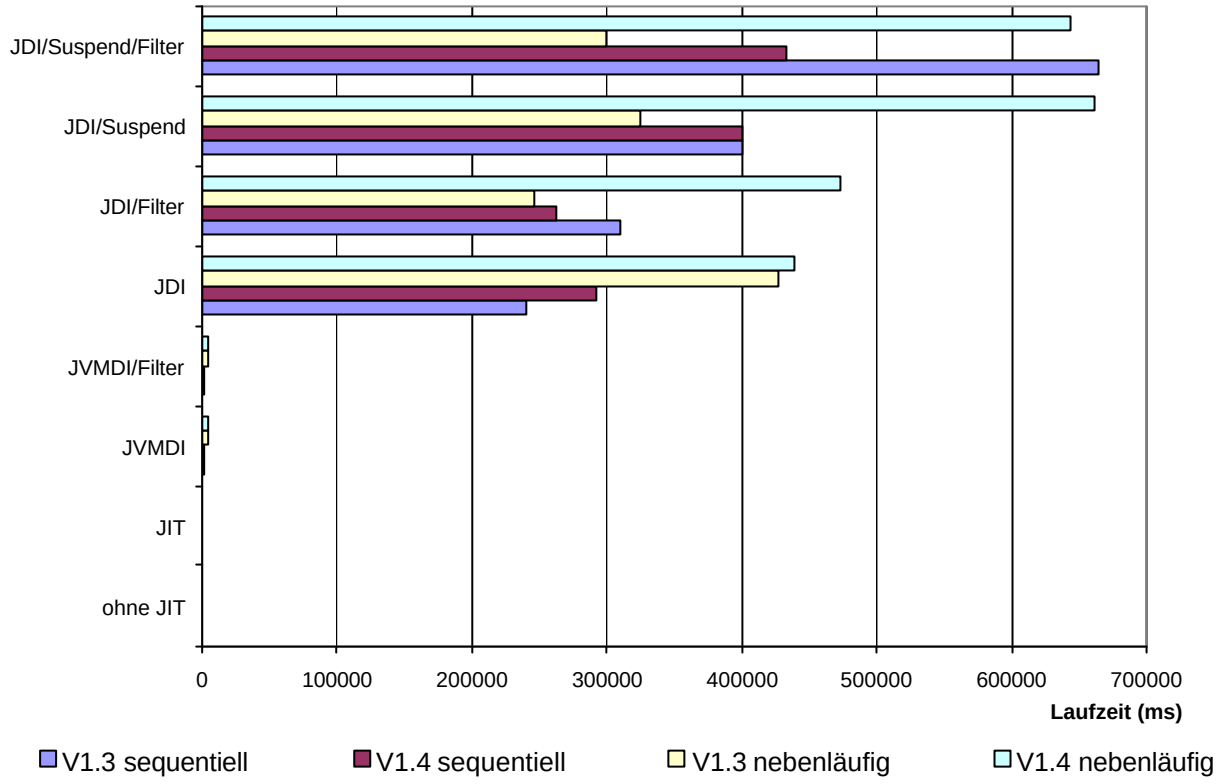
In den folgenden Grafiken (siehe Abbildung 2 und 3) ist die Laufzeit der Beispiele, markiert durch verschiedenfarbige Balken, gegenüber den Testapplikationen aufgetragen. Dabei ist unter Laufzeit der Mittelwert in Millisekunden (ms) angegeben.



Laufzeit (ms)	ohne JIT	JIT	JVMDI	JVMDI/ Filter	JDI	JDI/ Filter	JDI/ Suspend	JDI/ Suspend/ Filter
V1.3 sequentiell	10	7	1041	1031	38248	39811	902861	882259
V1.4 sequentiell	10	3	991	984	39053	41483	902761	889545
V1.3 nebenläufig	10	20	2180	2196	36579	37861	1199410	1203130
V1.4 nebenläufig	10	10	2206	2220	36439	38362	2913068	2914125

Abbildung 2 Messwerte unter Windows XP Professional

Im Folgenden führen wir zunächst einen Vergleich der Verlangsamungen durch. Die gemessene Laufzeit der nicht überwachten Programme ist sehr kurz und wesentlich kürzer als die der überwachten Programme. Hier wirken sich Messfehler oder Messungenauigkeiten sehr viel stärker aus. Es stellt sich heraus, dass der JIT bei den kurzen Programmen nahezu keinen und insbesondere keinen eindeutigen Effekt hat. Beim nachfolgenden Vergleich der Laufzeiten tragen wir dieser Beobachtung Rechnung, indem wir nur die Größenordnungen der Laufzeiten vergleichen. Für die nicht überwachten Programme interpretieren wir die Messwerte zwischen 1 und 10 ms so, dass die Programme eine Laufzeit in der Größenordnung von 10 ms haben. Wenn wir von einem Faktor sprechen, werden wir uns immer auf die 10 ms beziehen. Daher werden wir auch nicht die Unterschiede die sich durch Verwendung des JIT ergeben, berücksichtigen, sondern beim Vergleich von der Laufzeit des nicht überwachten Programms sprechen.



Laufzeit (ms)	ohne JIT	JIT	JVMDI	JVMDI/ Filter	JDI	JDI/ Filter	JDI/ Suspend	JDI/ Suspend/ Filter
V1.3 sequentiell	8	8	1778	1781	240881	310544	400640	664476
V1.4 sequentiell	8	7	1780	1788	292703	262778	400730	432888
V1.3 nebenläufig	14	10	3703	3896	427345	246911	324141	300050
V1.4 nebenläufig	17	13	4235	3839	438528	473131	661060	643670

Abbildung 3 Messwerte unter Linux 2.4 (SuSE 7.1)

6.1 Windows XP Professional

Sequentielles Beispielprogramm

Das überwachte Programm ist unter Verwendung JVMDI um einen Faktor in der Größenordnung von 100 langsamer als das nicht überwachte Programm. Mit dem JDI ohne Suspend ist die Überwachung um einen Faktor in der Größenordnung von 4000 langsamer als das nicht überwachte Programm, mit dem JDI mit Suspend sogar um den Faktor 100.000. Filter ergeben keine signifikanten Unterschiede. Dass die Beispielprogramme mit Filtern mal geringfügig schneller und mal geringfügig langsamer waren, kann man als Messfehler ansehen. Es gibt keinen Unterschied zwischen den SDK Versionen 1.3 und 1.4.

Nebenläufiges Beispielprogramm

Das überwachte Programm ist beim JVMDI um Faktor 200 langsamer als das nicht überwachte Programm, beim JDI ohne Suspend ist es um Faktor 3700 langsamer. Bei Version 1.3 ist das JDI mit Suspend um Faktor 100000 langsamer, bei 1.4 sogar um Faktor 290000. Es wurden keine weiteren Unterschiede zwischen den SDK Versionen 1.3 und 1.4 festgestellt. Auch Filter ergeben keine signifikanten Unterschiede.

Vergleich von sequentiellem und nebenläufigem Beispiel

Die Laufzeit von nebenläufigen und sequentiellen Beispielen ohne Überwachung fällt in die gleiche Größenordnung. Daher können wir die Verlangsamungen unter Überwachung direkt

mit einander vergleichen. Beim SDK 1.3 und 1.4 ist das nebenläufige Beispiel mit dem JVMDI um Faktor 2 schlechter als das sequentielle Beispiel. Beim SDK 1.3 und 1.4 ist die Verlangsamung mit dem JDI ohne Suspend beim nebenläufigen Beispiel genauso groß wie beim sequentiellen. Beim SDK 1.3 mit JDI mit Suspend ist das nebenläufige Beispiel in der gleichen Größenordnung langsamer als das sequentielle Beispiel, aber beim SDK 1.4 mit JDI mit Suspend ist das nebenläufige Beispiel um Faktor 3 langsamer als das sequentielle Beispiel.

6.2 Linux 2.4 (SuSE 7.1)

Sequentielles Beispielprogramm
Das überwachte Programm ist beim JVMDI um einen Faktor in der Größenordnung von 200 langsamer als das nicht überwachte Programm. Mit dem JDI ohne Suspend ist die Überwachung um einen Faktor in der Größenordnung von 30.000 langsamer als das nicht überwachte Programm, mit dem JDI mit Suspend ohne Filter sogar um den Faktor 40.000. Beim JDI mit Suspend und Filter ist das SDK 1.3 sogar um Faktor 60.000 langsamer, das SDK 1.4 aber genauso schnell wie ohne Filter, also um Faktor 40.000 langsamer

Nebenläufiges Beispielprogramm
Das überwachte Programm ist mit dem JVMDI um Faktor 300-400 langsamer als das nicht überwachte Programm, mit dem JDI ohne Suspend ist es um Faktor 40.000 langsamer. Beim JDI mit Suspend mit Filter für Version 1.3 erhalten wir einen niedrigeren Faktor von ca. 30.000. Das deuten wir eher als Messfehler, da einzelne Messwerte auch den Faktor 40.000 erreichten. Für das JDI mit Suspend erhalten wir bei 1.3 einen Faktor von 30.000, für 1.4 aber einen Faktor von 60.000. Abgesehen von dem erwähnten Fall ergeben Filter keine signifikanten Unterschiede.

Vergleich von sequentiellem und nebenläufigem Beispiel
Zunächst sind die nicht überwachten nebenläufigen Beispiele geringfügig langsamer als die sequentiellen, fallen aber dennoch in die gleiche Größenordnung. Beim SDK 1.3 und 1.4 ist das nebenläufige Beispiel mit dem JVMDI um Faktor 2 schlechter als das sequentielle Beispiel. Beim JDI ohne Suspend fällt die Verlangsamung für nebenläufige Beispiele in dieselbe Größenordnung wie für sequentiellen Beispiele. In der Version 1.3 fällt die Verlangsamung für JDI mit Suspend in die gleiche Größenordnung wie für sequentielle Beispiele. Lediglich beim JDI mit Suspend in Version 1.4 wird das nebenläufige Beispiel 1,5fach mehr verzögert als das sequentielle.

6.3 Overhead pro Methodenaufruf

Als letztes wird der Overhead für einen einzelnen Methodenaufruf bestimmt. Qualitativ ergeben sich dadurch keine neuen Erkenntnisse. Ist die Anzahl von Methodenaufrufen in einem Programm bekannt, kann man direkt einen ungefähren gesamten Overhead ableiten.

Wir nehmen an, dass sich unter der Überwachung die Ausführung von Code ohne Methodenaufrufe nicht wesentlich verlangsamt. Die Laufzeit des nicht überwachten Programms ist wesentlich kleiner ist als die eines überwachten Programms. Daher ist es nicht notwendig, eine Differenz der Laufzeiten zu betrachten. Die Laufzeit des überwachten Programms wird durch die Anzahl der Methodenaufrufe (10010) dividiert. Dies ergibt den Overhead für Methodenaufruf. Die folgenden Tabellen beschreiben den Overhead pro Methodenaufruf für die verschiedenen Formen der Überwachung.

Overhead pro Methodenaufruf (ms)	JVMDI	JVMDI/ Filter	JDI	JDI/Filter	JDI/ Suspend	JDI/ Suspend/ Filter
V1.3 sequentiell	0,1	0,1	3,82	3,98	90,2	88,14
V1.4 sequentiell	0,1	0,1	3,9	4,14	90,19	88,87
V1.3 nebenläufig	0,22	0,22	3,65	3,78	119,82	120,19
V1.4 nebenläufig	0,22	0,22	3,64	3,83	291,02	291,12

Abbildung 4 Overhead pro Methodenaufruf unter Windows XP Professional

Overhead pro Methodenaufruf (ms)	JVMDI	JVMDI/ Filter	JDI	JDI/Filter	JDI/ Suspend	JDI/ Suspend/ Filter
V1.3 sequentiell	0,18	0,18	24,06	31,02	40,02	66,38
V1.4 sequentiell	0,18	0,18	29,24	26,25	40,03	43,25
V1.3 nebenläufig	0,37	0,39	42,69	24,67	32,38	29,98
V1.4 nebenläufig	0,42	0,38	43,81	47,27	66,04	64,3

Abbildung 5 Overhead pro Methodenaufruf unter Linux 2.4 (SuSE 7.1)

7 Auswertung

Die Messungen sollen erste Anhaltspunkte geben über die Größenordnungen, um die die Programme mit der JPDA langsamer werden. Die Tendenz der Messwerte ist klar erkennbar, auch wenn Abweichungen existieren. Diese sind z.T. Nebeneffekten zuzuschreiben, die in einer komplexen Laufzeitumgebung wie sie bei einer JVM und Kommunikation über Ports gegeben ist, nur mit großem Aufwand vollständig zu beherrschen sind.

Insgesamt ist die Verlangsamung durch die verschiedenen Interfaces erheblich. Das JVMDI ist eindeutig effizienter als das JDI, hier schlägt ein Faktor im Rahmen von 100-200 zu Buche. Beim JDI ohne Suspend ist es schon wesentlich mehr mit einem Faktor von 4000 bei Windows und 30.000 bis 40.000 bei Linux. Der Suspend-Modus verlangsamt das JDI noch mal wesentlich, insgesamt Faktor 40.000 bis 300.000. Die Wahl des Interfaces und die Steuerung spielen also eine erhebliche Rolle. Beim Gebrauch weniger Filter wie im Falle der Java-Pakete muss man beim JVMDI nicht mit Einbussen rechnen und beim JDI mit hoher Wahrscheinlichkeit auch nicht. Filter waren jedoch nicht der Hauptgegenstand der Untersuchung und erfordern zur abschließenden Beurteilung genauere Untersuchungen.

Es gibt keinen wesentlichen Nachteil für nebenläufige Programme. Beim JVMDI sind sie um Faktor 2 langsamer als sequentielle Programme. Beim JDI gab es im Wesentlichen keine Unterschiede zwischen sequentiellem und nebenläufigem Programm. Eine Ausnahme war, dass sich bei Windows und SDK 1.4 unter JDI mit Suspend ein nebenläufiges Programm um Faktor 3 schlechter verhält als ein sequentielles. Die andere Ausnahme war, dass sich bei Linux und SDK 1.4 unter JDI mit Suspend ein nebenläufiges Programm um Faktor 2 schlechter verhielt. Die Verlangsamung für nebenläufige Programme kann bei Einsatz von wesentlich mehr Threads anders aussehen.

Die Untersuchungen zeigen keine wesentlichen Unterschiede zwischen 1.3 und 1.4, abgesehen vom schlechteren Verhalten für nebenläufige Programme unter JDI mit Suspend unter dem SDK 1.4, sowohl bei Windows wie bei Linux. Sowohl in der Version 1.3 wie in der Version 1.4. wird das überwachte Programm im Interpreter-Modus ausgeführt, wenn sich die überwachende Applikation für Methodeneintritte und –austritte registriert hat. Die Performanz wird in 1.4 verbessert, indem generell zur Ausführung des überwachten Programms die HotSpot JVM anstatt des Interpreters benutzt werden. Der Interpreter muss aber weiterhin benutzt werden für Methoden mit Breakpoints, bei der schrittweisen Abarbeitung und wenn Watchpoints gesetzt sind.

8 Fazit

Die Verlangsamung durch die JPDA ist erheblich und kann insbesondere bei Überwachung von interaktiven Programmen Probleme bereiten. Entweder muss man für Überwachung mit Benutzereinsatz viel Zeit einkalkulieren oder man muss sich auf besonders wichtige Programmausschnitte beschränken. Alternativ kann man auf Testreiber ausweichen. Probleme können aber auch entstehen, wenn das überwachte Programm mit anderen Programmen interagiert, die die Verlangsamung nicht akzeptieren.

Die Messungen machen vor allem den Tradeoff zwischen Funktionalität und Performanz deutlich. Das JDI ist nicht nur komfortabler als das JVMDI sondern durch das JDWP auch flexibler was den Ort der überwachenden Applikation angeht. Dadurch geht Performanz verloren. Der Suspend-Modus bietet noch mal mehr Funktionalität zu Lasten der Performanz. Immerhin lässt die JPDA dem Anwender einen relativ großen Spielraum.

Der Fokus der Messungen lag auf dem Einsatz der JPDA für Tracing und Monitoring. Bei Debuggern kommt die Überwachung von Methodenaufrufen in ähnlicher Weise auch zum Einsatz, z.B. beim Überspringen der Abarbeitung einer Methode. Auch andere Funktionen des Debuggers sind ähnlich langsam. Da ein Debugger aber interaktiv genutzt wird und vor allem in der Regel für eine wesentlich geringer Anzahl von Methodenaufrufen als in unserem gesamten Beispielprogramm (10010 Methodenaufrufe), springt die Verlangsamung nicht so sehr ins Auge und ist für den Benutzer tolerierbar.

Trotz der Performanzprobleme ist die Ergänzung der Java-Platform um die JPDA vorteilhaft. Die JPDA bietet einen Standard, nicht nur für Debugger, sondern auch für zahlreiche andere Werkzeuge für die Programmüberwachung, für die sonst jedes Mal eine proprietäre Lösung gefunden werden muss, sei es die Instrumentierung des Quellcodes oder des Bytecodes oder die Abwandlung des Classloaders.

9 Autor

Dipl.-Inform. Katharina Mehner ist seit 1997 wissenschaftliche Mitarbeiterin an der Universität Paderborn in der Arbeitsgruppe Datenbank- und Informationssysteme. Ihre Schwerpunkte sind objektorientierte und aspektorientierte Softwareentwicklung sowie Softwarevisualisierung. Sie promoviert über dynamisches Programmverstehen und visuelles Debugging von nebenläufigen Java-Programmen. (Kontakt: mehner@upb.de, www.upb.de/cs/mehner.html)

10 Literatur

[1] http://java.sun.com/j2se
[2] http://java.sun.com/products/jpda
[3] http://www.borland.com/jbuilder
[4] http://www.research.ibm.com/jinsight
[5] K. Mehner. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. In S. Diehl (Ed.), Software Visualization International Seminar Dagstuhl Castle. Revised Papers. LNCS 2269
[6] B. Weymann. Visualisierung der Synchronisation von Java-Threads mit der Unified Modeling Language. Diplomarbeit, Universität Paderborn, 2000.