Framework für die empirische Bestimmung der Ausführungszeit auf Mehrkernprozessoren

Julian Godesa und Robert Hilbrich

Fraunhofer-Institut für Offene Kommunikationssysteme, FOKUS
Embedded Systems Quality Management (QUEST)
10589 Berlin

julian.godesa@fokus.fraunhofer.de
robert.hilbrich@fokus.fraunhofer.de

Zusammenfassung. Eine hohes Maß an funktionaler Sicherheit eines sicherheitskritischen Systems ist meist auch mit der Forderung nach einem deterministischen Systemverhalten verbunden. Die Entwicklung eines Systems mit deterministischen Verhalten erfordert die Kenntnis der Ausführungszeit aller auszuführenden Anwendungen. Mit der Einführung von Mehrkernprozessoren geraten klassische Verfahren auf Basis einer statischen Analyse zur Bestimmung der Ausführungszeit an ihre Grenzen. In dieser Arbeit werden Erweiterungen zur empirischen Analyse der Ausführungszeit von Anwendungen untersucht. Ein besonderer Schwerpunkt liegt dabei auf de Hinzunehmen von Architekturwissen über den verwendeten Prozessor sowie der strukturellen Informationen über die verwendeten Eingabedaten. Diese Erweiterungen bilden die Grundlage für die automatisierte Konstruktion eines störenden Kontextes für die zu untersuchende Anwendung. Durch den Einsatz von Störfaktoren wird hierbei eine möglichst langen Ausführungszeit provoziert. Zur Analyse der Wirksamkeit dieses Ansatzes werden die entwickelten Störfaktoren und die verwendete Experimentierplattform zusammen mit ersten Messungen vorgestellt.

1 Einleitung

Komplexe eingebettete Systeme übernehmen in immer mehr Lebensbereichen des Menschen wichtige Steuerungsfunktionen. Damit es unter keinen Umständen zu signifikanten Sachschäden kommt, müssen diese Systeme ein hohes Maß an Funktionaler Sicherheit bieten. Neben der funktionalen Korrektheit der Steuersoftware ist auch der Zeitpunkt der Bereitstellung von berechneten Ergebnissen wichtiger Teil dieser Korrektheit [1]. So muss beispielsweise ein Drehwächter für Industrieroboter die Position des Roboterarms periodisch analysieren, um innerhalb einer gegebenen Zeitschranke eine potentielle Gefahr feststellen zu können und ggf. eine sofortige Abschaltung herbeizuführen.

Für die erfolgreiche Entwicklung und Zertifizierung eines sicherheitskritischen Systems, muss dessen logisches und zeitliches Verhalten häufig vollständig deterministisch sein. In der Praxis wird dazu oftmals ein deterministischer Ablaufplan

– *ein statischer Schedule* – zur Steuerung der Ausführung einzelner periodischer Anwendungen eingesetzt. Dieser ist nur dann korrekt, wenn alle Anwendungen hinreichenden Zugang zu Ressourcen erhalten, um ihre Funktionalität innerhalb der vorgegebenen Zeitschranken zu erfüllen. Da ein sicherheitskritisches System die Anforderungen der funktionalen Sicherheit unter *allen Umständen* gewährleisten muss, muss der Ablaufplan genügend Rechenzeit für alle Anwendungen und Eventualitäten bereitstellen. Dies wird erreicht, indem bei der Erstellung des Plans für jede Anwendung die längst mögliche Laufzeit (*Worst-Case Execution Time - WCET*) bestimmt und als Grundlage für die Zuteilung der Ressourcen (z.B. Rechenzeit auf dem Prozessor) verwendet wird [2].

In der Praxis wird die WCET meist mit Hilfe einer statischen Analyse pessimistisch nach oben abgeschätzt. Die zunehmende Komplexität von Prozessoren, die mangelnde Verfügbarkeit von Architektur-Dokumentationen und der Trend zur parallelen Ausführung auf Mehrkernprozessoren führen die etablierten Verfahren der statischen Analyse an ihre Grenzen. Schon bei Einkernprozessoren führen Pipelining und Out-Of-Order Execution zu Ausführungsanomalien, die nur schwer zu modellieren sind [3]. Parallel ausgeführte Anwendungen auf einem Mehrkernprozessor mit gemeinsam genutzten Ressourcen fügen dieser Herausforderung noch eine weitere Komplexitätsdimension hinzu.

Nichtsdestotrotz erfordert die zunehmende Funktionskomplexität von Softwarekomponenten immer mehr Leistung von Hardware, die gegenwärtig nur durch die Nutzung von Mehrkernprozessoren realisiert werden kann [4]. Daraus ergibt sich eine zentrale Herausforderung für die Entwicklung von Echtzeitsystemen in sicherheitskritischen Bereichen: *Wie kann das Zeitverhalten von Anwendungen auf Mehrkernprozessoren sinnvoll und effizient ermittelt werden?*

2 Erweiterte empirische Analyse einer operativen WCET

Da Verfahren der *statischen* Analyse zur Ermittlung der WCET an ihre Grenzen stoßen, ist zu untersuchen inwieweit alternative Ansätze, z.B. auf der Basis von *empirischen* Messungen am realen System, erweitert und für die Entwicklung in der Praxis eingesetzt werden können. Insbesondere bei den hier untersuchten empirischen Verfahren steht dann allerdings das ermittelte Laufzeitverhalten nicht mehr für die *echte* WCET im Sinne einer pessimistischen, oberen Schranke. Stattdessen basieren die Ergebnisse auf konkreten Beobachtungen, so dass wir den Begriff *operative WCET* verwenden, um diesen Unterschied kenntlich zu machen. Im Folgenden wird die Entwicklung einer Analyseumgebung zur automatisierten Ermittlung der operativen WCET vorgestellt, welche im Rahmen einer Diplomarbeit an der Humboldt-Universität zu Berlin in Kooperation mit dem Fraunhofer-Institut FOKUS entstanden ist. Das Ziel der Arbeit bestand darin, die bestehenden empirischen Ansätze zur Bestimmung der Ausführungszeit, um Informationen über die Prozessorarchitektur und die Struktur der Eingabedaten zu erweitern und damit gezielt möglichst lange Ausführungszeiten hervorzurufen. Ein weiterer Schwerpunkt der Arbeit bestand in der weitestgehenden Automatisierung dieser empirischen Analyse um auch die Praxistauglichkeit und

Wiederholbarkeit dieses Verfahrens zu gewährleisten. Als Evaluationsplattform wird der Mehrkernprozessor *Freescale QorIQ P4080* mit acht Kernen in Verbindung mit dem Echtzeitbetriebssystem *WindRiver VxWorks 6.9* verwendet. Eine genaue Beschreibung der Evaluationsplattform ist in Abschnitt 5 zu finden.

2.1 Erweiterung der WCET

Grundlegend basiert die Experimentierumgebung auf der Annahme, dass die operative WCET von Anwendungen auf Mehrkernprozessoren einerseits von der Wahl der Eingabeparameter abhängig ist, die einen konkreten Ausführungspfad innerhalb der Anwendung bestimmen. Darüber hinaus wird die Laufzeit durch Wartezeiten beim Zugriff auf gemeinsame Ressourcen eines Mehrkernprozessors beeinflusst, die parallel von anderen Anwendungen verwendet werden. Diese Annahmen sind in folgender Definition festgehalten:

Definition 1. *Die dynamisch ermittelte* ***operative WCET*** *einer Anwendung t_0 ist die längste gemessene Ausführung unter Verwendung unterschiedlicher Belegungen e der Eingaben E, ausgeführt auf der Zielplattform Z unter dem Einfluss einer parallelen Ausführung von Anwendungen $T - \{t_0\}$. Die Zuordnung einer Anwendung t_i zu einem Rechenkern z_j wird durch die Funktion $m(i) = j$ definiert.*

$$\begin{aligned} &WCET_{operativ}(T, E, Z, m) = \max\{ \\ &time(t_0 \parallel \ldots \parallel t_n,\ e,\ Z) \mid \\ &\ e \in E,\ t_{0\ldots n} \in T,\ m(t_i) = z_j\ \}. \end{aligned} \tag{1}$$

2.2 Identifikation von Störfaktoren

Um die operative WCET einer zu messenden Anwendung näher an die *echte* WCET zu bringen, wird das Verfahren durch den Einsatz von *Störfaktoren* erweitert. Ein Störfaktor ist eine Anwendung, die gemeinsam verfügbare Ressourcen gezielt verwendet und so die Laufzeit einer parallel ausgeführten Anwendung indirekt beeinflusst.

Hierzu müssen in einem ersten Schritt mögliche Interferenzen zwischen Anwendungen auf dem eingesetzten Mehrkernprozessor identifiziert werden. Interferenzen können sich nur über gemeinsam genutzte Ressourcen auf dem Prozessor ergeben. Eine genaue Analyse der Architektur des verwendeten Prozessors bezüglich gemeinsam verwendeter Ressourcen bildet daher den Startpunkt für die Entwicklung geeigneter Störfaktoren. Mit Hilfe dieser Störfaktoren sollten die ermittelten Interferenzen möglichst effektiv ausgelöst werden können.

Schon die nebenläufige Ausführung von Anwendungen auf einem Einkernprozessor kann Einfluss auf das Laufzeitverhalten der Anwendung haben, zum Beispiel indem die Anwendungen abwechselnd die Inhalte im Zwischenspeicher überschreiben. Auf einem Mehrkernprozessor können Interferenzen durch die Verwendung der folgenden, häufig gemeinsam genutzten, Ressourcen auftreten [5]:

- Der gemeinsam verwendete Last-Level Cache
- Die Prefetching Hardware
- Der Front-Side Bus
- Der DRAM-Controller

Auf dieser Basis wurden zwei Störfaktoren entworfen, die mit Hilfe der Experimentierumgebung auf deren Effektivität hin untersucht wurden. Je stärker die Ausführungszeit der untersuchten Anwendung durch einen Störfaktor verlängert werden konnte, desto effektiver arbeitete dieser. Ein Überblick über die entwickelten Störfaktoren ist in Abschnitt 4 zu finden.

3 Vorstellung der Experimentierumgebung

Da bei der Ermittlung der operativen WCET die zu messende Anwendung auf der realen Zielplattform ausgeführt wird, ist dieser Prozess mit einem realen Zeitaufwand verbunden, den es für eine effiziente Entwicklung zu minimieren gilt. Hierfür versucht die Experimentierumgebung den Benutzer beginnend von der Beschreibung einer durchzuführenden Messung, über die Ausführung bis hin zur Auswertung zu unterstützen und wenn möglich Aufgaben automatisiert durchzuführen. Die Experimentierumgebung selbst besteht aus folgenden Komponenten:

NoizdPlugin: Das NoizdPlugin stellt die Schnittstelle zwischen Benutzer und Experimentierumgebung dar. Das NoizdPlugin ist ein Eclipse-Plugin [6], welches selbst wiederum eine Reihe an Plugins verwendet. Das zentrale Plugin ist dabei der Eingabeeditor, mit dessen Hilfe eine Messung beschrieben wird. Ein Plugin zur Visualisierung und Auswertung der ermittelten Laufzeiten ist ebenfalls enthalten.

NoizdDaemon: Der NoizdDaemon ist ein Stellvertreterprogramm, welches auf dem Zielsystem vor der Durchführung einer Messung gestartet werden muss. Das Programm nimmt die Daten und Befehle des NoizdPlugins entgegen und sorgt für ein geordnetes Starten der auszuführenden Anwendungen. Es ist grundsätzlich möglich mehrere Zielsysteme für eine Messung zu verwenden.

NoizdLib: Die NoizdLib ist das Verbindungsglied zwischen einer auszuführenden Anwendung und den restlichen Komponenten der Experimentierumgebung. Es ist eine Programmbibliothek, welche die auszuführende Anwendung „umhüllt“. Eine mit der NoizdLib gelinkte Anwendung wird zur Unterscheidung im Folgenden als *Agent* bezeichnet. Die zu messende Anwendung wird gesondert als *SUT* (für „Software Under Test“) gekennzeichnet.

Die Kommunikation der einzelnen Komponenten zur Durchführung einer Messung ist in Abbildung 1 dargestellt. Es wird zwischen der Experimentierplattform, von der aus die Messung gestartet wird und der Zielplattform, auf der die Messung durchgeführt wird, unterschieden. Das NoizdPlugin überträgt in einem ersten Schritt die Agenten auf das Zielsystem. Diese Daten werden vom NoizdDaemon entgegengenommen und zur Ausführung gebracht. Über ein auf

TCP/IP basierendes Kommunikationsprotokoll sendet nun das NoizdPlugin den Agenten Befehle, dass die Ausführung der Anwendung beginnen kann. Nachdem das SUT das Beenden der zu messenden Anwendung gemeldet hat, wird die Ausführung der Agenten beendet und das Ergebnis der Messung ausgewertet. Das NoizdPlugin kann für die Ausführung einer Anwendung existierende Eingabedaten verwenden oder diese zufällig erzeugen. Des Weiteren kann das NoizdPlugin auch automatisch nach einer Belegung der Eingabedaten suchen, welche zu einer möglichst langen Laufzeit der zu messenden Anwendung führen.

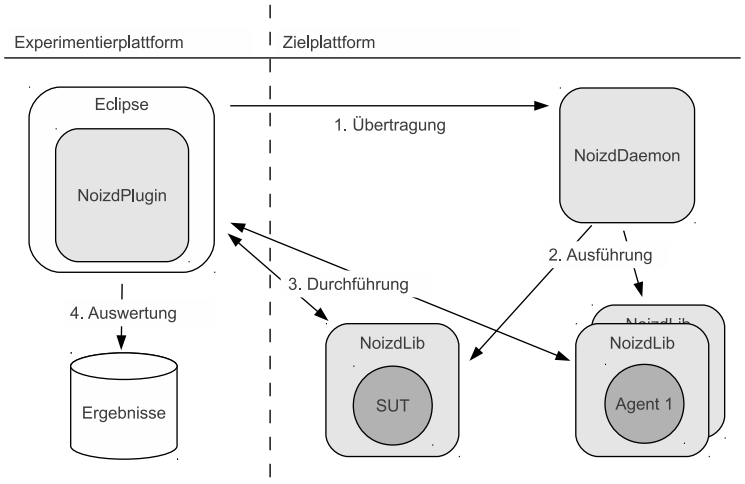

Abb. 1. Kommunikation der einzelnen Komponenten

Hierfür wird ein *evolutionärer Algorithmus* verwendet [7]. Es wird lediglich die Definition der Wertebereiche für die Eingabeparameter vom Benutzer benötigt. Die Suche nach konkreten Werten, die zu möglichst langen Ausführungszeiten führen, wird durch das NoizdPlugin automatisch durchgeführt. In Listing 1.1 und 1.2 ist eine solche Definition bzw. die erzeugte Belegung dargestellt. Zur Messung der Laufzeit wird auf die Funktion `getticks()` von dem FFTW-Projekt [8] zurückgegriffen. Diese Funktion liefert bei einem Aufruf einen Zeitstempel mit Hilfe dessen die Laufzeiten verglichen werden können.

4 Vorstellung der Störfaktoren

Die Experimentierumgebung ermöglicht eine einfache parallele Ausführung von mehreren Agenten, wovon einer die zu messende Anwendung ist. Um die Ausführung der SUT möglichst stark negativ zu beeinflussen, sind im Rahmen dieser Diplomarbeit zwei Störfaktoren entwickelt worden: Der *CacheTrasher* (kurz CT) und der *DatarateTrasher* (kurz DT).

Listing 1.1. Definition von Eingaben

```
input random {
  delimiter = "\n"
  seeds { 123456 }
  definition {
    int [5] range 0 to 15
    /* comment */
    float range -9.1 to 12.0
    "not true"
    bool value = true
    bool [2] value = false
  } }
```

Listing 1.2. Erzeugte Eingaben

```
6
14
15
7
4
/* comment */
0.7056636
not true
true
false
false
```

4.1 Das Überschreiben von zwischengespeicherten Daten

Dem ersten Störfaktor liegt eine einfache Annahme zugrunde: Je mehr Daten einer Anwendung im Zwischenspeicher überschrieben werden, desto mehr Daten müssen bei einem erneuten Zugriff aus dem Hauptspeicher nachgeladen werden. Dies verlängert die Laufzeit einer Anwendung.

Hierzu greift der Störfaktor lesend und dann schreibend auf die Elemente einer Liste zu. Die Größe der Liste kann durch die Variable `arraySize` kontrolliert werden. Um möglichst schnell den spezifizierten Speicherplatz zu belegen, wird versucht auf nur ein Datum innerhalb einer Cache-Line zuzugreifen. Dies wird durch ein Überspringen einzelner Elemente der Liste erzielt. Wie viele Elemente übersprungen werden, wird durch die Variable `cacheline` definiert. Die Anzahl der Zugriffe kann über die Variable `numberOfAccess` konfiguriert werden.

Listing 1.3. Der CacheTrasher Algorithmus

```
for (i=0; i<numberOfAccess; i++) {
  array[(i * cacheline) & (arraySize-1)]++;
}
```

4.2 Das Belegen des Bussystems und des Speicher-Controllers

Der zweite Störfaktor versucht durch häufige Speicheranfragen den Speicher-Controller sowie das Interconnect (die Kommunikationsverbindung zwischen den Kernen auf dem QorIQ P4080 Prozessor) zu belasten. Da der Speicher-Controller um ein Vielfaches langsamer arbeitet als ein Rechenkern werden mehrere Daten vom Speicher-Controller in Blöcken gesendet. Dadurch bleibt die Latenz gleich, aber die Datenrate steigt. Der Datarate-Trasher nutzt diese Datenparallelität aus, um möglichst viele Speicheranfragen an den Speicher-Controller zu senden. Anhand der Variable `numberOfLists` kann die zu verwendende „Parallelität“ eingestellt werden. Jede Liste enthält entweder eine zufällige oder geordnete Reihenfolge von Positionen, welche angeben wie über die Liste iteriert werden

Listing 1.4. Der DatarateTrasher Algorithmus

```
TYPE **list = ...;
TYPE *next = ...;
for (i=0; i<iterations; i++) {
  switch (numberOfLists) {
    case 18:
    next[17] = list[17][next[17]];
    ...
    case 1:
    next[0] = list[0][next[0]];
  } }
```

soll. Sind die Listen mit einer zufälligen Reihenfolge initialisiert, so werden in kürzester Zeit mehrere Lesezugriffe auf unabhängige Daten an den Controller gesendet.

5 Das Zielsystem: QorIQ P4080 und VxWorks 6.9

Die Hardwareplattform *QorIQ P4080* verwendet acht gleiche Rechenkerne die den PowerPC Befehlssatz implementieren. Sie sind mit 1,2 GHz getaktet. Jeder Rechenkern besitzt einen privaten L1 Daten- sowie Befehlscache mit einer Speichergröße von jeweils 32 KByte. Zusätzlich verfügt jeder Rechenkern über einen privaten L2 Cache der Größe 128 KByte. Laut Referenzhandbuch beträgt die Größe einer Cache-Line 64 Byte [9]. Die einzelnen Kerne sind über ein Crossbar-Switch miteinander verbunden. Zwischen dem Interconnect und den beiden DDR2/3 Speicher-Controllern befinden sich jeweils 1 MByte große L3 Caches, die von den Rechenkernen gemeinsam verwendet werden. Die eingesetzte Plattform verwendet 4 GByte an DDR3 Speicher.

Als Betriebssystem wird das Echtzeit-Betriebssystem VxWorks 6.9 von WindRiver eingesetzt. Durch die Verwendung sogenannter „Real-Time Prozesse“ (kurz RTP) laufen die Anwendungen nicht mehr im Kernelmodus, sondern werden im Benutzermodus ausgeführt. Im Gegensatz zu Kernel-Tasks, die ohne Schutzvorkehrungen auf den gesamten Speicher Zugriff haben, kann eine RTP Anwendung nur seinen eigenen Speicherbereich verwenden. Die evaluierten Störfaktoren wurden als solche RTP Anwendungen implementiert.

6 Durchgeführte Experimente

Um die Anwendbarkeit des Ansatzes einer operativen WCET Ermittlung evaluieren zu können, wurde mit Hilfe der Experimentierumgebung die Effektivität der vorgestellten Störfaktoren evaluiert. Aus Platzgründen wird hier auf die Darstellung der durchgeführten Experimente zum Datarate-Trasher verzichtet und näher auf das Belegen des gemeinsam verwendeten Zwischenspeichers eingegangen.

6.1 Referenzmessung

Für die folgenden Experimente wird als SUT der Algorithmus aus Listing 1.3 verwendet. In einer ersten Referenzmessung wird die Laufzeit ohne Beeinflussung durch Störfaktoren gemessen. Hierbei wird die Größe der zu durchlaufenden Liste schrittweise verdoppelt (von 1 KByte bis 8 MByte). Die Anzahl der Speicherzugriffe wird bei jeder Messung beibehalten (`numberOfAccess` = 524288). Insgesamt werden 14 Laufzeiten ermittelt, mit gleicher Anzahl an Zugriffen aber unterschiedlicher Listengröße.

Beschränken sich die Datenzugriffe auf den L1-Zwischenspeicher, so beträgt die Laufzeit 419433 Ticks (1KB). Wird auf Daten zugegriffen, die sich im L2-Zwischenspeicher befinden, steigt die Laufzeit auf 628434 Ticks (64KB). Dies entspricht einem Faktor von 1,49 gegenüber der Laufzeit im Falle des L1-Caches. Wird der L3-Cache verwendet, so steigt die Laufzeit auf 3528208 Ticks (8MB). Dies entspricht einem Faktor von 8,41.

6.2 Belegen von privatem Zwischenspeicher

Um einen möglichen Einfluss der Störfaktoren abschätzen zu können, wird das obige Experiment wiederholt, wobei diesmal sechs Störfaktoren parallel zur SUT ausgeführt werden. Die Störfaktoren verwenden den gleichen Algorithmus, wobei einmal 32 KByte und in einem weiteren Experiment 128 KByte an Speicherplatz belegt werden. In Abbildung 2 sind die Laufzeiten der Messungen in Ticks angegeben. Die Linie *Referenzmessung* ist die Laufzeit ohne Störfaktoren. Die beiden anderen Linien geben die Laufzeiten unter Beeinflussung wieder. Wie zu erwarten gelang es den Störfaktoren nicht, die Ausführung einer Anwendung zu stören, die nur auf Daten zugreift, die im privaten Zwischenspeicher liegen.

6.3 Belegen von gemeinsam verwendeten Zwischenspeicher

Das Experiment mit den sechs Störfaktoren wurde wiederholt, wobei nun die Störfaktoren jeweils 1 MByte, 2 MByte und dann 4 MByte an Speicherplatz belegen sollen. In Abbildung 3 sind die Laufzeitmessungen der SUT unter Beeinflussung dargestellt. Es ist zu erkennen, dass die Störfaktoren das SUT erfolgreich beeinflussen, wenn dieses auf Daten in dem gemeinsam verwendeten L3-Cache zugreift (ab 128KB). Die größte Beeinflussung durch die sechs Störfaktoren ist schon bei der Verwendung von 512 KByte erreicht. Die Laufzeit der SUT verlangsamt sich auf 7373440 Ticks, was einem Faktor von 5,43 entspricht. Hierbei verwenden die Störfaktoren jeweils 4 MByte an Daten. Die längste Ausführungsdauer wurde beobachtet, als sowohl die Störfaktoren, als auch das SUT auf 2 MByte an Daten zugriffen (7785542 Ticks).

7 Fazit

Die hier vorgestellten Störfaktoren sind ein erster Schritt in Richtung einer verbesserten dynamischen Bestimmung der WCET. Ihre Wirksamkeit wurde in

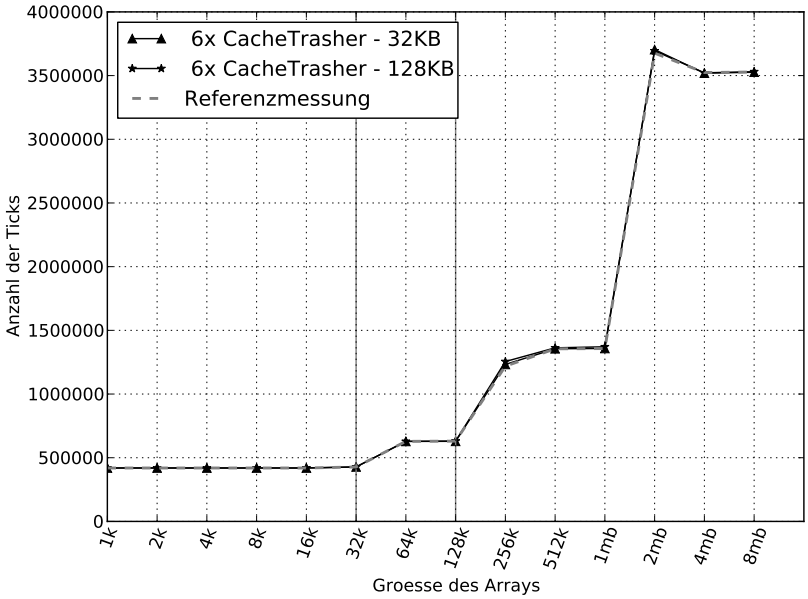

Abb. 2. Belegen von privatem Zwischenspeicher

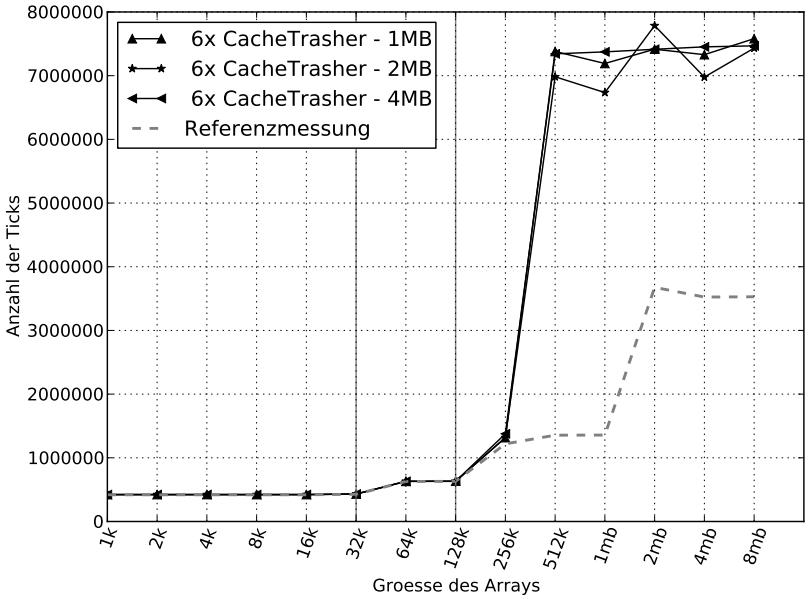

Abb. 3. Belegen von gemeinsam verwendeten Zwischenspeicher

verschiedenen Experimenten auf einem Mehrkernprozessor untersucht und dokumentiert. Dabei wurde eine eigens für diesen Zweck entwickelte Experimentierumgebung verwendet, die eine weitgehende Automatisierung der Messungen ermöglicht. Es wurde deutlich, dass die durchgeführten Experimente mit einem anders konfigurierten Betriebssystem ohne viel Aufwand wiederholt werden können, um beispielsweise die Eigenschaften verschiedener Echtzeitbetriebssysteme miteinander zu vergleichen. Die Ergebnisse der Experimente zeigen, dass durch das Hinzunehmen von Architekturwissen und die gezielte Suche im Raum möglicher Eingabewerte, die Ausführungszeit von Anwendungen wesentlich verlängert werden kann. Innerhalb einiger Experimente, bei denen die Verwendung des gemeinsam genutzten L3-Caches gestört wurde, wurde teilweise eine Abweichung von mehr als 300000 Ticks gegenüber dem arithmetischem Mittel beobachtet. Dies zeigt noch einmal die starken Schwankungen in der Ausführungszeit, die sich als Folge der Verwendung komplexer Hardware-Architekturen ergeben und den Einsatz statischer Verfahren zur Abschätzung der Ausführungszeit wesentlich erschweren.

Literaturverzeichnis

1. Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.
2. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.
3. Christoph Berg, Jakob Engblom, and Reinhard Wilhelm. Requirements for and design of a processor with predictable timing. In *Perspectives Workshop: Design of Systems with Predictable Behaviour, 16.-19. November 2003, volume 03471 of Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl*, 2004.
4. Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
5. Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGARCH Comput. Archit. News*, 38(1):129–142, March 2010.
6. The eclipse foundation. `http://www.eclipse.org/`.
7. Daniel W. Dyer. The watchmaker framework for evolutionary computation. `http://watchmaker.uncommons.org`.
8. Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".
9. Freescale. *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual*, g edition, April 2010.