Perc Pico

Sebastian Maisch

Seminar: Formale Softwareentwicklung
Institut für theoretische Informatik, Universität Karlsruhe
24. Juni 2008

Inhaltsverzeichnis

1 Was ist Perc Pico? 3

2 Eingebettete Systeme, Echtzeitsysteme 3
2.1 Eingebettete Systeme . . . 3
2.2 Echtzeitsysteme . . . 3
2.3 Harte und weiche Echtzeitanforderungen . . . 3

3 Harte Echtzeit und Java 4
3.1 Anforderugen an ein Echtzeit-Java . . . 4
3.2 Probleme mit Java und Echtzeit . . . 4
3.3 Lösungsansätze der Probleme von Perc Pico . . . 5

4 Speicherverwaltung in Perc Pico 5
4.1 Wie funktioniert das *Thread Stack Memory Model*? . . . 5
4.2 Wie vermeidet Perc Pico *dangling references*? . . . 6
4.2.1 Zugriff einer äußeren Region in eine innere . . . 6
4.2.2 Zugriff von einem übergeordneten Thread in einen „Kindthread“ . . . 6
4.2.3 Zugriff auf beliebigen Thread . . . 7
4.2.4 Zugriff von einem Kindthread in einen übergeordneten Thread . . . 7
4.2.5 Erlaubte Zugriffe . . . 8
4.3 Annotations zur Handhabung des Speichers mit Beispielen . . . 8
4.3.1 `@ScopedMemorySize` und `@ConstructedScopedMemorySize` 8
4.3.2 `@CallerAllocatedResult` . . . 8
4.3.3 `@Scoped` . . . 9

5 Statische Analyse 9
5.1 Allgemeine Erklärung des Systems . . . 9
5.2 Einige wichtige Annotations und Assertions . . . 9
5.2.1 `@StaticAnalyzable` . . . 9
5.2.2 `StaticLimit.InvocationMode` . . . 10
5.2.3 `StaticLimit.IterationBound` . . . 10

6 Perc Pico Architektur 10

1 Was ist Perc Pico?

Bisher wurden Echtzeitsysteme fast ausschließlich mit C/C++ programmiert. Da viele Programmierer heute allerdings deutlich besser mit Java umgehen können als mit C++, und C bei großen Projekten schnell unübersichtlich wird, existiert eine Nachfrage nach einer Möglichkeit, Java zur Programmierung von Echtzeitsystemen heranzuziehen. Dazu gibt es verschiedene Ansätze. Von Sun und IBM wird die RTSJ (Real Time Specification for Java) entwickelt.

Perc Pico ist ein Produkt von Aonix, das möglichst nah an den, gemäß RTSJ spezifizierten, Bedingungen, ebenfalls Java unter Echtzeitbedingungen anbietet. Auf die wichtigsten Eigenschaften von Perc Pico möchte ich im folgenden eingehen.

2 Eingebettete Systeme, Echtzeitsysteme

2.1 Eingebettete Systeme

Als Eingebettete Systeme bezeichnet man Computer, die innerhalb von größeren (Informatik-) Systemen bestimmte Teilaufgaben übernehmen. Sie werden zur Steuerung, Regelung oder Überwachung des Systems eingesetzt.

Meist sind die Resourcen auf diesen Systemen stark beschränkt und die Ausführung der Aufgaben muss in einer bestimmten Zeit geschehen. Man spricht dann auch von Echtzeitsystemen.

Als Beispiele kann man die Steuerung von modernen Waschmaschinen (Wasserzufuhr zur richtigen Zeit etc.), Motorsteuerung im Auto (da ein Versagen des Systems hier im schlimmsten Fall Menschenleben kostet, ist dies ein gutes Beispiel für die Wichtigkeit der fehlerfreien und zeitkritischen Ausführung der Aufgaben).

2.2 Echtzeitsysteme

Alle (Informatik-) Systeme, in denen die fehlerfreie Ausführung der Aufgaben zeitkritisch ist, bezeichnet man als Echtzeitsysteme. Software, die für solche Systeme entwickelt wird unterliegt daher speziellen Anforderungen. Je nach Art des zu lösenden Problems spricht man von harten und weichen Echtzeitanforderungen.

2.3 Harte und weiche Echtzeitanforderungen

Man unterscheidet Echtzeitsysteme grundsätzlich in zwei Gruppen.

In der einen ist eine zeitkritische Ausführung wünschenswert, aber nicht notwendig für den Erfolg. Als Beispiel kann eine Videowiedergabesoftware herhalten. Wird der Zeitrahmen nicht eingehalten, steht das Bild zwar eine kurze Zeit, kann aber dann normal weiterlaufen. Diese Software muss nur *weiche Echtzeitanforderungen* erfüllen. Das bedeutet, dass die Software zwar im Durchschnitt ihre Aufgaben innerhalb des gegebenen Zeitrahmens ausführen kann, in Einzelfällen aber scheitert. Ob eine Software weiche Echtzeitanforderungen erfüllt, wird meist mit einfachem Testen ohne Beweis entschieden.

Im Gegensatz dazu stehen *harte Echtzeitanforderungen*. Um diese zu erfüllen, muss ein formaler Beweiß geführt werden. Die Ausführung der Aufgaben innerhalb eines gewissen Zeitrahmens ist dann nicht mehr nur wünschenswert, sondern notwendig für den Erfolg. Das obige Beispiel der Motorsteuerung bei Autos fällt in diese Gruppe.

Perc Pico richtet sich insbesondere an Entwickler, die mit ihrer Software harte Echtzeitanforderungen erfüllen müssen, und bietet dafür die geeigneten Werkzeuge (unter anderem ein statisches Analysetool, dass dem Programmierer den Beweis abnimmt).

3 Harte Echtzeit und Java

3.1 Anforderugen an ein Echtzeit-Java

An ein Java, das harte Echtzeitanforderungen erfüllt, werden (neben den offensichtlichen) weitere Anforderungen gestellt.

Um die Grundidee von Java auch für Echtzeitsysteme beizubehalten, wird gefordert, dass der geschriebene Quelltext unabhängig von der Zielplatform ist.

Ebenso soll die Syntax von Java nicht verändert werden, um einen leichten Umstieg von „normalem“ Java auf das Java für Echtzeitsysteme zu erleichtern. Syntaxveränderung betrifft hierbei insbesondere das Hinzufügen von Schlüsselwörtern. Kleine Veränderungen durch Einführung von sog. Annotations sind kaum vermeidbar.

Um harte Echtzeitanforderungen gerecht zu werden, muss es Möglichkeiten geben die Korrektheit eines Programms innerhalb bestimmter Grenzen, nämlich der, die das Zielsystem vorgibt, zu überprüfen.

3.2 Probleme mit Java und Echtzeit

Dass eine normale Java-Implementierung unter Echtzeitbedingungen kaum einsetzbar ist, liegt insbesondere an zwei Faktoren.

Die Speicherverwaltung, in Java realisiert durch einen Garbage Collector, ist für den Programmierer kaum durchschaubar oder steuerbar. Weder ist festgelegt, wann er seine Arbeit verrichtet, noch wieviel Zeit er dafür benötigt. Das allerdings ist eine Grundvoraussetzung, dafür ein Programm statisch analysieren zu können. Ebenfalls nötig für die Analyse ist es, zu dem Zeitpunkt an dem sie stattfindet, wissen zu müssen wieviel Speicher zur Laufzeit benötigt wird. Auch das ist mit der normalen Speicherverwaltung von Java nicht möglich, da z.B. die Längen mancher Arrays erst zur Laufzeit feststehen.

Ein weiteres Problem, das formale Beweise unmöglich macht, ist die Laufzeit einzelner Algorithmen. Da, wie schon erwähnt, die Anzahl der Elemente in Arrays oder allgemein Containern nicht während das Programm geschrieben wird begrenzt werden kann, kann auch die Laufzeit von z.B. Sortieralgorithmen über diese nicht vorhergesagt werden. Im Allgemeinen ist es nicht möglich, die exakte Laufzeit von Algorithmen, deren Iterationen oder Rekursionen nicht beschränkt sind, vorherzusagen.

Zusammenfassend kann man sagen, dass Java die Möglichkeit fehlt, Resourcen, die durch die Sprache eigentlich unbegrenzt sind, zu begrenzen, um so einen formalen Korrektheitsbeweis durchführen zu können.

3.3 Lösungsansätze der Probleme von Perc Pico

Um die formale Beweisbarkeit zu ermöglichen, werden von Perc Pico Assertions und Annotations bereitgestellt, die ermöglichen, eben diese Resourcen zu beschränken. Auch unbeschränkte Resourcen sind weiterhin erlaubt, allerdings ist dann keine statische Analyse des Codes mehr möglich. Allerdings sind auch auf Echtzeitsystemen nicht alle Teile des Programms zeitkritisch, so dass der Programmierer entscheiden kann, seinen Code an den notwendigen Stellen an Echtzeitanforderungen anzupassen.

Um sicherzustellen, dass auch der Garbage Collector vorhersehbar ist, benutzt Perc Pico diesen nicht und verzichtet darauf, Speicher im Heap zu verwenden. Stattdessen wird der gesamte Speicher als Stack verwaltet. Damit nicht benötigter Speicher trotzdem frühzeitig freigegeben werden kann, werden sog. *Scopes* eingerichtet. Um Objekte einem Scope zuzuordnen werden diese durch spezielle Annotations gekennzeichnet. Fehlen die Annotations, werden neue Objekte im sog. *Immortal Memory* angelegt und bleiben bis zum Programmende im Speicher. Da Stackspeicher nicht freigegeben werden muss, kann auf neue Schlüsselwörter in der Sprache zum Verwalten des Speichers verzichtet werden. Das von Perc Pico verwendete Speichermodel wird auch *Thread Stack Memory Model* genannt.

4 Speicherverwaltung in Perc Pico

4.1 Wie funktioniert das *Thread Stack Memory Model*?

In Perc Pico erhält jeder Thread seinen eigenen Stack. Dieser wird dabei auf dem Stack des aufrufenden Threads angelegt. Bei Programmstart existiert also ein Stack, der bei Methodenaufruf befüllt und beim Beenden der Methode wieder geleert wird. Von der anderen Seite des Stacks wird dieser mit Objekten des *Immortal Memory* und dem Speicher, der für die Stacks neu gestarteter Threads benutzt wird befüllt[Aon08, The Thread Stack Memory Model]. Außerdem kann man in diesem Bereich eigene *Scopes* erstellen, die sich ebenfalls nach den Prinzipien eines Stacks verhalten. Neben dem Vorteil, die Lebenszeit von Objekten im Verhältnis zueinander bestimmen zu können, ergibt sich durch den stackbasierten Ansatz ein deutlicher Performancegewinn beim Bereitstellen des Speichers. Er entsteht dadurch, dass der Speicher nicht mehr fragmentieren kann. Daher muss bei der Bereitstellung nicht mehr nach passend großen Blöcken gesucht werden, sondern der Platz im Speicher ist durch die Spitze des Stacks festgelegt. Weitere Probleme, die bei der Fragmentierung entstehen (insbesondere Fehlschlagen einer Bereitstellung aufgrund von fehlenden Blöcken der benötigten Größe) treten ebenfalls nicht auf.

Dabei ergeben sich Probleme mit der Sichtbarkeit von Speicherbereichen. Bei Programmen mit nur einem Thread sind diese leicht ersichtlich. Eine *aufrufende* Methode kann den Speicher von *aufgerufenen* Methoden nicht referenzieren. Umgekehrt ist das allerdings möglich. Um jetzt Objekte an Methoden zu übergeben oder von ihnen zu empfangen, gibt es die Möglichkeit Perc Pico mitzuteilen, in welchem Speicher das Objekt liegen soll. Dadurch ist es möglich, Objekte in einer aufgerufenen Methode zu erstellen, und diese in der Hierarchie nach unten durchzureichen, ohne dass Konflikte auftreten.

Deutlich komplizierter ist das Verwalten des Stacks bei mehreren Threads. Wie oben beschrieben, besitzt jeder Thread seinen eigenen Stack, auf dem von ihm verwaltete Objekte liegen. Außerdem finden sich hier die „Unterstacks“ der durch diesen Thread gestarteten Threads. Jeder Thread verwaltet seine eigenen Speicherregionen deren Zugriff er mit Threads weiter oben in der Hierarchie teilt. Referenzen zwischen zwei Kindthreads zu teilen sind nicht möglich, bzw. nur indem man den zugehörigen Speicher in der Hierarchie nach unten schiebt.

Trotz dieser Einschränkungen kann es in einigen Fällen immer noch zu hängenden Referenzen oder *dangling references* kommen.

4.2 Wie vermeidet Perc Pico *dangling references*?

Es gibt vier Fälle von Speicherreferenzierung, die zu *dangling references* führen können. *Dangling references* sind Referenzen, die auf Objekte verweisen, die nicht mehr im Speicher existieren. Ein Zugriff auf diese Referenzen führt in jedem Fall zu undefiniertem Verhalten des Programms. Die einzelnen Zugriffe werden jeweils durch die Abbildungen 1 bis 4 erläutert. In diesen wächst der Stack nach unten und wird von dort auch wieder abgebaut.

4.2.1 Zugriff einer äußeren Region in eine innere

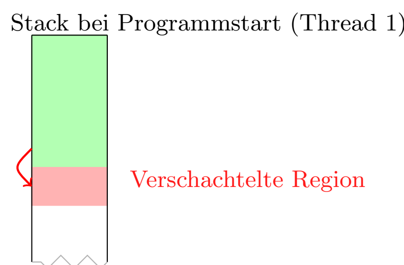

Abbildung 1: Zugriff von außen in eine innere Region

Es ist leicht ersichtlich, dass Zugriffe von Objekten, die in der Speicherhierarchie weiter unten liegen, auf Objekte, die weiter oben liegen in einem stackbasiertem Speichersystem zu Fehlern führen. Objekte, die in der Hierarchie weiter unten liegen, haben eine längere Lebensdauer als Objekte weiter oben. Daher passiert es leicht, dass Referenzen in diese Richtung auf Objekte verweisen, die nicht mehr existieren. Sie sind daher in Perc Pico nicht erlaubt. Abbildung 1 zeigt einen solchen Zugriff.

4.2.2 Zugriff von einem übergeordneten Thread in einen „Kindthread“

Ähnlich verhält es sich beim Zugriff eines übergeordneten Threads auf den Stack eines Kindthreads. Objekte des Kindthreads werden freigegeben, wenn dieser terminiert. Da dies aber immer passiert bevor der aufrufende Thread terminiert, entstehen ebenfalls Probleme mit der Lebensdauer von Objekten auf die in diese Richtung referenziert wird. Auch hier kann nicht garantiert werden, dass die

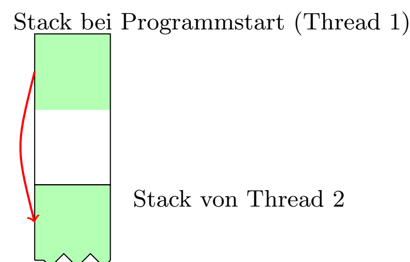

Abbildung 2: Zugriff auf einen Kindthread

Referenzen gültig sind solange sie existieren. Daher sind auch Zugriffe wie in Abbildung 2 in Perc Pico nicht erlaubt.

4.2.3 Zugriff auf beliebigen Thread

Wie bereits im letzten Abschnitt angedeutet, werden Kindthreads immer vor ihren Eltern terminiert. Ein solcher Zusammenhang besteht zwischen zwei unabhängigen Threads nicht. Daher kann auch im Falle einer Referenzierung zwischen zwei beliebigen Threads nicht sichergestellt werden, dass die Referenzen während ihrer Existenz auf gültigen Speicher verweißen. Der in Abbildung 3 gezeigte Zugriff ist deshalb nicht möglich.

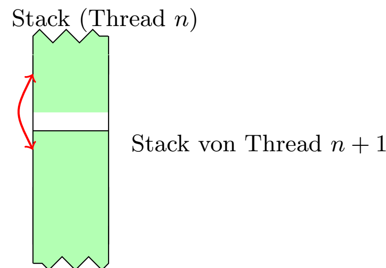

Abbildung 3: Zugriff auf beliebigen Thread

4.2.4 Zugriff von einem Kindthread in einen übergeordneten Thread

Im Gegensatz zu den Vorangegangenen Fällen ist nicht klar ersichtlich, warum es bei dem in Abbildung 4 dargestellten Fall zu Problemen kommen kann. Der übergeordnete Thread wird wie weiter oben erwähnt erst beendet, wenn alle Kindthreads schon beendet wurden. Dadurch sollte es grundsätzlich möglich sein, Speicher dort aus einem Kindthread heraus zu referenzieren. Da allerdings der Stack des übergeordneten Threads weitergeführt wird, während der Kindthread läuft, das heißt gefüllt und wieder geleert wird, sind auch hier keine Garantien möglich, dass der referenzierte Speicher mindestens so lange gültig ist wie die Referenz.

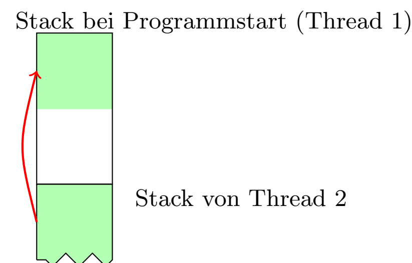

Abbildung 4: Zugriff auf übergeordneten Thread

Um dieses Problem zu umgehen, wird die Speicherfreigabe mit dem terminieren von Threads synkronisiert. Der Speicher eines Threads wird offensichtlich erst dann freigegeben, wenn dieser terminiert ist. Außerdem dürfen durch Speicherfreigabe im übergeordneten Thread die Stackeigenschaften nicht verletzt werden. Das heißt, dass Speicher, der bereitgestellt wurde bevor ein Thread gestartet wurde, auch so lange gültig ist und nicht freigegeben wird, bis der nach der Bereitstellung gestartete Thread terminiert wurde.

Perc Pico kann also garantieren, dass Speicher in einem übergeordneten Thread genau dann länger gültig ist, als die Laufzeit des Kindthreads, wenn dieser Speicher angefordert wurde, bevor der Kindthread gestartet wurde. Auf Speicher, der nach dem Start bereitgestellt wurde, hat ein Kindthread keinen Zugriff. Dieser Speicher liegt dann auch weiter oben auf dem Stack des übergeordneten Threads als der Stack des Kindthreads. Ein Zugriff wie er in Abbildung 4 dargestellt wird, ist also erlaubt.

4.2.5 Erlaubte Zugriffe

Zusammenfassend kann man sagen, dass Referenzierung, die von höher gelegenen Stackeinträgen auf tiefer gelegene Einträge erfolgen bis auf eine Ausnahme erlaubt sind. Die Ausnahme ist der in Abbildung 3 gezeigte Fall, dass zwei unabhängige Threads einander referenzieren.

4.3 Annotations zur Handhabung des Speichers mit Beispielen

In diesem Abschnitt möchte ich ein paar wichtige Annotations vorstellen, die Perc Pico benutzt, um die Handhabung des Speichers zu erleichtern und bestimmte Konstruktionen zu ermöglichen. Eine vollständige Liste findet sich in [Nil04, Appendix B].

4.3.1 `@ScopedMemorySize` und `@ConstructedScopedMemorySize`

Diese beiden Annotations sorgen dafür, dass Perc Pico weiß, wieviel Speicher eine Methode benötigt, und wo dieser angelegt werden soll. Wird er als *ScopedMemory* angelegt, ist es temporärer Speicher, der freigegeben wird, sobald die Methode beendet wird. *ConstructedScopedMemory* wird vom *Thread Stack Memory Model* verwaltet.

4.3.2 `@CallerAllocatedResult`

Eine Methode in Perc Pico, die ein Objekt zurückgibt, kann mit der Annotation `@CallerAllocatedResult` dafür sorgen, dass dieses in einer Stackregion erstellt wird, die die aufrufende Methode festlegt.

Das folgende Beispiel wurde gekürzt aus [Aon08] entnommen.

```
public class Main {
    @ScopedMemorySize(instances = {1}, types = {Main.class})
    public static void main(String args[]) {
        Main aMainObject;
        aMainObject = makeMainObject();
    }

    @CallerAllocatedResult
    @ScopedMemorySize(instances = {1}, types = {Object.class})
    public static Main makeMainObject() {
        Object tmp = new Object();
        return new Main();
    }
}
```

Hier kann man gut erkennen, dass das in `makeMainObject()` erstellte Main-Objekt in einer Speicherregion liegt, die von `main()` verwaltet wird. Im Gegensatz dazu wird das Objekt, das in den von `tmp` referenzierten Speicher erstellt wird, von `makeMainObject()` selbst verwaltet (und damit beim Ende der Ausführung der Methode freigegeben).

4.3.3 `@Scoped`

`@Scoped` wird benutzt, um anzuzeigen, dass eine Methode ein Objekt liefert, das schon fest zugewiesenen Speicher in einer Region hat. Dies ist nur dann problemlos möglich, wenn der Speicher nicht in der Methode, oder einer Methode die sie aufruft angefordert wurde.

5 Statische Analyse

5.1 Allgemeine Erklärung des Systems

Deutlich einfacher als das Speichermanagement ist die statische Analyse des Quelltextes in Perc Pico zu erklären. Wie schon erwähnt, kann man diese bei Bedarf durchführen lassen. Dabei werden einzelne Methoden von einem *Byte-Code-Verifizierer* getestet, und nur dann akzeptiert, wenn dieser obere Schranken für gewisse Werte, wie Ausführungszeit oder Speicherverbrauch finden kann.

Um diese Werte zu finden, stützt sich der Verifizierer auf Annahmen, die der Entwickler im Quelltext macht. So kann man z.B. die Anzahl der Iterationen einer Schleife begrenzen. Um sich bei sehr allgemeinen Algorithmen, wie Sortieralgorithmen nicht zu sehr einschränken zu müssen, kann man jede Methode mit verschiedenen Voraussetzungen analysieren lassen. Bei einem Sortieralgorithmus ist es evtl. sinnvoll Iterationsbeschränkungen, je nach Struktur oder Größe des zu sortierenden Containers, härter oder weicher zu fassen. Falls man keine Aussagen über die Eingangsdaten hat, kann man auch ganz auf die Analyse verzichten.

Um das zu realisieren, kann der Entwickler, je nach Bedarf, verschiedene Analysemodi erstellen, die er dann in der Methode mit passenden Annahmen versieht. Wann welcher Modus zur Anwendung kommt, kann dann ebenfalls der Entwickler bei ihrem Aufruf festlegen.

Was man durch diese Technik allerdings nur schwer, bis gar nicht analysieren lassen kann sind rekursive Methoden. Es ist zwar leicht möglich, dem *Byte-Code-Verifizierer* Annahmen über Schleifeniterationen mitzuteilen, für Rekursionen fehlt diese Möglichkeit allerdings.

5.2 Einige wichtige Annotations und Assertions

Eine vollständige Auflistung der Annotations und Assertions zur statischen Analyse findet sich unter [Nil04, Appendix A]. Dort findet man auch ein übersichtliches Beispiel für diese im praktischen Einsatz.

5.2.1 `@StaticAnalyzable`

Mit dieser Annotation markiert man Methoden, die statisch analysiert werden sollen. Hier gibt man auch die Analysemodi für die Methode mit an. Die beiden wichtigsten Parameter sind hier `enforce_analysis` und `modes`. Ersterer erwartet ein Array von boolschen Werten, die aussagen für welche Modi eine Analyse durchgeführt werden soll. Die Modi selber werden als selbst definierter `enum` an `modes` übergeben.

Beispiel (aus [Nil04]):

```
// ...
    public enum ModeEnumeration {UNBOUNDED, SMALL, MEDIUM, LARGE};
    @StaticAnalyzable(
        enforce_analysis = {false, true, true, true},
        modes = ModeEnumeration.class
    )

// ...
```

5.2.2 `StaticLimit.InvocationMode`

`StaticLimit.InvocationMode` ist eine Assertion, die immer zu `true` evaluiert. Sie wird im Rumpf einer statisch analysierbaren Methode direkt vor den Aufruf einer weiteren Methode gestellt. Übergibt man der Assertion einen Parameter, so wird die aufgerufene Methode in diesem Modus analysiert.

Wenn man den Modus abhängig vom Modus der aufrufenden Methode machen will, muss man zwei Parameter übergeben. Der erste gibt einen möglichen Modus der aufrufenden Methode an, der zweite den Modus der aufgerufenen Methode, unter dem diese analysiert werden soll, wenn der Aufrufer den Modus hat, der im ersten Parameter angegeben ist. Man kann mehrere `StaticLimit.InvocationMode` Assertions vor einen Methodenaufruf stellen um beliebig viele Fälle abzufangen.

5.2.3 `StaticLimit.IterationBound`

Diese Assertion limitiert die Iterationen einer Schleife. Auch hier können ein oder zwei Parameter übergeben werden. Bei einem Parameter bestimmt dieser die Anzahl der Iterationen.

Bei zwei Parametern bestimmt der erste den Modus, unter dem die Schleife auf die Anzahl der Iterationen im zweiten Parameter beschränkt werden soll.

6 Perc Pico Architektur

Zum Ende dieses Artikels soll noch kurz auf die Struktur von Perc Pico eingegangen werden. Perc Pico selbst besteht grundsätzlich aus einem Tool, dem *Pico Builder*. Dieser fügt sich nahtlos in Eclipse ein, so dass man diese weit verbreitete und bewährte Entwicklungsumgebung für Java auch hier einsetzen kann.

Der *Pico Builder* übernimmt alle Aufgaben, die vom Quelltext zu einem ausführbaren Programm führen. Dazu gehört auch erstellen des Java Byte-Codes und die statische Analyse. Die dabei resultierenden Dateien kann man, je nach Notwendigkeit in anderen Perc Tools verwenden, wie z.B. Perc Ultra (eine Java VM für Echtzeitsysteme). Für besonders performancekritische Fälle wird auch ein C-Code bereitgestellt, der mit einem C-Compiler für die Zielplatform in ein direkt ausführbares Programm übersetzt wird[Aon06].

Durch die Überführung in Maschinencode, schafft es Perc Pico zusammen mit seinem innovativen Speichermanagement laut Aussagen des Herstellers selbst native C-Programme in manchen Punkten in der Performance zu schlagen.

Abbildungsverzeichnis

1 Zugriff von außen in eine innere Region 6
2 Zugriff auf einen Kindthread 6
3 Zugriff auf beliebigen Thread 7
4 Zugriff auf übergeordneten Thread 7

Literatur

[Aon06] Aonix. *PERC Pico - The Power of Java, The Efficiency of C*, 2006. http://www.aonix.com/pdf/pico-web.pdf.

[Aon08] Aonix. *PERC Pico 1.1 User Manual*, 2008. http://research.aonix.com/jsc/pico-manual.4-19-08.pdf.

[Nil04] Kelvin Nilsen. Questions and answers regarding proposed static analyzable memory model, October 2004. http://research.aonix.com/jsc/jsc.mem.model.qa.pdf.