
Cache-Kohärenz in hart echtzeitfähigen Mehrkern-Prozessoren

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Arthur Pyka

Dortmund

2015

Tag der mündlichen Prüfung: 19.03.2015

Dekan:	Prof. Dr. Gernot A. Fink
Gutachter:	Prof. Dr. Sascha Uhrig
	Prof. Dr. Peter Marwedel

Danksagung

Ich möchte mich gerne bei einigen Personen bedanken, die auf vielfältige Weise zur Entstehung dieser Arbeit beigetragen haben.

Zuerst geht mein Dank an Prof. Dr. Sascha Uhrig und Prof. Dr. Peter Marwedel für die Begutachtung dieser Arbeit. Ich danke auch Prof. Dr. Olaf Spinczyk für die Bereitschaft als mein Mentor zu fungieren.

Ein besonderer Dank geht an Prof. Uhrig für seine ausgezeichnete Betreuung in den letzten Jahren und die Chance in seiner Arbeitsgruppe und an diesem interessanten Thema arbeiten zu dürfen. Ihm verdanke ich die Möglichkeit an spannenden Projekten in Forschung und Lehre zu arbeiten und an zahlreichen Konferenzen und Workshops teilzunehmen. Dank gilt auch meinen Kollegen Mathias Rohde und Lilian Tadros, die mich in den letzten Jahren immer wieder unterstützt haben. Ich denke gerne an die gemeinsame Arbeitszeit zurück.

Ebenso danke ich Prof. Marwedel, der mir bereits während meines Studiums eine Mitarbeit in seiner Arbeitsgruppe ermöglicht und so mein Interesse an der Forschung geweckt hat. Dazu beigetragen haben besonders Prof. Dr. Heiko Falk und Dr. Sascha Plazar, denen ich hiermit meinen Dank ausdrücken möchte.

Unverzichtbar für diese Arbeit war die Zusammenarbeit mit den zahlreichen Kollegen, die im parMERASA Projekt involviert waren. Die vielen fruchtbaren Diskussionen waren stets ein Gewinn für meine Arbeit. Ich möchte besonders die Zusammenarbeit mit den Kollegen des „Institut de Recherche en Informatique“ in Toulouse hervorheben. Christine Rochange, Hugues Cassé und Haluk Ozaktas haben mit ihrer Gastfreundschaft und besonders mit ihrer fortlaufenden Unterstützung einen wichtigen Anteil an der Entstehung dieser Arbeit.

An dieser Stelle möchte ich auch meiner Familie danken. Insbesondere meinen Eltern, die mich in all den Jahren meiner Ausbildung so selbstlos unterstützt haben. Ich danke meiner Frau Ramona für ihre Liebe und ihr Verständnis. Und ich danke meinen Geschwistern und engen Freunden, auf deren Unterstützung ich mich immer verlassen kann.

Kurzfassung

Im Bereich der Echtzeitsysteme rücken Mehrkern-Prozessoren zunehmend in den Fokus. Dabei stellen Echtzeitsysteme besondere Anforderungen an die eingesetzte Systemarchitektur. Neben der logischen Korrektheit, ist in Echtzeitsystemen eine zeitlich vorhersagbare Ausführung entscheidend. Cache-Speicher spielen in dieser Hinsicht eine besondere Rolle. Zum einen sind sie notwendig, um schnelle Zugriffe auf Instruktionen und Daten zu gewährleisten, zum anderen beeinträchtigen sie die zeitliche Vorhersagbarkeit der Ausführung. Beim Zugriff auf gemeinsame Daten in Mehrkern-Prozessoren ist zudem der Einsatz eines Cache-Kohärenzverfahrens notwendig. Gängige Kohärenzverfahren können die Anforderungen an Performanz und Echtzeitfähigkeit nicht hinreichend erfüllen. Die in hardwarebasierten Kohärenzverfahren eingesetzten Kohärenzoperationen machen eine präzise WCET-Abschätzung undurchführbar.

Der On-Demand Coherent Cache (ODC²) stellt ein Cache-Kohärenzverfahren dar, das im Hinblick auf den Einsatz in Echtzeitsystemen entwickelt wurde. Es verzichtet auf eine gegenseitige Beeinflussung von Cache-Speicher durch Kohärenzoperationen und erreicht dadurch eine hinreichende zeitliche Vorhersagbarkeit der Zugriffe auf gemeinsame Daten. Das Verfahren des ODC² zielt auf eine möglichst effiziente Nutzung des Cache-Speichers hin. Im Vergleich zu gängigen, softwarebasierten Verfahren ermöglicht es eine signifikant höhere (Worst-Case) Performanz.

Veröffentlichungen

Teile dieser Arbeit wurden in den Veröffentlichungen verschiedener Konferenzen und Workshops publiziert. Es folgt eine chronologische Auflistung dieser Veröffentlichungen:

PYKA, A. ; ROHDE, M. ; UHRIG, S. : A Real-time Capable First-level Cache for Multi-cores. In: *Workshop on High Performance and Real-time Embedded Systems (HiRES) in conjunction with HiPEAC'13, Berlin, Germany*

PYKA, A. ; ROHDE, M. ; FERNANDES, J. ; UHRIG, S. : On-Demand Coherent Cache for Parallelised Hard Real-Time Applications. In: *Proceedings of the Work-in-Progress Session of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013), Paris, France*

PYKA, A. ; ROHDE, M. ; UHRIG, S. : Performance Evaluation of the Time Analysable On-Demand Coherent Cache. In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, 2013, S. 1887–1892

UNGERER, T. ; BRADATSCH, C. ; GERDES, M. ; KLUGE, F. ; JAHR, R. ; MISCHKE, J. ; FERNANDES, J. ; ZAYKOV, P. ; PETROV, Z. ; BODDEKER, B. ; KEHR, S. ; REGLER, H. ; HUGL, A. ; ROCHANGE, C. ; OZAKTAS, H. ; CASSÉ, H. ; BONENFANT, A. ; SAINRAT, P. ; BROSTER, I. ; LAY, N. ; GEORGE, D. ; QUINONES, E. ; PANIC, M. ; ABELLA, J. ; CAZORLA, F. ; UHRIG, S. ; ROHDE, M. ; PYKA, A. : parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. In: *Digital System Design (DSD), 2013 Euromicro Conference on*, 2013, S. 363–370

PYKA, A. ; ROHDE, M. ; ZAYKOV, P. G. ; UHRIG, S. : Case Study: On-Demand Coherent Cache for Avionic Applications. In: *2nd Workshop on High-performance and Real-time Embedded Systems (HiRES 2014), Vienna, Austria*

PYKA, A. ; ROHDE, M. ; UHRIG, S. : A real-time capable coherent data cache for multicores. In: *Concurrency and Computation: Practice and Experience* 26 (2014), Nr. 6, S. 1342–1354. <http://dx.doi.org/10.1002/cpe.3172>. – DOI 10.1002/cpe.3172. – ISSN 1532–0634

PYKA, A. ; ROHDE, M. ; UHRIG, S. : Extended performance analysis of the time predictable on-demand coherent data cache for multi- and many-core

systems. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014*, 2014, S. 107–114

PYKA, A. ; TADROS, L. ; UHRIG, S. ; CASSÉ, H. ; OZAKTAS, H. ; ROCHANGE, C. : WCET Analysis of Parallel Benchmarks using On-Demand Coherent Cache. In: *3rd Workshop on High-performance and Real-time Embedded Systems (HiRES 2015), Amsterdam, the Netherlands*

UNGERER, T. ; BRADATSCH, C. ; FRIEB, M. ; KLUGE, F. ; MISCHÉ, J. ; STEGMEIER, A. ; JAHR, R. ; GERDES, M. ; ZAYKOV, P. ; MATUSOVA, L. ; LI, Z. J. J. ; PETROV, Z. ; BÖDDEKER, B. ; KEHR, S. ; REGLER, H. ; HUGL, A. ; ROCHANGE, C. ; OZAKTAS, H. ; CASSÉ, H. ; BONENFANT, A. ; SAINRAT, P. ; LAY, N. ; GEORGE, D. ; BROSTER, I. ; QUIÑONES, E. ; PANIC, M. ; ABELLA, J. ; HERNANDEZ, C. ; CAZORLA, F. ; UHRIG, S. ; ROHDE, M. ; PYKA, A. : Experiences and Results of Parallelisation of Industrial Hard Real-time Applications for the parMERASA Multi-core. In: *3rd Workshop on High-performance and Real-time Embedded Systems (HiRES 2015), Amsterdam, the Netherlands*

UHRIG, S. ; TADROS, L. ; PYKA, A. : MESI-based Cache Coherence for Hard Real-time Multicore Systems. In: *28th GI/ITG International Conference on Architecture of Computing Systems, Porto, Portugal*

Inhaltsverzeichnis

1	Einführung	1
1.1	Motivation	1
1.2	Ziele dieser Arbeit	3
1.3	Aufbau	4
1.4	Beitrag des Autors zur Dissertation	4
2	Grundlagen	7
2.1	Cache-Speicher	7
2.2	Mehrkern-Prozessoren	11
2.2.1	Aufbau eines Mehrkern-Prozessors	12
2.2.2	Speicherhierarchie	14
2.3	Konsistenz und Kohärenz in Mehrkern-Prozessoren	15
2.3.1	Konsistenz und Konsistenzmodelle	16
2.3.2	Cache-Kohärenz	19
2.4	Cache-Kohärenzprotokolle	21
2.4.1	Snooping-Protokolle	21
2.4.2	Verzeichnis-Protokolle	23
2.4.3	Software Cache-Kohärenz	25
2.5	Echtzeitsysteme	26
2.5.1	WCET	27
2.5.2	Laufzeitanalyse	28
2.5.3	Cacheanalyse in der statischen Laufzeitanalyse	30
2.5.4	Echtzeitfähigkeit in Mehrkern-Prozessoren	31
3	Cache-Kohärenz in Echtzeitsystemen	35
3.1	Voraussetzungen für echtzeitfähige Cache-Kohärenz	36
3.2	Cache-Kohärenz in aktuellen Mehrkern-Prozessoren	37
3.3	Analyse bekannter Cache-Kohärenzprotokolle	39

3.3.1	Einfluss auf die Cacheanalyse	39
3.3.2	Einfluss auf die Latenz eine Cache-Zugriffs	45
3.3.3	Cache-Miss Latenz in Snooping-Protokollen	47
3.3.4	Cache-Miss Latenz in Verzeichnis-Protokollen	52
3.3.5	Latenz bei Schreibzugriffen	53
3.3.6	Vorhersagbarkeit von Schreiblatenzen	58
3.3.7	Schlussfolgerungen	58
3.4	Echtzeitfähige Speicherhierarchie in der Forschung	60
4	On-Demand Coherent Cache	63
4.1	Funktionsweise	65
4.1.1	Private-Mode und Shared-Mode	65
4.1.2	Restore-Procedure	67
4.1.3	Schreibstrategie des ODC ²	68
4.1.4	Auswirkungen auf die Performanz	70
4.2	Realisierung	71
4.2.1	ODC ² Kontrollanweisungen	71
4.2.2	Synchronisationstechniken	72
4.2.3	Abbildung privater und gemeinsamer Daten	73
4.2.4	Hardwareanforderungen	73
4.3	Echtzeitfähigkeit	75
4.3.1	Ausführung der Kontrollanweisungen	75
4.3.2	Markierung der Cache-Zeilen	76
4.3.3	Restore-Procedure	77
4.3.4	Vorhersage markierter Cache-Zeilen	78
4.4	Bezug zu verwandten Arbeiten	79
5	Evaluation	81
5.1	Vorangehende Studien	82
5.2	Evaluationsplattform	83
5.2.1	Zugriffslatenzen in der Evaluation	87
5.3	Tools der Evaluation	88
5.3.1	Ausführungssimulation	88
5.3.2	Statische WCET-Analyse	88
5.4	Benchmarks	89
5.5	Auswertung	91
5.5.1	Performanz im Echtzeitsystem	94
5.5.2	Einfluss der Kohärenzoperationen	99
5.5.3	Zugriffsverhalten im Cache	101
5.5.4	Variation der Cachekonfiguration	109
5.6	Fazit	113
6	Zusammenfassung und Ausblick	115

6.1 Ausblick	117
Abkürzungsverzeichnis	119
Abbildungsverzeichnis	120
Tabellenverzeichnis	123
Literaturverzeichnis	125
7 Anhang	143
Anhang	143
Anhang 1: Laufzeitmessung	144
Anhang 2: WCET-Abschätzung	145
Anhang 3: Skalierung der Laufzeit und WCET	146
Anhang 4: Histogramm der Invalidierung	148
Anhang 5: Zugriffstypen	151

Kapitel 1

Einführung

You may delay, but time will not. *Benjamin Franklin*

1.1 Motivation

Der Prozessor eines Computersystems wird häufig mit einem menschlichen Gehirn verglichen. Als zentrale „Ausführungseinheiten“ sind beide für den Betrieb und die Informationsverarbeitung zuständig. Doch so beliebt wie der Vergleich ist, so sehr hinkt er auch. Kann sich doch eine sequentielle Rechenmaschine kaum mit der komplexen Interaktion von Milliarden von Nervenzellen messen. Betrachtet man jedoch den aktuellen Fortschritt in der Entwicklung von Prozessorarchitekturen, bekommt man den Eindruck, dass unsere elektronischen Gegenstücke allmählich aufholen.

In heutigen Computersystemen haben Mehrkern-Prozessoren den Einsatz von Einkern-Prozessoren abgelöst. Auf der Jagd nach immer höheren Taktraten sind Entwickler von Mikroprozessoren an physikalische Grenzen gestoßen, die eine weitere Performanzsteigerung mit Einkern-Prozessoren nicht mehr effizient möglich machen. In Hinblick auf den zu Beginn aufgestellten Vergleich, ist es offensichtlich, dass Performanz nicht zwingend eine Frage der Rechengeschwindigkeit ist, sondern ebenso von der Möglichkeit profitiert, mehrere Aufgaben parallel ausführen zu können. Die Ersetzung von Einkern-Prozessoren durch Mehrkern-Prozessoren liegt da auf der Hand.

Mehrkern-Prozessoren ermöglichen eine vollständig parallelisierte Ausführung von Applikationen auf den im Prozessor verfügbaren Rechenkernen. Ein

Zustand, der mit mehrfädigen Einkern-Prozessoren bisher nur scheinbar erreicht wurde. Mehrkern-Prozessoren sind heute in nahezu allen Bereichen computergestützter Berechnung anzutreffen, sei es für Datenbanksysteme, Automatisierung, High-Performance Anwendungen oder den einfachen Heimcomputer. Wurde bei der Einschätzung zukünftiger Performanzsteigerung bislang meist Moore's Gesetz [93] zitiert, welches einen kontinuierlichen technologischen Fortschritt in der Entwicklung von Prozessoren vorhersagt, werden in Zukunft wohl eher Betrachtungen wie das Gesetz von Amdahl [63], das Grenzen für die Effizienz parallelisierter Berechnungen aufzeigt, im Vordergrund stehen. Wir sind heute noch weit davon entfernt die Möglichkeiten, welche die Mehrkern-Technologie bietet, erschöpfend auszunutzen. Die Entwicklung von parallelisierten Applikationen ist sehr komplex und harmoniert nicht gerade mit unserem eigenen Denkschema. Unser Bewusstsein ist, um den Vergleich ein letztes Mal zu bemühen, eher auf die sequentielle Verrichtung einer einzelnen Aufgabe ausgerichtet. Doch der hohe Bedarf an effizient parallelisierten Applikationen, der mit der Ausbreitung von Mehrkern-Prozessoren beflügelt wird, ist Antrieb genug, um diese Lücke zu schließen.

Mag der Markt auf den ersten Blick bereits von Mehrkern-Prozessoren durchdrungen sein, ist ein buchstäblich flächendeckender Einsatz im Moment noch nicht erreicht. Gerade in Bereichen, in denen maximale Performanz nicht das entscheidende Kriterium ist, ist eine starke Zurückhaltung vor der Anwendung neuer Mehrkern-Architekturen zu erkennen. So dringen Mehrkern-Prozessoren erst langsam in den Bereich der sicherheitskritischen Systeme oder harten Echtzeitsysteme vor.

Harte Echtzeitsysteme stellen hohe Anforderungen an die verwendete Systemarchitektur. Anstelle einer möglichst kurzen Ausführungszeit, liegt das Hauptaugenmerk auf der zeitlichen Vorhersagbarkeit der Ausführung. Aktuelle Mehrkern-Architekturen sind in der Regel auf hohe Performanz ausgerichtet und können diese Anforderung nicht hinreichend erfüllen. Die darin verwendeten Pipeline-Optimierungen und eine komplexe Speicherhierarchie machen den zeitlichen Ablauf einer Ausführung schwer vorhersehbar. Gerade Cache-Speicher haben sich als eine Quelle zeitlicher Unvorhersagbarkeit erwiesen, insbesondere in einem Mehrkern-Prozessor. Verfügen die Rechenkerne eines Mehrkern-Prozessors über private Cache-Speicher, macht dies den Einsatz einer Cache-Kohärenztechnik notwendig, um einen korrekten Zugriff auf gemeinsame Daten zu gewährleisten. Gängige Cache-Kohärenzprotokolle verschlechtern jedoch die zeitliche Vorhersagbarkeit der Ausführung in hohem Maße und sind daher für den Einsatz in harten Echtzeitsystemen nicht geeignet.

Bei der Ausführung sicherheitskritischer Anwendungen kommen Cache-Speicher selten bis gar nicht zum Einsatz. Zeitliche Vorhersagbarkeit hat im Bereich der Echtzeitsysteme Priorität, so dass auf einen Performanzgewinn mit Hilfe eines Caches verzichtet wird. Doch auch für sicherheitskritische Systeme

werden die Anwendungen zunehmend komplexer und fordern immer mehr Rechenzeit ein. Die Tatsache, dass gerade in Mehrkern-Prozessoren ohne Caches eine hohe Performanz kaum zu erreichen ist, erklärt die Zurückhaltung vor der Verwendung dieser Architekturen in Echtzeitsystemen. Aktuelle Spezifikationen die Standards für Software-Architekturen im Automobil- (AUTOSAR 4.0 [13], ISO 26262 [66]) oder Avionik-Bereich (ARINC 653 [64]) festlegen, setzen der Verwendung von Mehrkern-Architekturen enge Grenzen.

Die Vielzahl aktueller Forschungsprojekte, die sich mit paralleler Ausführung in Echtzeitsystemen beschäftigen, wie z.B. parMERASA [139], T-CREST [47], MultiPARTES [46], CERTAINTY [44], ARAMiS [20], DREAMS [45] sowie EMC2 [11], zeigt den dringenden Bedarf einer effizienten Mehrkern-Architektur für den Bereich der Echtzeitsysteme. Die effiziente Nutzung privater Cache-Speicher ist einer der Stolpersteine auf diesem Weg. Es ist daher notwendig, den Aufbau einer Cache-Kohärenztechnik neu zu überdenken und an die Anforderungen harter Echtzeitsysteme anzupassen. Dabei ist es lohnenswert bekannte Konzepte der Cache-Kohärenz aufzugreifen, die aufgrund unzureichender Performanz nicht eingesetzt werden, aber zeitliche Vorhersagbarkeit ermöglichen, und in einem neuartigen Konzept zu vereinen.

1.2 Ziele dieser Arbeit

Ziel dieser Arbeit ist es zum einen, bekannte Cache-Kohärenzverfahren auf ihre Anwendbarkeit in harten Echtzeitsystemen hin zu untersuchen. Es werden zuerst Voraussetzungen für den Einsatz solcher Techniken in harten Echtzeitsystemen definiert. Das Augenmerk liegt dabei auf einer möglichst effizienten Nutzung des Cache-Speichers bei gleichzeitiger Sicherstellung der zeitlichen Vorhersagbarkeit.

Gängige Kohärenzprotokolle, wie sie in heutigen Mehrkern-Prozessoren eingesetzt werden, gelten als ungeeignet für harte Echtzeitsysteme. Auf Grundlage der definierten Voraussetzungen werden gängige Kohärenzprotokolle auf ihre Eignung für harte Echtzeitsysteme hin untersucht. Es werden die in den verschiedenen Verfahren zum Einsatz kommenden Kohärenztechniken analysiert, und es wird dargestellt, inwieweit diese Techniken die zeitliche Vorhersagbarkeit der Ausführung einer parallelen Anwendung beeinflussen.

Desweiteren wird in dieser Arbeit ein Cache-Kohärenzverfahren vorgestellt, der On-Demand Coherent Cache (ODC²), das den Anforderungen eines harten Echtzeitsystems gerecht wird. Es wird erläutert, inwieweit das Verfahren auf bekannte Kohärenztechniken aufbaut und im Verhältnis zu aktuellen Forschungsansätzen steht. Die Anwendung des ODC² bei der Ausführung paralleler Applikationen wird demonstriert und es wird gezeigt wie kohärente Zugriffe auf gemeinsame Daten in einem Mehrkern-Prozessor ermöglicht werden.

Es wird detailliert dargestellt, welche Anforderungen dieses Verfahren an die Systemhardware und an die Entwicklung von parallelen Programmen stellt.

Im Rahmen einer Evaluation wird der ODC² mit anderen Verfahren zur Sicherstellung kohärenter Zugriffe auf gemeinsame Daten verglichen, welche aktuell in sicherheitskritischen Systemen Anwendung finden. Dazu werden die Auswirkungen dieser Verfahren auf die Umsetzung von Speicherzugriffen bei der Ausführung paralleler Benchmarks untersucht. Es werden sowohl Resultate einer simulierten Laufzeitmessung, als auch einer statischen Laufzeitanalyse in Betracht gezogen. Anhand der Ergebnisse der Evaluation wird gezeigt, dass bei Einsatz des ODC² eine hinreichend präzise Abschätzung der maximalen Ausführungszeit möglich ist und das Verfahren somit für harte Echtzeitsysteme geeignet ist.

1.3 Aufbau

Diese Arbeit ist wie folgt aufgebaut: Kapitel 2 erläutert Grundlagen über Cache-Speicher, Mehrkern-Prozessoren und Echtzeitsysteme, die für das Verständnis der Arbeit notwendig sind. In Kapitel 3 werden Voraussetzungen für echtzeitfähige Cache-Kohärenz definiert und bekannte Kohärenzprotokolle auf Grundlagen dieser Voraussetzungen analysiert. Kapitel 4 stellt den zeitlich vorhersagbaren On-Demand Coherent Cache vor, der in Kapitel 5 zusammen mit anderen Verfahren evaluiert wird. Die Arbeit schließt mit einem Fazit und Ausblick in Kapitel 6 ab.

1.4 Beitrag des Autors zur Dissertation

Nach §10(2) der „Promotionsordnung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011“ muss der Autor einer Dissertation in einem gesonderten Abschnitt beschreiben, welche Ergebnisse in Kooperation mit anderen Personen erzielt wurden und worin der Eigenanteil des Autors an diesen Ergebnissen liegt. Im Folgenden wird dies zu den einzelnen Kapiteln dieser Arbeit erläutert:

- **Kapitel 3:** Die Analyse bekannter Cache-Kohärenzprotokolle wurde zum Teil in Zusammenarbeit mit Prof. Dr. Sascha Uhrig erstellt. Der Autor hat an der Publikation über die Analyse busbasierter Cache-Kohärenz mitgewirkt [137], deren Ergebnisse in diese Arbeit eingeflossen sind. Eine Vertiefung der Analyse um die Auswirkungen auf die Must-/May-Analyse sowie die Erweiterung um weitere Kohärenzprotokolle und Kommunikationsmedien wurde vom Autor durchgeführt.

- **Kapitel 4:** Die Grundidee zu dem in dieser Arbeit präsentierten ODC² Kohärenzverfahren stammt von Prof. Uhrig. Der Autor arbeitete an der Ausarbeitung des Verfahrens mit und war für die Realisierung bzw. Implementation des ODC² verantwortlich. Im Zuge dessen erweiterte der Autor das Verfahren um die Adressüberprüfung und den bedingten Wechsel der Schreibstrategie. Die Auswirkungen des ODC²-Verfahrens auf die Cacheanalyse einer statische WCET-Analyse wurden in Zusammenarbeit mit den Mitarbeitern des „Institute de Recherche en Informatique (IRIT)“ erarbeitet.
- **Kapitel 5:** Für die Laufzeitsimulation wurden verschiedene Modelle einer Mehrkern-Architektur eingesetzt, an deren Implementationen der Autor mitgewirkt hat. Basierend auf Standard-Komponenten der SoCLib-Plattform [131], implementierte der Autor Cache-Modelle, die den ODC² und das M(O)ESI-Protokoll in einem busbasierten System umsetzen. Ein Großteil der Laufzeitsimulation wurde mit Hilfe des parMERASA Simulators durchgeführt, der im Rahmen des parMERASA Projekts [138] von zahlreichen Personen entwickelt wurde. Der Autor implementierte für diesen Simulator das ODC² Cache-Modell und modifizierte zahlreiche anderen Modelle hinsichtlich den Anforderungen für diese Evaluation. Für die statische WCET-Analyse wurde die OTAWA Toolbox eingesetzt. Die Integration des ODC² Modells in OTAWA wurde von den Mitarbeitern von IRIT, allen voran Hugues Cassé durchgeführt. Der Autor hat diese Arbeit vor Ort unterstützt. Die Durchführung der WCET-Analyse wurde größtenteils vom Autor selbst bewerkstelligt. Die Analyse des 3DPP Benchmarks entstand mit Unterstützung von Haluk Ozaktas (IRIT).

Kapitel 2

Grundlagen

Dieses Kapitel dient als Einführung in das Themengebiet und vermittelt Grundlagen, die zum Verständnis der Arbeit hilfreich bzw. notwendig sind. Das Thema der Arbeit berührt mehrere Gebiete, so dass Ausführungen über Cache-Speicher, Mehrkern-Prozessoren sowie Echtzeitsysteme nötig sind.

Im Folgenden wird das Konzept einer Speicherhierarchie mit Cache-Speichern und deren Aufbau erläutert. Anschließend wird auf die Merkmale der Architektur eines Mehrkern-Prozessors eingegangen. Es folgt eine Definition der Begriffe Konsistenz und Kohärenz und eine Erläuterung der besonderen Rolle der Cache-Kohärenz in Mehrkern-Prozessoren. Der nachfolgende Abschnitt stellt klassische Cache-Kohärenzprotokolle und ihre Optimierungen vor. Dabei werden sowohl Hardware- als auch Softwarelösungen dargelegt. Abschließend werden die speziellen Anforderungen von Echtzeitsystemen erläutert. Es werden Grundlagen über die Laufzeitanalyse für Echtzeitsysteme vermittelt und im Speziellen auf die Cacheanalyse in einem Mehrkern-Prozessor eingegangen.

2.1 Cache-Speicher

Speicher für Programmdaten und Instruktionen in einem Computersystem müssen sowohl Anforderungen an hoher Speicherkapazität, als auch an kurzer Zugriffszeit genügen. Da diese Eigenschaften bei Speichern im Gegensatz zueinander stehen, kann keine Speicherart beiden Anforderungen zugleich gerecht werden. Daher wird eine Vielzahl verschiedener Speichertypen (z.B. SRAM, DRAM, Flash, Magnetspeicher etc.) verwendet, die miteinander zusammenarbeiten und eine Speicherhierarchie bilden. Abbildung 2.1 zeigt ein typisches

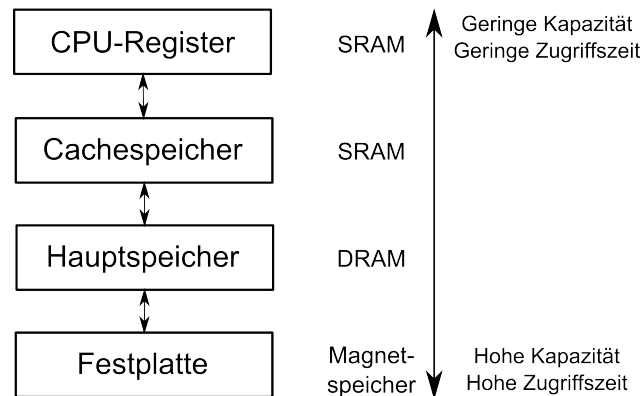


Abbildung 2.1: Aufbau und Merkmale einer Speicherhierarchie.

Beispiel einer Speicherhierarchie, wie sie in einem Computersystem zu finden ist. An oberen Ende der Hierarchie stehen die CPU-Register eines Rechenkerns, die einen sehr schnellen Zugriff, aber eine geringe Kapazität bieten. Das untere Ende bildet die Festplatte mit hoher Speicherkapazität und ebenso hoher Zugriffszeit.

Eine zentrale Rolle in der Speicherhierarchie nimmt hierbei der Hauptspeicher ein, der die Instruktionen und Daten eines Programms während dessen Ausführung bereitstellt. Ein Zugriff auf den Hauptspeicher, typischerweise ein DRAM Speicher, benötigt sehr viel Zeit, verglichen mit einem Zugriff auf die internen Register eines Rechenkerns. Um diese Zugriffe zu beschleunigen wird ein Cache-Speicher dazwischen geschaltet. Ein Cache ist ein kleiner und schneller SRAM Speicher mit einer Kapazität von wenigen Kilobyte bis einigen Megabyte, der zur Zwischenspeicherung von Daten verwendet wird.

Die Aufgabe eines Cache-Speichers ist es, Kopien der momentan benötigten Instruktionen und Daten vom Hauptspeicher zu laden und temporär für einen schnellen Zugriff bereitzuhalten. Dieser Vorgang wird, für den Benutzer nicht sichtbar, vom Cache-Controller durchgeführt. Dabei macht er sich die Prinzipien der zeitlichen und räumlichen Lokalität zu Nutze. Das Prinzip der räumlichen Lokalität besagt, dass die benachbarten Daten eines aktuell verwendeten Datums, mit hoher Wahrscheinlichkeit in Kürze benötigt werden. Daher werden beim Laden eines neuen Datums in den Cache auch eine bestimmte Anzahl benachbarter Daten mit in die Cache-Zeile geladen. Dem Prinzip der zeitlichen Lokalität zufolge, wird auf ein aktuell verwendetes Datum mit hoher Wahrscheinlichkeit in Kürze erneut zugegriffen. Aus diesem Grund wird bei der Ersetzung von Daten im Cache versucht, diejenigen Daten zu ersetzen, auf die voraussichtlich in naher Zukunft nicht mehr zugegriffen wird.

Organisatorisch ist ein Cache in eine Vielzahl von Cache-Blöcken unterteilt.

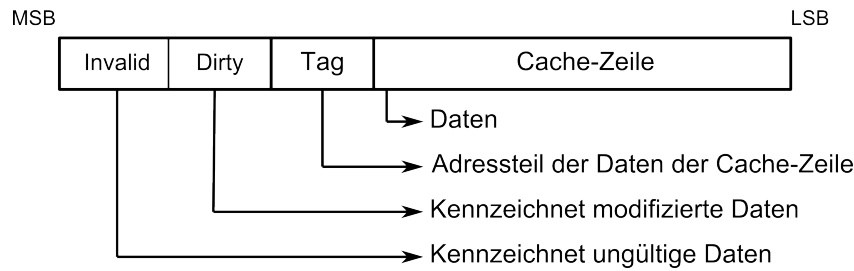


Abbildung 2.2: Aufbau eines Cache-Blocks mit Kontrollinformationen.

Abbildung 2.2 zeigt den Aufbau eines solchen Cache-Blocks, bestehend aus den Kontrollinformationen zur Verwaltung des Zustands des Cache-Blocks (Dirty-Bit, Invalid-Bit), dem Tag (Teil der Adresse der gespeicherten Daten) und der Cache-Zeile, in der mehrere Datenwörter gespeichert werden.

Ablauf eines Cache-Zugriffs

Ein Cache-Speicher dient bei der Beantwortung eines Speicherzugriffs als erste Instanz, d.h. der Prozessor richtet seine Lese- oder Schreibanfrage direkt an den Cache. Dieser leitet die Anfrage gegebenenfalls an den Hauptspeicher weiter. Häufig existieren mehrere Ebenen von Cache-Speichern in einer Speicherhierarchie. Anfragen werden dann, wenn es notwendig ist, an die jeweils nächste Cache-Ebene weitergeleitet. Im Folgenden wird der Ablauf eines Cache-Zugriffs bei nur einer existierenden Cache-Ebene erläutert.

Erhält der Cache eine Anfrage für einen Speicherzugriff, so prüft dieser ob das gewünschte Datum im Cache vorhanden ist, indem er zuerst die anliegende Zieladresse in drei Segmente aufgeteilt. Abbildung 2.3 zeigt die Gliederung einer Cache-Adresse mit den Segmenten Tag, Index und Offset. Mit Hilfe des Index wird die genaue Position, d.h. der konkrete Block, an dem die Daten im Cache gespeichert werden, bestimmt. Dabei wird zwischen den Organisationsarten direkt-abgebildet, satz-assoziativ und voll-assoziativ unterschieden. Bei direkt-abgebildeten Caches wird über den Index ein einzelner konkreter Cache-Block adressiert. In satz-assoziativen Caches adressiert der Index einen Cache-Satz, der aus mehreren Cache-Blöcken besteht. Innerhalb dieses Cache-Satzes kann der Cache-Block frei gewählt werden. So stehen bei einem 4-fach assoziativen Cache 4 Cache-Blöcke pro Satz zur Auswahl. Ein voll-assoziativer Cache besteht nur aus einem einzigen Satz, der alle vorhandenen Cache-Blöcke enthält. Darin wird der Ziel-Block frei bzw. von Cache-Controller eigenständig gewählt, so dass kein Index mehr notwendig ist. Wurde der entsprechende Cache-Block für die Zieladresse eines Speicherzugriffs ermittelt, wird über den Tag geprüft, ob die Cache-Zeile das gewünschte Datum enthält. Ist dies der

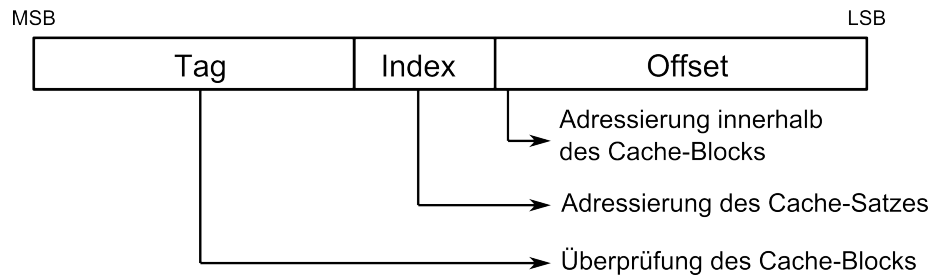


Abbildung 2.3: Gliederung einer Cache-Adresse in die Segmente Tag, Index und Offset.

Fall, so wird mit Hilfe des Offsets das gesuchte Datum innerhalb der Cache-Zeile adressiert.

Ist das gesuchte Datum eines Speicherzugriffs im Cache vorhanden, so spricht man von einem Cache-Hit und der Zugriff kann lokal im Cache durchgeführt werden. Im Falle eines Lesezugriffs wird das Datum vom Cache an der Prozessor übermittelt. Bei einem Schreibzugriff wird zwischen einer Write-Back und einer Write-Through Schreibstrategie unterschieden. Ein Write-Back Schreibzugriff modifiziert lediglich das Datum im Cache und markiert die Cache-Zeile in den Kontrollinformationen des Cache-Blocks als „modifiziert“ (*Dirty*). Diese Markierung gibt an, dass diese Kopie des Datums im Cache nicht mehr konsistent zum Datum im Hauptspeicher ist, d.h. im Hauptspeicher befindet sich ein anderer, veralteter Wert. Bei einer späteren Ersetzung der Cache-Zeile müssen die Daten dann in den Hauptspeicher zurückgeschrieben werden. Bei einem Write-Through Schreibzugriff wird zusätzlich zum Cache auch das Datum im Hauptspeicher geändert. Dabei bleibt das Datum im Cache konsistent zum Hauptspeicher und es ist nicht nötig es als „modifiziert“ zu markieren.

Ein Zugriff auf ein Datum, das nicht im Cache vorhanden ist, wird als Cache-Miss bezeichnet. Das gewünschte Datum muss nun erst vom Hauptspeicher in den Cache geladen werden. Dabei wird in der Regel mittels einer *Read-Burst* die gesamte Cache-Zeile, ausgerichtet an der Größe des Offsets, in den Cache geladen. Anschließend kann auf das Datum im Cache lesend oder schreibend zugegriffen werden. Für einen Schreibzugriff wird dieser Ablauf als Write-Allocate bezeichnet. Alternativ kann eine Schreiboperation bei einem Cache-Miss als Non-Write-Allocate Zugriff durchgeführt werden. Dabei wird der modifizierte Wert direkt in den Hauptspeicher geschrieben, ohne vorher die zugehörige Cache-Zeile zu laden. Ein weiterer Zugriff auf dieses Datum führt dann wieder zu einem Cache-Miss.

Wenn im Falle eines Cache-Miss im entsprechenden Cache-Satz (bei Satz-Assoziativität) kein freier Cache-Block mehr zur Verfügung steht, muss einer der vorhandenen Cache-Blöcke ersetzt werden. Die Entscheidung darüber

welcher Block ersetzt wird, wird anhand der eingesetzten Ersetzungsstrategie getroffen. Es existiert eine Vielzahl verschiedener Ersetzungsstrategien, die versuchen das Prinzip der zeitlichen Lokalität bestmöglich auszunutzen. Dazu wird auf unterschiedliche Weise das „Alter“ eines Cache-Blocks ermittelt. Im Falle einer Ersetzung kann dann der älteste Block ausgewechselt werden. Gängige Ersetzungsstrategien sind *Least-Recently-Used* (LRU), *First-In-First-Out* (FIFO) und *Random*. Aufgrund ihrer Komplexität werden oft Implementierungen eingesetzt, die dieses Verfahren bestmöglich approximieren (z.B. pseudo-LRU). Wurde die Wahl des zu ersetzenden Cache-Blocks getroffen, wird zuerst überprüft, ob die Cache-Zeile als *Dirty* markiert ist und gegebenenfalls zurückgeschrieben werden muss. Anschließend wird die Cache-Zeile invalidiert, d.h. als ungültig markiert und, wie oben beschrieben, der Cache-Miss durchgeführt. Bei direkt-abgebildeten Cache-Speichern besteht keine Wahl zwischen mehreren zu ersetzenden Cache-Zeilen und eine Ersetzungsstrategie ist daher unnötig.

Alle gängigen Cache-Speicher bieten zusätzliche Möglichkeiten an, den Inhalt des Caches zu modifizieren. So ist es z.B. möglich mit Hilfe einer speziellen Instruktion sämtliche Daten aus dem Cache-Speicher zu löschen, um eine vollständige Konsistenz zum Hauptspeicher und einen freien Cache zu erhalten. Dabei werden die Daten in den Cache-Zeilen nicht buchstäblich gelöscht, sondern lediglich über das *Invalid-Bit* als ungültig markiert. Das *Invalid-Bit* ist, ebenso wie das *Dirty-Bit*, Teil der Kontrollinformationen eines Cache-Blocks (siehe Abbildung 2.2).

2.2 Mehrkern-Prozessoren

Computersysteme mit mehreren parallel arbeitenden Prozessoren gehen bis in die 1960er Jahre zurück. So gab es z.B. den IBM System 360/67 [81] als Dual-Processor Modell. In den nachfolgenden Jahren wurden immer komplexere Systeme mit teilweise hunderten von Prozessoren entwickelt, u.a. den BBN Butterfly [83].

Ein Mehrkern-Prozessor unterscheidet sich in erster Linie dadurch von einem Mehrprozessorsystem, dass die Rechenkerne zusammen auf dem selben Mikrochip verbaut sind. Man kann Mehrkern-Prozessoren als eine konsequente Weiterentwicklung von Mehrprozessorsystemen betrachten. Beide Systemtypen stellen Architekturen mit separaten Recheneinheiten und gemeinsam genutzten Ressourcen dar. Im Falle eines Mehrkern-Prozessors befinden sich sämtliche Rechenkerne auf einem Chip. Sie sind dadurch deutlich enger gekoppelt und teilen sich mehr Ressourcen als in einem Mehrprozessorsystem. Dadurch ist eine deutlich höhere Performanz und Energieeffizienz möglich. Eine andere Sichtweise ist es, Mehrkern-Prozessoren als Weiterentwicklung der simultanen Mehrfädigkeit [135] zu betrachten. Aus diesem Blickwinkel sind die

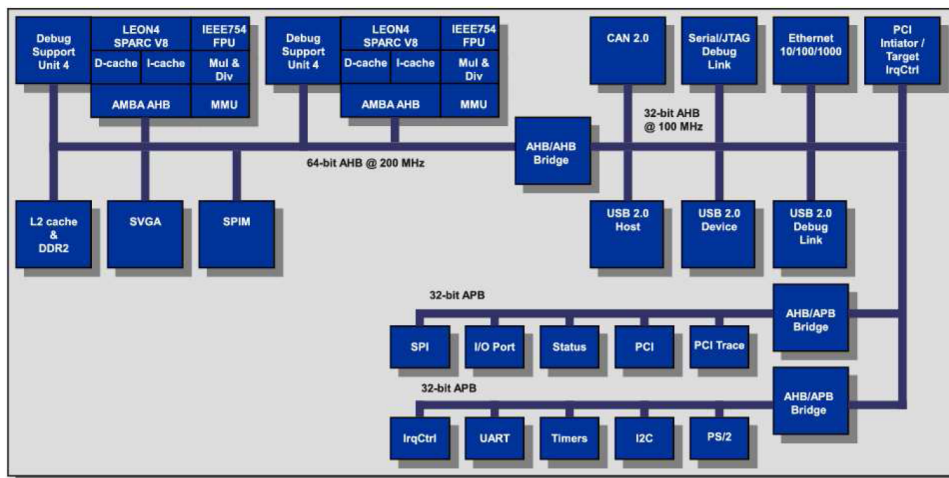


Abbildung 2.4: Blockdiagramm des LEON4 SPARC V8 Prozessor von Aeroflex Gaisler [4].

Recheneinheiten in einem Mehrkern-Prozessor weniger eng gekoppelt, teilen sich weniger Ressourcen und erreichen dadurch eine höhere Parallelität. Unabhängig von der Betrachtungsweise gelten Mehrkern-Prozessoren heutzutage als die effizienteste Methode Parallelität in der Ausführung von Programmen zu realisieren.

2.2.1 Aufbau eines Mehrkern-Prozessors

Die Rechenkerne eines Mehrkern-Prozessors beinhalten eine eigene Ausführungspipeline incl. Registersatz und besitzen in der Regel einen privaten Daten- und Instruktionscache. Weitere Speicher, Peripheriemodule sowie Ein-/Ausgabeschnittstellen stehen mehreren oder allen Rechenkernen als gemeinsame Ressourcen zur Verfügung. Eine weitere gemeinsame Ressource ist das Verbindungsmedium, das der Kommunikation der Rechenkerne untereinander und mit der Peripherie dient. Die Wahl des Kommunikationsmediums ist ein wichtiger Faktor, sowohl für die Performanz des Systems, als auch für den Energieverbrauch und die Größe der Chipfläche (siehe Kumar et al. [76]).

Die in gängigen Mehrkern-Architekturen eingesetzten Kommunikationsmedien sind Bus, Crossbar und netzwerkartige Verbindungen (vgl. Bjerregaard et al. [19]). Abbildung 2.6 illustriert die Topologie dieser Kommunikationstypen. Mehrkern-Prozessoren mit einer geringen Anzahl an Rechenkernen (ca. 2-8), wie z.B. der LEON4 SPARC V8 Prozessor [3] (siehe Abbildung 2.4), werden häufig über einen gemeinsamen Bus verbunden. Da alle Rechenkerne auf die selbe Kommunikationsleitung zugreifen, ist eine Arbitrierung der Zugriffe durch einen Bus-Controller notwendig. Ein Vorteil busbasierter Systeme

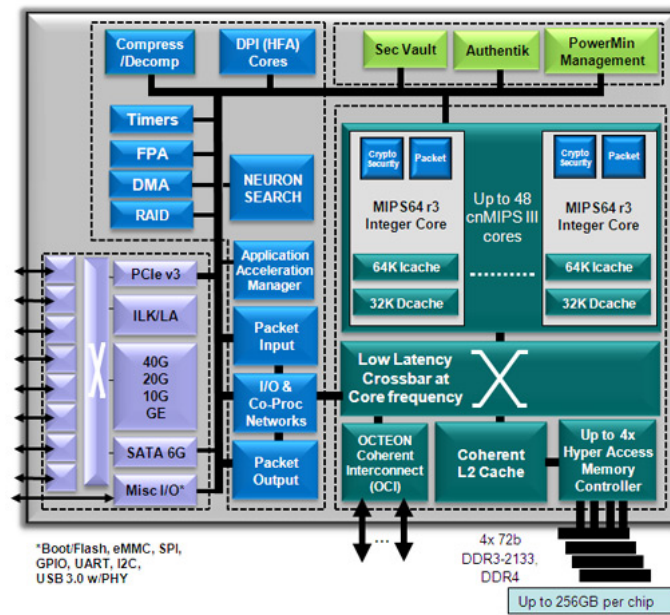


Abbildung 2.5: Blockdiagramm der OCTEON III CN7XXX Architektur von Cavium [25].

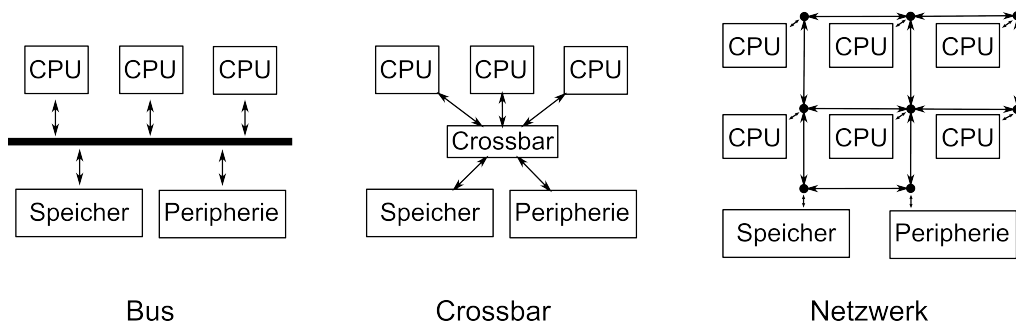


Abbildung 2.6: Veranschaulichung der Verbindungstypen Bus, Crossbar und Netzwerk.

me ist die effiziente Umsetzung von Broadcast-Nachrichten, d.h. Nachrichten, die an alle Teilnehmer im System gesendet werden. In anderen Architekturen, wie dem OCTEON III CN7XXX [73] (siehe Abbildung 2.5) oder dem Freescale P4080 [21], wird eine Crossbar eingesetzt. Im Gegensatz zu einem Bus, sind in einer Crossbar separate Punkt-zu-Punkt Verbindungen der Rechenkern untereinander und zu relevanten Peripheriekomponenten, wie gemeinsamen Speicher, vorhanden. Dies ermöglicht eine erhöhte Parallelität bei Speicherzugriffen, bringt jedoch einen signifikanten Overhead an Chipfläche mit sich (vgl. Kumar et al. [76]). Für Architekturen mit einer besonders hohen Anzahl von Rechenkernen (32+), wie der TilePro64 von Tileria [134] (siehe

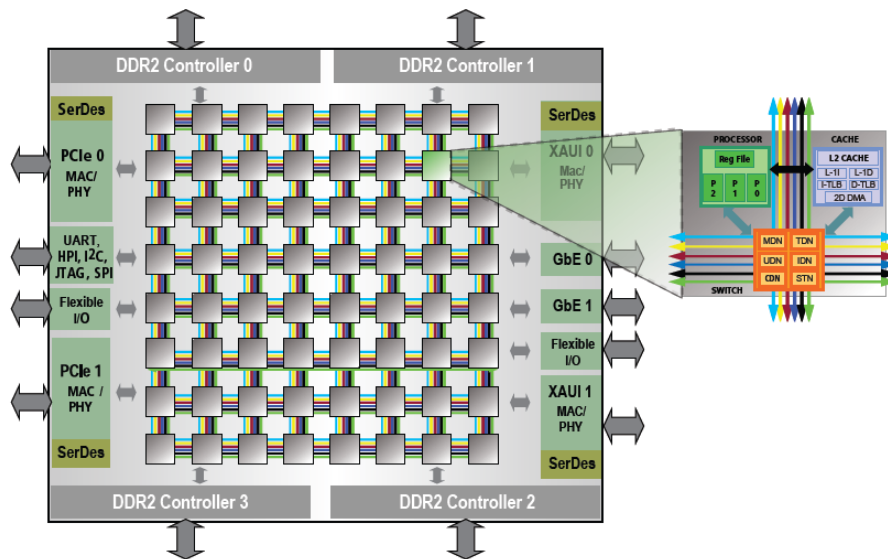


Abbildung 2.7: Blockdiagramm der TilePro64 Architektur von Tiler [133].

Abbildung 2.7), eignen sich gitterförmige On-Chip Netzwerke. Hierbei sind benachbarte Rechenkerns mittels Router über Punkt-zu-Punkt Verbindungen miteinander verbunden und nutzen ein paketbasiertes Kommunikationsprotokoll. Eine solche Architektur ermöglicht eine hohe Skalierbarkeit und die Integration von 100 oder mehr Rechenkernen.

2.2.2 Speicherhierarchie

Ein kennzeichnendes Merkmal von Mehrkern-Architekturen ist der Aufbau der Speicherhierarchie und die Abbildung des Adressraums auf die vorhandenen Speicher. Es werden verschiedene Ansätze unterschieden, wie in Programmen auf gemeinsamen Speicher zugegriffen wird. In aktuellen Implementierungen hat sich das *Shared Memory Model* [2] durchgesetzt. Bei diesem Speichermodell verfügt das System über einen gemeinsamen physikalischen Adressraum, der jedem Rechenkern zur Verfügung steht. Damit wird ein einziger gemeinsamer Speicher abstrahiert, der in Wirklichkeit aus mehreren physikalischer Speichermodulen besteht (*Distributed Shared Memory*), die über das System verteilt sind. Die Umsetzung der Speicherzugriffe vom gemeinsamen Adressraum auf die verteilten Speicher liegt in der Verantwortung der *Memory Management Unit* (MMU).

Man unterscheidet in einem Mehrkern-Prozessor zwischen On-Chip und Off-Chip Speichern. Der Hauptspeicher ist, auch in Mehrkern-Prozessoren, zumeist als gemeinsamer Off-Chip DRAM realisiert. Mit steigender Anzahl an Rechenkernen wird der Zugriff auf einen gemeinsamen Off-Chip Speicher un-

verhältnismäßig teuer, im Sinne der Zugriffszeit. Das liegt zum einen daran, dass mit wachsender Chipfläche die Distanz eines Rechenkerns zum Off-Chip RAM länger wird, zum anderen muss mit mehr Rechenkernen um den Zugriff auf den Speicher konkurriert werden. Daher spielen in Mehrkern-Prozessoren lokale On-Chip Speicher eine entscheidende Rolle. Sie stehen z.B. als zusätzlicher gemeinsamer Speicher für eine kleinere Gruppe bzw. Cluster von Rechenkernen zur Verfügung (wie z.B. beim Kalray MPPA256 [39]). Andererseits besitzen Rechenkerne ihre privaten On-Chip Speicher, realisiert als Cache und/oder Scratchpad-Speicher [15]. Ziel dabei ist es, die Häufigkeit von Off-Chip Zugriffen zu minimieren.

Hinsichtlich einer konkreten Realisierung einer Speicherhierarchie existieren verschiedene Ansätze. Die Verwendung eines privaten Instruktions- und Datencaches (L1-Cache) haben alle gängigen Architekturen gemeinsam, der Zugriff auf die darauffolgende Speicher-Ebene wird dagegen unterschiedlich gehandhabt. Möglich ist ein gemeinsamer L2-Cache, wie im Cavium Octeon oder der LEON4 Architektur, sowie private L2-Caches für jeden Rechenkern wie im Freescale P4080. Diese können wie in Tileras TilePro Architektur einen verteilten gemeinsamen L3-Cache bilden. Solche Architekturen werden auch als *Non-Uniform-Cache-Architecture* (NUCA) bezeichnet. Alternativ wird in einigen Systemen gänzlich auf einen L2-Cache verzichtet und stattdessen ein gemeinsamer SRAM Speicher genutzt (z.B. bei Infineon Aurix [65] und Kalray MPPA256).

Wie an den zuvor erwähnten Architekturen zu sehen ist, ist der genaue Aufbau eines Mehrkern-Prozessors abhängig von den Anforderungen, die an ihn gestellt werden und kann daher schlecht verallgemeinert werden. Die Anforderungen an einen Prozessor für den Einsatz in einem Echtzeitsystem sind aber sehr strikt und grenzen die Wahlmöglichkeiten sichtlich ein. Auf den Aufbau einer Mehrkern-Architektur für Echtzeitsysteme wird in Kapitel 2.5.4 genauer eingegangen.

2.3 Konsistenz und Kohärenz in Mehrkern-Prozessoren

Für eine korrekte Durchführung von Speicherzugriffen in einem Computersystem sind die Begriffe Konsistenz und Kohärenz ausschlaggebend. Sie definieren den Zustand von Daten im System sowie die Korrektheit der Zugriffe darauf. Bevor das Problem der Kohärenz erläutert werden kann ist eine detaillierte Ausführung der Konsistenz notwendig.

Die Begriffe Konsistenz und Kohärenz lassen sich sowohl im Kontext eines Mehrprozessorsystems als auch eines Mehrkern-Prozessors erläutern. Im Grunde tritt die Problematik in beiden Systemtypen in gleicher Weise auf. Da

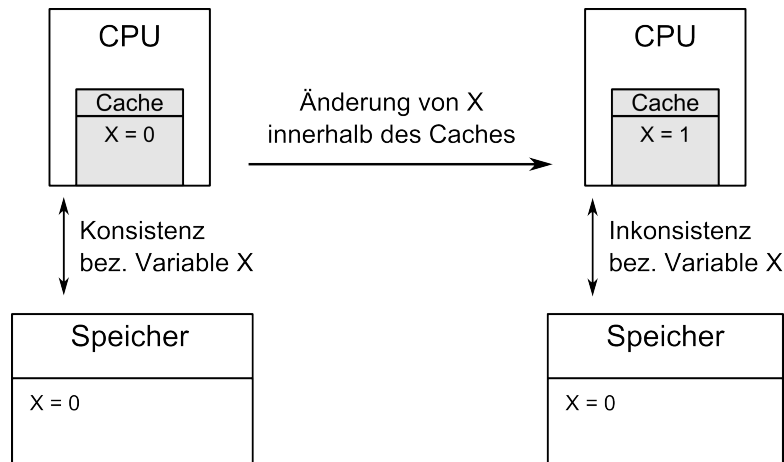


Abbildung 2.8: Veranschaulichung der Dateninkonsistenz in einem Einkern-Processor mit privaten Cache-Speichern.

der Fokus dieser Arbeit auf Mehrkern-Processoren liegt, werden die Konzepte als auch die Lösungsansätze anhand eines Mehrkern-Processors erläutert, wenngleich sie zum Teil für Mehrprozessorsysteme entwickelt wurden. Außerdem beschränkt sich die Betrachtung der Problematik in dieser Arbeit auf die Korrektheit beim Zugriff auf gemeinsame Daten. Private Daten können ebenso dem Kohärenzproblem unterliegen, wenn die Ausführung eines Programms im zeitlichen Wechsel auf mehreren Rechenkernen geschieht (*Prozess Scheduling*). Doch Prozess Scheduling findet in Echtzeitsystemen in der Regel keine Anwendung und die Problematik ist aus der Sicht des Prozessors in beiden Fällen die gleiche.

In dieser Arbeit werden die Begriffe „gemeinsame Daten“ sowie „geteilte Daten“ verwendet. Ist von gemeinsamen Daten die Rede, so sind Daten gemeint, die potentiell von mehreren Rechenkernen verwendet werden, als Abgrenzung zu privaten Daten. Geteilte Daten hingegen stellen konkrete Kopien von gemeinsamen Daten dar, die sich momentan in mehreren Cache-Speichern befinden.

2.3.1 Konsistenz und Konsistenzmodelle

Der Begriff der Konsistenz beschreibt den Zustand mehrerer Instanzen eines Datums in einem Speichersystem. Betrachten wir zunächst einen Einkern-Processor mit einfacher Speicherhierarchie (Hauptspeicher und L1-Cache). In einem solchen System kann ein einzelnes Datum in mehreren Hierarchie-Ebenen zugleich vorhanden sein, zum Beispiel wenn das Datum vom Hauptspeicher in den Cache kopiert wird. Es existieren dann mehrere Instanzen bzw. Kopien dieses Datums. Von einer Dateninkonsistenz spricht man, wenn zu ei-

nem bestimmten Zeitpunkt Kopien des Datums mit unterschiedlichen Werten existieren. Dies geschieht z.B. bei Caches die mit einer Write-Back Strategie arbeiten. Die Änderung des Wertes eines Datums im Cache wird dabei nicht an den Hauptspeicher weitergegeben. Das Datum im Hauptspeicher hat noch den alten Wert und wird erst zu einem späteren Zeitpunkt aktualisiert (siehe Abbildung 2.8). Dateninkonsistenz ist ein wesentliches Merkmal einer Speicherhierarchie und keine Fehlersituation. Die Inkonsistenz ist nur temporär und es wird zu gegebener Zeit wieder eine Konsistenz hergestellt.

Eine weitere Facette der Konsistenz betrifft die Reihenfolge von Datenzugriffen. Eine Sequenz von Datenzugriffen ist konsistent, wenn sie in der vom Programm vorgesehen Reihenfolge durchgeführt wird. In modernen Architekturen können Schreib-Puffer und spezielle Pipeline-Techniken eine Veränderung der vorgesehenen Reihenfolge herbeiführen. In einem Mehrkern-Prozessor gewinnt diese Art der Konsistenz eine weitere Dimension. Zusätzlich zur Reihenfolge der Zugriffe innerhalb eines Rechenkerns, ist ebenso die Reihenfolge von Speicherzugriffen mehrerer Rechenkerns entscheidend. Man kann bei paralleler Ausführung zwar nicht mehr von einer global sequentiellen Befehlsfolge sprechen, dennoch ist eine korrekte Abfolge einzelner Befehlssequenzen einzuhalten. Die Konsistenz von Datenzugriffen muss dann rechenkernübergreifend betrachtet werden.

Die erläuterten Situationen, in denen eine Dateninkonsistenz auftritt, führen, wie bereits erwähnt, nicht unweigerlich zu einem Fehlerfall. Es ist also nicht notwendig, dass zu jeder Zeit eine vollständige Konsistenz besteht. Vielmehr sind Maßnahmen zu ergreifen, die eine korrekte Ausführung, auch bei zeitweiliger Inkonsistenz, garantieren. Diese Maßnahmen können entweder vom System, dem Benutzer oder von beiden gemeinsam ergriffen werden. Um die Aufteilung der Verantwortlichkeiten für die Sicherstellung eines korrekten Ablaufs von Speicherzugriffen für den Benutzer transparent zu machen, wurden verschiedene Konsistenzmodelle eingeführt [54][2]. Das einer Ausführung zugrunde liegende Konsistenzmodell ist ausschlaggebend für Sicherstellung der Kohärenz.

Im *Sequential Consistency* Modell nach Lamport [80] müssen für die korrekte Ausführung von Programmen in Mehrkern-Prozessoren folgende Bedingungen erfüllt sein:

1. Das Resultat einer Ausführung ist immer gleich, so als ob die Operationen aller Rechenkerns in einer sequenziellen Reihenfolge ausgeführt werden, und
2. die Operationen eines jeden Rechenkerns müssen, in der von Programm vorgegebenen Reihenfolge, für das System sichtbar sein.

Dies darf nicht als Bedingung missverstanden werden, dass sämtliche Befehle eines Rechenkerns in der im Programm vorgeschriebenen Reihenfolge stattfinden müssen. Aus Sicht der anderen Rechenkerne im System muss dies lediglich so erscheinen. Intern ist es zu einem gewissen Grad erlaubt von der Reihenfolge abzuweichen. Es muss lediglich sichergestellt werden, dass die vorgesehene Folge von Befehlssequenzen der parallelen Ausführung eingehalten wird. Dieses sehr strikte Modell verlangt eine Geradlinigkeit und Transparenz von Datenzugriffen, wie sie implizit in einem Einkern-Prozessor gegeben ist und erzielt damit ein deterministisches Verhalten. Die Umsetzung dieses Modells unterbindet zahlreiche Optimierungen der Mikroarchitektur sowie der Codegenerierung. So ist z.B. das Umsortieren oder Puffern von Schreib- und Lesezugriffen zu Gunsten einer schnelleren Ausführung untersagt. Daher gilt das Sequentiell Consistency Modell als hinderlich für Systeme die auf eine hohe Performanz ausgelegt sind und findet bei der Gestaltung von Mehrkern-Prozessoren in der Regel keine Anwendung. Hill [61] zufolge wird jedoch das Sequentiell Consistency Modell strenger ausgelegt, als es von seiner Definition her notwendig wäre. Er argumentiert, dass das Sequentiell Consistency Modell hinreichend Möglichkeiten zur Anwendung von Optimierungen bietet und den Programmierer zum Teil von der Verantwortung einen korrekten Programmablauf sicherzustellen entlastet.

Weniger strenge Modelle werden unter dem Begriff *Relaxed Consistency* zusammengefasst. Solche Modelle erlauben eine zeitweilige Inkonsistenz und die Durchführung von Datenzugriffen losgelöst von der tatsächlichen Reihenfolge. Gharachorloo et al. [52] definiert mit *Release Consistency* ein Konsistenzmodell, welches heute überwiegend in Mehrkern-Prozessoren angewendet wird. Es ist ein Modell mit sehr hohem Freiheitsgrad bezüglich der Durchführung von Datenzugriffen und verlangt zugleich den Einsatz von Techniken zur Synchronisation, um eine fehlerfreie Ausführung zu gewährleisten. Speicherzugriffe auf gemeinsame Daten müssen, im Falle eines möglichen konkurrierenden Zugriffs, durch Synchronisationstechniken wie Mutex, Semaphoren oder Barrieren geschützt sein. Zugleich muss der korrekte Zugriff auf die verwendeten Synchronisationsprimitiven gewährleistet sein. Gemäß dem Release Consistency Model dürfen Speicherzugriffe umsortiert werden. Dies kann beim Zugriff auf Synchronisationsvariablen zu einem Fehlschlagen der Synchronisation führen. Aus diesem Grund verfügen die Befehlssätze solcher Architekturen über spezielle atomare Operationen (wie z.B. Read-Modify-Write). Solche atomaren Operationen sind Zugriffssequenzen, die das Lesen und anschließende Schreiben eines Datums als einzelnen Zugriff realisieren und nicht unterbrochen werden können. Sie sind erforderlich, um das Konsistenzmodell korrekt anwenden zu können und sind fester Bestandteil des Instruktionssatzes aktueller Mehrkern-Prozessoren.

Die Architektur heutiger Mehrkern-Prozessoren und die dafür verwen-

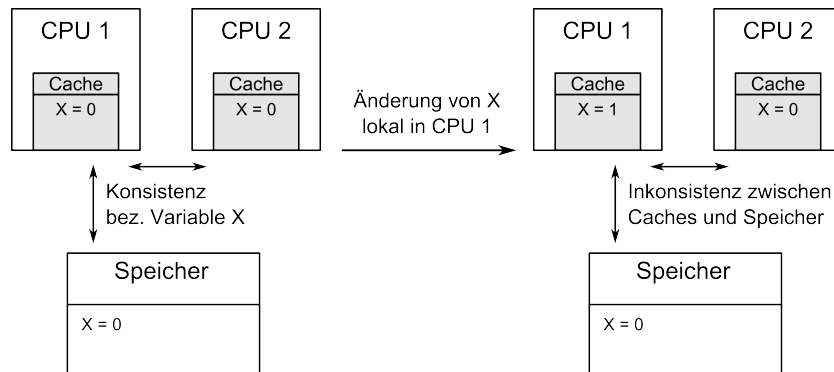


Abbildung 2.9: Veranschaulichung eines inkohärenten Lesezugriffes in einem Mehrkern-Prozessor mit privaten Cache-Speichern.

deten Programmiermodelle wie OpenMP oder POSIX-Threads entsprechen weitestgehend einem Relaxed Consistency Modell. Verfügen die Mehrkern-Prozessoren über Cache-Speicher, so wird die Anwendung eines Cache-Kohärenzverfahrens vorausgesetzt.

2.3.2 Cache-Kohärenz

Der Begriff Kohärenz beschreibt die Korrektheit eines Zugriffs eines Rechenkerns auf ein Datum. Wenn ein Datum im System konsistent ist, d.h. alle Instanzen besitzen den gleichen Wert, so ist der Zugriff darauf stets kohärent. Sind die Kopien eines Datums inkonsistent, so kann ein inkohärenter und damit fehlerhafter Zugriff auftreten. Um eine korrekte Programmausführung sicherzustellen zu können, müssen inkohärenter Zugriffe zwingend vermieden werden.

Im folgenden Beispiel soll das Auftreten eines inkohärenten Zugriffs in einem Mehrkern-Prozessor mit privaten Cache-Speichern erläutert werden. Greifen zwei Rechenkern im Laufe der Ausführung auf das selbe Datum zu, so befindet sich jeweils eine Kopie dieses Datums in beiden Caches (siehe Abbildung 2.9, linke Seite). Ändert nun der erste Rechenkern den Wert des Datum in seinem Cache, so existiert nicht nur eine Inkonsistenz der Daten zwischen Hauptspeicher und Cache, sondern auch zwischen den beiden Cache-Speichern (siehe Abbildung 2.9, rechte Seite). Der Wert des Datums im zweiten Cache ist nun veraltet. Der zweite Rechenkern hat keinen Zugriff auf den geänderten Wert, seine Zugriffe auf das Datum sind nun inkohärent.

Der Begriff der Inkohärenz lässt sich in Anlehnung an Veidenbaum [141] wie folgt formalisieren:

Ein Lesezugriff eines Rechenkerns auf ein Datum ist inkohärent, wenn der gelesene Wert nicht identisch mit dem zuletzt auf dieses

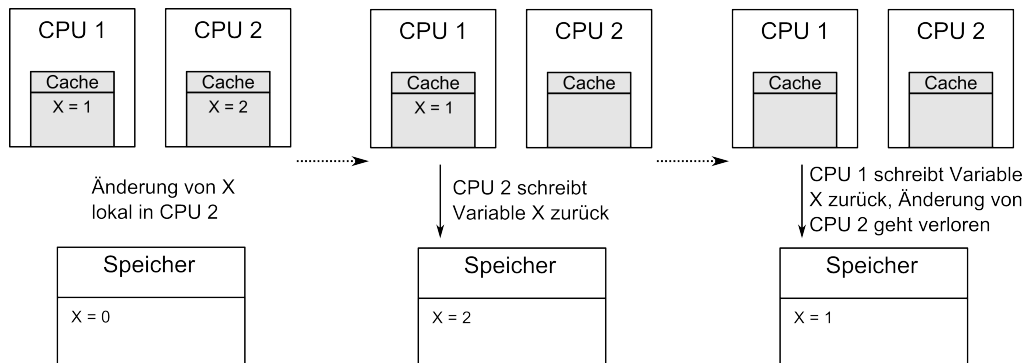


Abbildung 2.10: Veranschaulichung eines Datenverlusts bei Inkonsistenz in Cache-Speichern eines Mehrkern-Prozessors.

Datum geschriebenen Wertes ist.

Der Begriff Datum steht hierbei nicht für eine konkrete Repräsentation einer Information, sondern für die logische Instanz eines Datums, das in mehreren Ausprägungen bzw. Kopien im System vorliegen kann. Inkohärenz ist nach dieser Definition auf Lesezugriffe beschränkt und kann nicht bei einem Schreibzugriff auftreten. Damit ist Inkohärenz die Folge einer zuvor aufgetretenen Inkonsistenz von Kopien eines Datums im System.

Inkohärenz kann auch als Folge eines durch Inkonsistenz verursachten Datenverlust auftreten. Ausgehend vom vorherigen Beispiel wird dies in Abbildung 2.10 dargestellt. Beide Rechenkerne haben die selbe Cache-Zeile mit Datum X geladen und nacheinander modifiziert. Unabhängig von der Reihenfolge in denen die Rechenkerne die Änderungen an Datum X durchgeführt haben, wird die Reihenfolge in denen das Datum in den gemeinsame Speicher zurückgeschrieben wird vom Cache bestimmt, abhängig davon wann die jeweilige Cache-Zeile ersetzt werden muss. Die zuletzt von CPU 2 durchgeführte Änderung von Datum X geht dabei verloren. Dieses Problem ist in der Programmierung als Wettlaufsituation (*Race Condition*) bekannt. Die im vorherigen Kapitel erörterten Konsistenzmodelle untersagen das Auftreten einer Wettlaufsituation. Doch die dazu eingesetzten Synchronisierungstechniken wirken sich nicht auf Zugriffe aus, die vom Cache-Speicher initiiert werden. Das Zurückschreiben der Daten in den Hauptspeicher wird vom Cache selbst ausgelöst und kann von den konventionellen Synchronisationsprimitiven nicht beeinflusst werden. So wird die durch die Synchronisation vorgegebene Reihenfolge von Datenzugriffen, durch Caches in gewisser Weise umgangen. Um die daraus resultierenden Probleme zu unterbinden, wurden Lösungen entwickelt, die eine Kohärenz der Datenzugriffe mit Caches sicherstellen.

2.4 Cache-Kohärenzprotokolle

Das Problem der Cache-Kohärenz ist bereits seit den frühen Anfängen der Mehrprozessorsysteme Gegenstand der Forschung [149]. In den 80er Jahren war die Parallelisierung von Rechnersystemen ein großes Thema und entsprechende Mehrprozessorsysteme (Multimax, Firefly, RP3 etc.) waren, hauptsächlich in Universitäten und Forschungseinrichtungen, zu finden. Die in diesen Architekturen eingesetzte Speicherhierarchie, ist mit der in heutigen Mehrkern-Prozessoren vergleichbar. Mehrere Prozessoren besaßen private Cache-Speicher und griffen zusätzlich auf einen gemeinsamen Speicher zu. So war es bereits damals notwendig, sich Gedanken über die Lösung des Kohärenzproblems zu machen. Die dabei entwickelten Konzepte bzw. Lösungen bilden den Grundstein für die heute verwendeten Kohärenzverfahren.

Es wird grundlegend zwischen Hardware- und Software-Kohärenzverfahren unterschieden. Hardware-Kohärenzverfahren basieren auf einer erweiterten Logik im Cache-Speicher und im Kommunikationssystem. Sie realisieren kohärente Zugriffe ohne Zutun der Software bzw. eines Betriebssystems, sondern mit Hilfe hardware-gesteuerter Kommunikation zwischen den beteiligten Caches. Diese Kommunikation besteht aus Kohärenznachrichten, die zum Teil eingebettet in Speicherzugriffe, den Kohärenzzustand gemeinsamer Daten bekanntgeben und modifizieren. Ein Hardware-Kohärenzverfahren hat den Vorteil, dass die Erhaltung von kohärenten Cache-Zugriffen vollständig von der Hardware sichergestellt wird, und somit nicht in der Verantwortung des Benutzers liegt. Entsprechend hoch ist der Aufwand an zusätzlicher Hardwarelogik, die zur Realisierung der Kohärenzoperationen nötig ist. Die konkrete Realisierung hängt stark von der verwendeten Architektur des Computersystems ab. Der Archetyp eines Hardware-Kohärenzprotokolls ist das MESI-Protokoll.

2.4.1 Snooping-Protokolle

Das sogenannte MESI-Konzept zur Beschreibung des Zustands eines Cache-Blocks kann als Paradigma für die Handhabung der Cache-Kohärenz gesehen werden, da es als Grundlage für nahezu alle gängigen Hardware Kohärenzverfahren dient. Das Konzept entspricht im Grunde dem erstmals vom Goodman vorgestellten Write-Once Protokoll [53], jedoch hat sich die als MESI oder auch Illinois Protokoll [98] bezeichnete Variante durchgesetzt. MESI (Modified, Exclusive, Shared, Invalid) bezeichnet die möglichen Zustände einer Cache-Zeile bei der gemeinsamen Verwendung im Mehrkern-Prozessor. Der Zustand *Exclusive* kennzeichnet eine Cache-Zeile als einzige und unveränderte Kopie im gesamten System. Ist mindestens eine weitere Kopie der Cache-Zeile vorhanden, so tragen alle Kopien den Zustand *Shared*. Werden die Daten einer Cache-Zeile modifiziert, so bekommt sie den Zustand *Modified* bzw. *Exclusive Modified*, da in diesem Fall wiederum nur eine einzige Kopie im System erlaubt ist. Mögli-

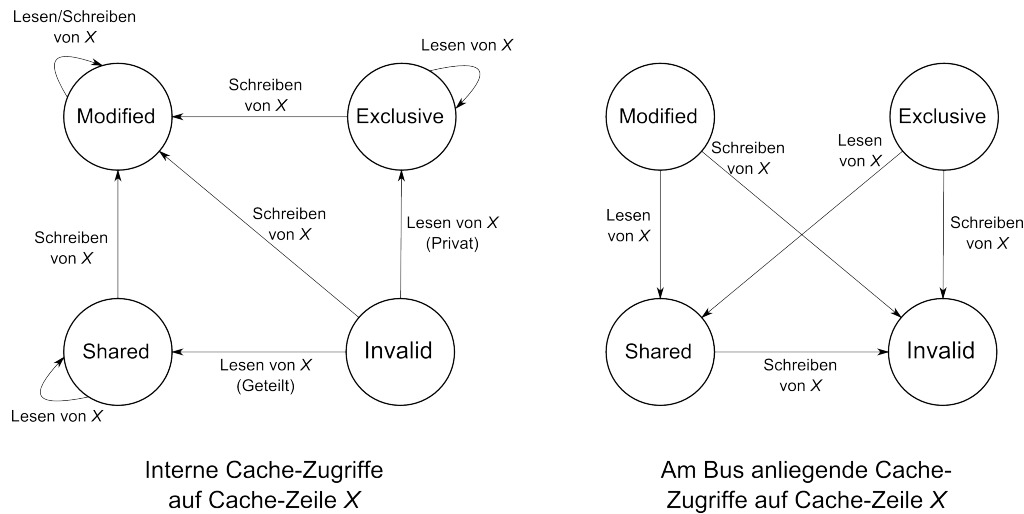


Abbildung 2.11: Zustandsdiagramm einer Cache-Zeile beim MESI-Protokoll für interne (links) und externe (rechts) Ereignisse.

che weitere Kopien müssen dann in den Zustand *Invalid* gesetzt werden und sind nicht mehr zugreifbar.

Diese Zustände bilden die Grundlage des MESI-Protokolls, welches als Cache-Kohärenzprotokoll für Systeme mit einem Bus als Kommunikationsmedium entwickelt wurde. Busbasierte Architekturen sind besonders für die Realisierung eines Hardware-Kohärenzprotokolls geeignet, da es für Rechenkern schnell und effizient möglich ist Informationen auszutauschen. Eine an den Bus gelegte Nachricht kann von allen Rechenkernen gelesen werden (Broadcast). Jeder Rechenkern kann die gesamte Kommunikation im System über den Bus mithören (engl. snooping) und auf, die Cache-Kohärenz betreffende, Zugriffe reagieren. Daher werden solche Techniken *Snooping*-Protokolle genannt.

Die Zustandsänderungen im MESI-Protokoll basieren auf einer Kommunikation zwischen allen Caches im System und werden von Lese- und Schreibzugriffen auf gemeinsame Daten ausgelöst. In Abbildung 2.11 ist das Zustandsdiagramm eines MESI-Protokolls dargestellt. Die Zustandswechsel und gezielte Invalidierung veralteter Cache-Zeilen ermöglichen die Erhaltung kohärenter Zugriffe auf gemeinsame Daten. Kohärenzprotokolle, die Invalidierungen nach Schreibzugriffen einsetzen, werden daher auch *Write-Invalidate* Protokolle genannt. Die MOESI genannte Erweiterung des MESI-Protokolls beinhaltet einen *Owned* Zustand. Der zusätzliche Zustand kennzeichnet eine vom Rechenkern modifizierte Cache-Zeile, die auch von weiteren Rechenkernen geteilt wird. Im Falle eines Cache-Misses übermittelt der Rechenkern die als Owned gekennzeichnete Cache-Zeile direkt an den anfragenden Cache. Dieses Verfahren wird *Bus-Snarfing* genannt. Ein entsprechendes Verfahren wurde erstmals

vom Rudolph et al. [117] vorgestellt. Es folgten zahlreiche Variationen des M(O)ESI Protokolls [69][129].

Anstelle einer Invalidierung von Kopien bei Schreibzugriffen auf geteilte Daten, kann eine direkte Aktualisierung der Daten in den anderen Caches erfolgen. Man spricht in diesem Fall von einem *Write-Update* Protokoll. Implementierungen eines Write-Update Protokolls finden sich im Firefly Protokoll und im Dragon Protokoll [10]. Der Schreibzugriff eines Rechenkerns auf ein Datum im Cache wird an alle im System vorhandenen Kopien weitergeleitet, so dass die Kopien immer konsistent bleiben. Eine Invalidierung ist nicht mehr nötig.

Ein Effekt beim Einsatz von Cache-Speichern ist, dass die Häufigkeit von Speicherzugriffen verringert und damit die Ausführungszeit beschleunigt wird. Jedoch zeigen bereits frühe Untersuchungen von Archibald et al. [10] und Eggers et al. [42], dass Kohärenzoperationen einen signifikanten Overhead an zusätzlich benötigter Ausführungszeit verursachen. Dieser Overhead steigt mit zunehmender Anzahl an beteiligten Rechenkernen. In den folgenden Jahrzehnten hat sich die Forschung daher stark mit der Optimierung der klassischen Kohärenzprotokolle beschäftigt. Stenström et al. [126] führen einige der frühen Ansätze auf.

Betrachtet man aber die in heutigen Architekturen eingesetzten Kohärenzverfahren, so stellt man fest, dass diese Optimierungen von Snooping-Kohärenzprotokollen kaum Anwendung finden. Das liegt darin begründet, dass die Optimierungen in der Regel keine allgemeingültigen Verbesserungen darstellen, sondern eher gezielte Optimierungen hinsichtlich bestimmter Anwendungsbereiche sind.

2.4.2 Verzeichnis-Protokolle

Mit zunehmender Zahl an Rechenkernen in einem Rechenkern wird der Bus als Verbindungsmedium unpraktikabel, da der Wettstreit um den Zugriff auf dieses gemeinsame Medium zum Flaschenhals für die Performanz im System wird. Netzartige Kommunikationstopologien mit Punkt zu Punkt Verbindungen ermöglichen die Kommunikation zwischen mehreren Partnern zur selben Zeit. Snooping-Protokolle wiederum sind in solchen Systemen nicht mehr praktikabel, da ein Broadcast von Kohärenznachrichten an alle Prozessoren nicht mehr so effizient bewerkstelligt werden kann. Es bedarf einer übergeordneten Einheit zum Zusammentragen und Verteilen von Kohärenzinformationen. Erste verzeichnisbasierte Protokolle mit einer zentralen Verzeichnisstruktur wurden Ende der 70er Jahre von Tang [130] und Censier et al. [26] vorgestellt.

Verzeichnisbasierte Kohärenzprotokolle verwalten Informationen über den Status der Kopien gemeinsamer Daten innerhalb einer meist zentral gelegenen Verzeichnisstruktur, auf die jeder Rechenkern Zugriff hat. Für jedes gemein-

same Datum werden die Position und der Zustand aller Kopien des Datums protokolliert. Treten Änderungen im Zustand eines gemeinsamen Datums auf, wird diese Information an das Verzeichnis weitergegeben, welches wiederum die tatsächlich davon betroffenen Rechenkerne informiert. Der Hauptvorteil dieses Verfahrens ist, dass diese Kohärenznachrichten nur noch an die Rechenkerne geschickt werden, die als Empfänger der Nachricht vorgesehen sind. Es fällt somit keine unnötige Last am Kommunikationsmedium an.

Das Führen dieser Kohärenzverzeichnisse benötigt jedoch viel Speicherplatz, der linear mit der Anzahl an Rechenkernen wächst. Zahlreiche Optimierungen zur Verringerung der Verzeichnisgröße wurden vorgestellt und von Agarwal et al. [5] und Chaiken et al. [28] evaluiert. Archibald et al. [9] präsentierten ein Verzeichnisprotokoll, das einen geringen Speicherbedarf aufweist, jedoch auf Broadcast Nachrichten, ähnlich denen in einem Snooping-Protokoll, basiert.

Ungeachtet des Speicherbedarfs kann auch der Zugriff auf ein zentrales Verzeichnis allein zum Flaschenhals für die Performanz werden. Für den DASH Multiprocessor [86] wurde daher ein Protokoll entwickelt, das auf einem über das System verteilten Kohärenzverzeichnis beruht. Bei einem verteilten Verzeichnis hat jedes gemeinsame Datum einen, zu einem Rechenkern bzw. Cluster zugehörigen Heimatknoten. Zugriffe auf dieses Datum werden dann, z.B. mit Hilfe einer Hashing-Funktion, an das Kohärenzverzeichnis im Heimatknoten geleitet. Der Heimatknoten ist für die Erhaltung kohärenter Zugriffe auf dieses Datum zuständig. Er speichert erweiterte Kontextinformationen zum Teil direkt in den Cache-Blöcken (In-Cache Verzeichnis).

Verzeichnisbasierte Kohärenzprotokolle bringen auch Nachteile mit sich. Der hohe Kommunikationsaufwand und die zum Teil langen Wege zwischen den beteiligten Caches sorgen für hohe Latenzen beim Zugriff auf gemeinsame Daten. Mit zunehmender Anzahl von Prozessoren wird dieser Overhead immer gravierender. So wurden auch in der jüngeren Forschung weiterhin Optimierungen entwickelt, um den Speicherbedarf von Kohärenzverzeichnissen zu minimieren [29][150][102] oder die Zugriffszeit zu erhöhen [23][115][84].

Ein Ansatz, den hohen Kommunikationsaufwand von Verzeichnisprotokollen zu umgehen, sind Shared-NUCA Systeme. Hierbei sind die lokalen Cache-Speicher eines Rechenkerns auch von anderen Rechenkernen zugreifbar. Analog zu dem verteilten Speicher des Shared-Memory Modells, bilden die einzelnen Caches einen gemeinsamen Cache für das ganze System. Somit werden redundante Kopien gemeinsamer Daten weitestgehend vermieden und Zugriffe immer zum Heimatknoten eines Datums geleitet. Zahlreich Optimierungen basieren auf einer solchen Speicherarchitektur [151][56][74].

2.4.3 Software Cache-Kohärenz

Ein völlig anderer Ansatz Cache-Kohärenz zu gewährleisten als mit Hilfe komplexer Hardwareerweiterungen, ist die Software Cache-Kohärenz. Hierbei werden spezielle Einschränkungen bzw. Anforderungen an den Gebrauch der Software gestellt, die das Auftreten von inkohärenten Zugriffen verhindern. Eine besonders strikte und naheliegende Einschränkung ist es, Zugriffe auf gemeinsame Daten nur unter Umgehung des Cache-Speichers durchzuführen. Folglich werden ausschließlich private Daten im Cache gespeichert und inkohärente Zugriffe vorbeugend verhindert. Damit geht jedoch eine hohe Auslastung des Verbindungsnetzes einher, da Zugriffe auf gemeinsame Daten immer in Zugriffen auf den gemeinsamen Speicher resultieren, der damit zum Flaschenhals für die Performanz werden kann. Gerade in Systemen mit einer Vielzahl an Rechenkernen ist diese Lösung wenig effizient.

Software Cache-Kohärenzverfahren haben daher das Ziel Inkohärenz zu vermeiden und dabei trotzdem einen möglichst großen Nutzen aus dem Cache zu ziehen. Es gilt einen bestmöglichen Ansatz dafür zu finden, welche Daten zu welchem Zeitpunkt im Cache gehalten werden können und wann diese wieder aus dem Cache entfernt werden müssen. Gemeinsam haben diese Verfahren den Einsatz von Instruktionen, die den Cache gezielt steuern und Invalidierungen auslösen (*Self-Invalidation*). Sie kooperieren mit dem jeweils angewendeten Programmiermodell, um Zeitpunkte zu identifizieren, an denen Operationen zur Erhaltung der Kohärenz notwendig sind.

Bei einer Invalidierung des gesamten Cache-Speichers werden mehr Daten aus dem Cache entfernt werden, als tatsächlich notwendig wäre. Daher wurden Techniken entwickelt, um eine Invalidierung weitestgehend auf die Daten im Cache zu beschränken, die inkohärente Zugriffe möglich machen. Dieser Ansatz wird als *Selective Invalidation* bezeichnet [31][37].

Eine andere Klasse von Software-Kohärenzprotokollen beruht auf Compileranalysen, die den Programmcode auf Datenabhängigkeiten zwischen parallel ausgeführten Codeabschnitten analysieren. Für Speicherzugriffe, die bei der Ausführung potentiell inkohärent werden könnten, wird die Nutzung des Caches eingeschränkt bzw. untersagt. In frühen Ansätzen, eingesetzt im NYU Supercomputer [40] und IBM RP3 [99], werden einzelne Segmente des Programmcodes (*computational units*) auf die Verwendung gemeinsamer Daten hin analysiert. Lediglich Daten, auf die ein Prozessor exklusiven Zugriff im jeweiligen Segment hat, werden in den Cache geladen und nach Abschluss der *computational unit* invalidiert. Veidenbaum [141] sowie Lee et al. [85] entwickelten ähnliche Verfahren, die auf parallel ausgeführte Schleifen ausgelegt sind. Hierbei werden Cache Management Instruktionen zum Programm hinzugefügt, die den Cache für bestimmte Speicherzugriffe deaktivieren und bei Bedarf den gesamten Cache invalidieren.

Smith [124] führte mit seinem *One Time Identifier* Verfahren eine Klasse von Kohärenzprotokollen ein, die mit sogenannten Time Stamps arbeiten. Es handelt sich dabei um Zähler, die für jedes Datum den Zugriff des jeweils letzten Rechenkerns protokollieren. Ein Rechenkern vergleicht den lokalen und globalen Zähler eines Datums bei jedem Zugriff, um zu erkennen, ob es inzwischen von einem anderen Rechenkern verwendet wurde und daher invalidiert werden muss. Darauf basiert eine Vielzahl von Optimierungen [33][92][38].

Im Ansatz von Sandhu et al. [118] werden Zugriffe auf gemeinsame Daten anhand von *Shared Regions* identifiziert. Mit Shared Regions wird eine Synchronisation der Speicherzugriffe nicht über Abschnitte des Programmcodes, sondern über speziell definierte Speicherregionen ausgeführt. Das Konzept stellt damit eine Erweiterung des *Release Consistency* Modells dar.

Neuere Forschungsarbeiten beschäftigen sich mit einem kooperativen Einsatz von Hardware- und Software-Verfahren. Zahlreiche Weiterentwicklungen der compilerbasierten Verfahren haben zum Ziel, mit Hilfe verschiedener Hardwareerweiterungen, zukünftige Invalidierungen vorherzusagen und damit Zugriffslatenzen auf gemeinsame Daten und die Anzahl von Invalidierungsnachrichten zu verringern (vgl. [16][32][82][94][78][96]).

Es wurden Verfahren vorgestellt, die auf dem Einsatz eines speziellen Programmiermodells mit Unterstützung eines Verzeichnisprotokolls beruhen (vgl. [62][127][27][70]). Ashby et al. [12] verwenden eine über Bloom-Filter approximierte und zentral gespeicherte Liste aller Daten, die voraussichtlich von anderen Rechenkernen gelesen werden und daher invalidiert werden müssen. Eine weitere Klasse von Kohärenzprotokollen baut auf die Unterstützung des Betriebssystems bei der Abbildung von Daten im Cache [48].

Ein weiterer Ansatz kohärente Zugriffe auf gemeinsame Daten sicherzustellen ist *Transactional Memory* [60][58]. Das Konzept von Transactional Memory sieht eine atomare spekulative Ausführung festgelegter Codesequenzen vor, in denen auf gemeinsame Daten zugegriffen wird. Dabei ist es nicht nötig, diese Codesequenzen mit einem Lock oder ähnlichem zu schützen. Das Verfahren erkennt die simultane Ausführung einer solchen Codesequenz von mehreren Rechenkernen als Konflikt. Im Falle eines Konflikts wird die Ausführung der Codesequenz abgebrochen und erneut begonnen. Während der Ausführung einer solchen Codesequenz werden Modifikationen ausschließlich lokal in einem speziellen Cache durchgeführt und erst bei erfolgreicher, d.h. konfliktfreier Ausführung an den gemeinsamen Speicher und die anderen Rechenkerne weitergegeben.

2.5 Echtzeitsysteme

Die Vielzahl existierender Computersysteme lässt sich in Klassen unterteilen, die dem jeweiligen Anwendungsbereich entsprechen (z.B. Datenbanksysteme,

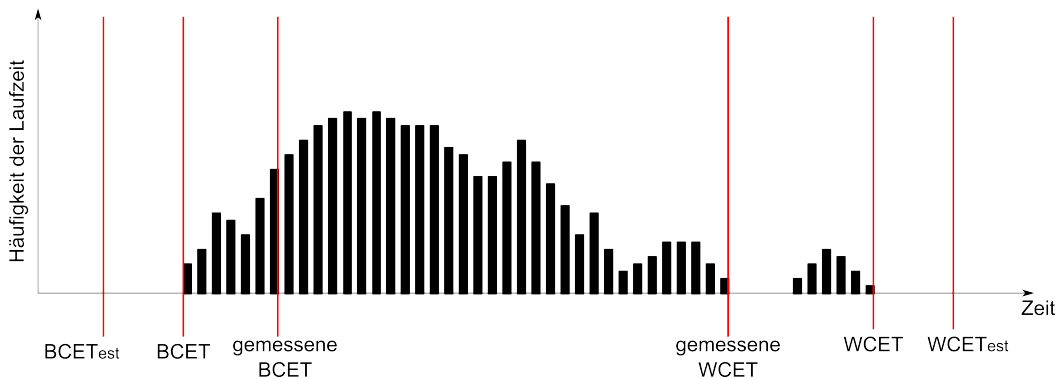


Abbildung 2.12: Veranschaulichung verschiedener Laufzeitschranken bei der Ausführung eines Programms, nach Wilhelm [147].

High-Performance etc.). Die Klasse der Echtzeitsysteme zeichnet sich im Vergleich zu anderen Systemen durch besonders strikte Anforderungen an die Ausführung von Programmen aus. Neben der logischen Korrektheit, ist auch die zeitliche Korrektheit der Ausführung entscheidend. Die Ausführungszeit einer Applikation ist an Zeitschranken gebunden, die eingehalten werden müssen um ein korrektes Systemverhalten zu garantieren. Bei Echtzeitsystemen handelt es sich oft um eingebettete Systeme [91] und sicherheitskritische Systeme. Für sicherheitskritische Systeme ist die Einhaltung der Echtzeitanforderungen besonders wichtig, da ein Materialschaden oder sogar die physische Gesundheit von Menschen von der zeitlich korrekten Ausführung abhängen. Für solche *harten Echtzeitsysteme* ist ein garantiertes Einhalten der Zeitanforderungen zwingend erforderlich. Echtzeitsysteme, bei denen eine Überschreitung der Zeitschranken in einem beschränktem Maße tolerierbar ist, werden *weiche Echtzeitsysteme* genannt. In beiden Fällen ist es notwendig eine obere Grenze für die maximale Ausführungszeit zu ermitteln, um ein mögliches Überschreiten der Zeitschranken erkennen zu können.

2.5.1 WCET

Die *Worst-Case Execution Time* (WCET) einer Applikation ist die höchstmögliche Ausführungszeit, die unter Berücksichtigung sämtlicher Faktoren auftreten kann [147]. Mit der Kenntnis der WCET einer Applikation kann das Einhalten von Laufzeitschranken garantiert werden, sofern die WCET unterhalb der gesetzten Zeitschranke liegt. Jedoch ist die Ermittlung der WCET eines Programms eine hochkomplexe Aufgabe, die einer Vielzahl von Einflüssen unterliegt. Das Zeitverhalten aller beteiligten Hardwarekomponenten muss hinreichend genau bekannt sein. Zum einen muss die Ausführungszeit der Instruktionen im Prozessor unter Berücksichtigung aller Eigenschaften der

Mikroarchitektur (u.a. Pipelinegröße, spekulative Ausführung, Instruktionsscheduling, Timing-Anomalien [114]) bekannt sein [6]. Zum anderen müssen die Zugriffslatenzen der verschiedenen Speichermodule, des Verbindungsmediums sowie weiterer Peripherie berücksichtigt werden. Hinsichtlich der Software muss der längste Ausführungspfad des Programms ermittelt werden, der von den Eingabedaten zu Beginn und im Laufe der Ausführung abhängt.

Den Einfluss all dieser Faktoren auf die Ausführungszeit exakt wiederzugeben ist mit den aktuell zur Verfügung stehenden Methoden der Laufzeitanalyse nicht möglich und auch in naher Zukunft nicht zu erwarten. Stattdessen begnügt man sich damit, mit Hilfe der Laufzeitanalyse eine Approximation der WCET zu ermitteln. Das Resultat einer solchen Analyse bezeichnet man als geschätzte WCET, im folgenden $WCET_{est}$ genannt. Sie entspricht einer oberen Schranke der Laufzeit und ist nicht mit der tatsächlichen WCET zu verwechseln. Analog zur Annäherung der WCET kann mit Hilfe der Laufzeitanalyse auch eine untere Schranke für die bestmögliche Laufzeit (*Best-Case Execution Time* - BCET) ermittelt werden. Abbildung 2.12 zeigt ein Histogramm der möglichen Ausführungszeiten eines Programms und illustriert verschiedene Laufzeitschranken.

Eine WCET-Abschätzung gilt als sicher, wenn sie die tatsächliche WCET nicht unterschreitet. Die Güte oder Präzision einer $WCET_{est}$, ist als die Differenz zur tatsächlichen WCET definiert. Da die WCET nicht bekannt ist, kann auch die Präzision einer Laufzeitanalyse nicht angegeben und somit nicht als Kriterium für eine Laufzeitanalyse verwendet werden. Hilfreich ist dieses Kriterium aber bei der Bewertung verschiedener Instanzen einer Laufzeitanalyse. So können Änderungen an der Analysetechnik oder der Analyseparameter zu einer Verbesserung bzw. Verschlechterung der Präzision führen.

2.5.2 Laufzeitanalyse

Man unterscheidet bei der Laufzeitanalyse zwischen statischer und dynamischer Analyse. Beide Methoden sind bezüglich ihrer Arbeitsweise, als auch der Bewertung ihrer Ergebnisse, grundlegend verschieden.

Die dynamische Laufzeitanalyse, auch Laufzeitmessung genannt, ermittelt die $WCET_{est}$ anhand tatsächlicher oder simulierter Ausführung der Applikation auf dem Zielsystem und durch Messung der resultierenden Ausführungszeit. Dabei wird das Programm sehr häufig mit verschiedenen Eingabedaten und Parametern ausgeführt. Ziel dieses Vorgehens ist es, eine Ausführung zu bewirken, bei der die tatsächliche WCET erreicht wird. Es kann jedoch nicht ermittelt werden, wann und ob dieser Fall jemals eintritt. Daher wird die höchste gemessene Ausführungszeit als $WCET_{est}$ anerkannt. Es muss davon ausgegangen werden, dass die so ermittelte $WCET_{est}$ möglicherweise kleiner ist als die tatsächliche WCET und daher nicht als obere Laufzeitschranke ver-

wendet werden kann. Aus diesem Grund wird die $WCET_{est}$ anschließend mit einem Faktor multipliziert, von dem ausgegangen wird, dass er zu einem Resultat führt, das größer als die WCET ist. Aber auch dies kann schlussendlich nicht verifiziert werden. Somit bleibt das Ergebnis einer dynamischen Analyse immer mit dem Risiko behaftet, dass es die tatsächliche WCET unterschreitet. Dennoch findet diese Methode im kommerziellen Bereich große Anwendung, z.B. wenn eine statische Analyse nicht anwendbar ist.

Die statische Laufzeitanalyse beruht auf einer Analyse des Programmcodes. Hierbei wird der Binärcode des Programms in sogenannte *Basic Blocks* unterteilt, Codeabschnitte, die streng sequentiell abgearbeitet werden. Eine *Pfadanalyse* ermittelt den möglichen Programmfluss durch diese Codeabschnitte. Die Ausführungszeit jedes einzelnen Basic Blocks wird errechnet und in die Pfadanalyse miteinbezogen. Somit kann der längste Ausführungspfad für das Programm ermittelt und anschließend die maximale Ausführungszeit für diesen speziellen Pfad berechnet werden. Der Berechnung der maximalen Ausführungszeit liegt ein detailliertes Zeitmodell der Systemhardware zugrunde. Darin werden obere Schranken für die Ausführungszeit der einzelnen Instruktionen (*Pipelineanalyse*) sowie der Zugriffe auf Speicher und Peripheriekomponenten festgelegt.

Der Vorteil einer statischen Laufzeitanalyse gegenüber einer Laufzeitmessung liegt in der Erfassung des längsten Ausführungspfades. Damit wird sichergestellt, dass das Resultat die tatsächliche WCET nicht unterschreiten kann (sofern das Zeitmodell der Architektur korrekt ist). Die ermittelte $WCET_{est}$ ist immer eine Überabschätzung der WCET und erlaubt somit eine Garantie über die Einhaltung der Zeitschranken. Jedoch kann die Überabschätzung der WCET bei der statischen Analyse sehr hoch ausfallen. Der ermittelte längste Ausführungspfad muss nicht dem Ausführungspfad entsprechen, der zur WCET führt. So kann es sein, dass der längste Ausführungspfad in der Realität nie ausgeführt wird. Außerdem beschreibt das Zeitmodell der Architektur jeweils die höchstmögliche Ausführungszeit für eine Instruktion und einen Speicherzugriff. Dadurch wird ein hoher Grad an Pessimismus in die Ermittlung der $WCET_{est}$ miteinbezogen, der sich im Resultat niederschlägt. Gegebenenfalls überschreitet die $WCET_{est}$ die in den Anforderungen gesetzte Zeitschranke und es müssen Anpassungen am Programmcodes oder der Systemhardware vorgenommen werden. Es steht eine Vielzahl von Tools zur Verfügung, die eine statische Laufzeitanalyse realisieren [147]. Dazu gehören u.a. das *aiT* Tool [1] (AbsInt, Saarbrücken) und die OTAWA Toolbox [14] (IRIT, Toulouse).

Es existiert auch eine Mischform der statischen und dynamischen Laufzeitanalyse, die hybride Laufzeitanalyse. Hierbei wird zuerst der längste Ausführungspfad des Programmcodes bestimmt. Für die einzelnen Programmabschnitte auf diesem Pfad werden mittels Laufzeitmessung obere Laufzeitschranken ermittelt. Die Summe aller Laufzeitschranken, die den längsten Pfad

abdecken, bildet die $WCET_{est}$. Dieses Verfahren kombiniert die Vorteile beider Verfahren. Einerseits wird mittels Pfadanalyse der längste Ausführungspfad berücksichtigt, andererseits wird eine hohe Überabschätzung vermieden. Allerdings erbt die hybride Laufzeitmessung auch die Nachteile beider Verfahren und macht es damit schwierig eine Einschätzung über das Resultat zu treffen. Es ist nun nicht mehr möglich zu sagen, ob das Resultat größer oder kleiner als die tatsächliche WCET ausfällt. Eine hybride Laufzeitanalyse wird von kommerziellen Tools wie RapiTime [112] (Rapita Systems) angeboten. Analysetools setzen weitere Analysemethoden, wie die Intervalanalyse [35] ein. Neben der Vielzahl möglicher Kontrollpfade, erschweren Speicherzugriffe mit erst zur Laufzeit bekannten Zieladressen (effektive Adresse) die Vorhersage eines Cache-Zustands.

Für harte Echtzeitsysteme, die ein Überschreiten der Laufzeitschranken in keinem Fall tolerieren, ist eine statische Laufzeitanalyse die sicherste Wahl. Eine hohe Überabschätzung der WCET wird in Kauf genommen, um eine sichere Garantie über die Einhaltung der Zeitschranken geben zu können. Um die Überschätzung möglichst gering zu halten, wird großen Wert auf eine Vorhersagbarkeit des Programmflusses sowie auf die Wahl der Systemarchitektur gelegt. Die eingesetzte Hardware muss ein möglichst vorhersagbares Zeitverhalten aufweisen können. Dies gilt insbesondere für an Speicherzugriffe beteiligte Komponenten, wie den Cache-Speicher.

2.5.3 Cacheanalyse in der statischen Laufzeitanalyse

Cache-Speicher nehmen bei der statischen Analyse eine besondere Rolle ein [146][119]. Dabei muss zwischen Instruktions- und Datencaches unterschieden werden. Instruktionscaches sind mit den aktuellen Methoden der Laufzeitanalyse sehr gut analysierbar [49]. Bei Datencaches ist die Zieladresse sowie die Latenz eines Zugriffs deutlich schwieriger vorherzusagen. Die Entscheidung, ob ein Speicherzugriff zu einem Cache-Hit oder Cache-Miss führt, hat große Auswirkungen auf die maximale Latenz des Zugriffs, welche wiederum in die $WCET_{est}$ einfließt. Im Falle eines Cache-Hits wird der Zugriff direkt vom Cache beantwortet, und es ist mit einer statischen Latenz von etwa einem Cache-Zyklus zu rechnen. Tritt ein Cache-Miss auf, müssen maximale Wartezeiten für den Zugriff auf gemeinsame Ressourcen, wie das Verbindungsmedium und den gemeinsamen Speicher, miteinbezogen werden. Bei einer Speicherhierarchie mit mehreren Cache-Ebenen können potentiell weitere Zugriffslatenzen auftreten.

Ein Cache-Miss kostet ein Vielfaches an Zugriffszeit im Vergleich zu einem Cache-Hit. Daher dreht sich bei der Analyse des Zeitverhaltens eines Cache-Speichers alles um eine möglichst gute Voraussage von Cache-Hits. Die Vorhersage beruht auf einer Einschätzung über den Zustand des Cache-Speichers vor

einem Speicherzugriff. Der Zustand des Caches zu einem konkreten Zeitpunkt umfasst den Inhalt der Cache-Blöcke, inklusive der Kontextinformationen. Dazu gehört auch der Zustand der Cache-Blöcke in Bezug auf die angewendete Ersetzungsstrategie. Ein umfassendes Wissen über den Zustand des Cache-Speichers an einer konkreten Position im Programmfluss würde eine sichere Vorhersage des nächsten Zugriffs ermöglichen. Jedoch sind zum Zeitpunkt eines Speicherzugriffs in der Regel mehrere Zustände möglich. Mit Hilfe einer Kontrollflussanalyse des Programms können die verschiedenen Pfade ermittelt werden, die zu diesem Punkt im Programmfluss führen. Jeder dieser möglichen Pfade muss für die Vorhersage berücksichtigt werden. Dies führt, unter Berücksichtigung weiterer Varianten, schnell zu einer Zustandsexplosion, das Problem ist nicht mehr effizient zu lösen. Daher wird versucht, mittels Anwendung der abstrakten Interpretation [34], den Zustandsraum zu verringern. Es wird eine *Must-* und *May-Analyse* des aktuellen Cache-Zustands durchgeführt.

Eine *Must-/May-Analyse* ermöglicht, für eine konkrete Ersetzungsstrategie, Aussagen darüber, ob sich ein Datum zu einem bestimmten Zeitpunkt im Cache befindet [50]. Dazu werden alle möglichen Zustände eines Caches zu einem abstrahierten Zustand zusammengefasst. Der abstrahierte Zustand enthält eine obere und untere Schranke für das Alter jedes Datums bzw. des entsprechenden Cache-Blocks. Die *Must-Analyse* ermittelt Cache-Blöcke, die in allen möglichen Zuständen enthalten sind und sich daher mit Sicherheit im Cache befinden. Die *May-Analyse* dagegen ermittelt Cache-Blöcke, die potentiell im Cache zu finden sind.

Abstrakte Interpretation der Cache-Zustände und eine *Must-/May-Analyse* ermöglichen es Cache-Zugriffe auf verschieden Weise zu klassifizieren. Die Klassifikation *Always-Hit* beschreibt einen Cache-Zugriff, der bei jeder Ausführung einen Hit erzeugt. Ein *Always-Miss* resultiert hingegen jedes Mal in einem Cache-Miss. Ist keiner der beiden Zustände möglich, gilt der Zugriff als nicht klassifizierbar (*Not Classified*). In weiterführenden Cacheanalysen werden zusätzliche Klassifikationen (z.B. *First-Miss*) miteinbezogen.

2.5.4 Echtzeitfähigkeit in Mehrkern-Prozessoren

Das Design eines Einkern-Prozessors hat bereits einen entscheidenden Einfluss auf die zeitliche Vorhersagbarkeit [132]. Mehrkern-Prozessoren stellen zusätzliche Anforderungen an eine statische Laufzeitanalyse. Die enge Kopplung der Komponenten und der Zugriff auf gemeinsame Ressourcen verursachen eine gegenseitige Beeinflussung der einzelnen Rechenkerne bei der Ausführung. Zugriffslatenzen wie sie in Einkern-Prozessoren auftreten, können nicht mehr verwendet werden, da der Wettstreit um gemeinsame Ressourcen die Zugriffszeit zusätzlich erhöht. Eine WCET-Analyse für Mehrkern-Prozessoren muss daher unter neuen Rahmenbedingungen stattfinden. Die Entwicklung neuer

Konzepte der Laufzeitanalyse, die eine gegenseitige Beeinträchtigung von Rechenkernen berücksichtigen, ist ein wachsendes Forschungsgebiet [72][30]. Eine echtzeitfähige Arbitrierung der Zugriffe auf gemeinsame Ressourcen ist notwendig, um mögliche Wartezeiten eingrenzen zu können. So wurden z.B. echtzeitfähige Speichercontroller entwickelt [7][79], die maximale Zugriffslatenzen in Abhängigkeit von der Anzahl beteiligter Rechenkerne bieten.

Präzise Schranken für die Zugriffszeit auf einen Cache-Speicher sind deutlich aufwändiger zu ermitteln. Die Latenz hängt davon ab, ob die Anfrage lokal im Cache beantwortet werden kann. Bei einem Cache-Hit wird die Anfrage des Rechenkerns direkt beantwortet, ohne dass auf weitere Ebenen der Speicherhierarchie zugegriffen wird. Die Latenz für einen Cache-Hit ($\Delta_{CacheHit}$) ist (ohne Berücksichtigung von Kohärenzoperationen) konstant und entspricht in der Regel einem Prozessor- bzw. Cache-Zyklus ($\Delta_{CacheCycle}$):

$$\Delta_{CacheHit} = \Delta_{CacheCycle} \quad (2.1)$$

Die Latenz für einen Cache-Miss hingegen ist abhängig von der Speicherarchitektur, da die benötigten Daten von der nächsten Ebene der Speicherhierarchie geladen werden müssen. Der Einsatz mehrerer Cache-Ebenen macht die Latenz eines Cache-Misses weniger vorhersehbar, daher sollte für harte Echtzeitsysteme lediglich eine private Cache-Ebene verwendet werden. Vor dem Laden einer Cache-Zeile kann es außerdem notwendig sein, eine andere Cache-Zeile auszulagern. Diese Ersetzung einer Cache-Zeile erhöht ebenso die Zugriffslatenz für den Cache-Miss.

Einem Zugriff auf den gemeinsamen Speicher geht ein Zugriff auf das Kommunikationsmedium voraus, ebenso eine gemeinsame Ressource. Die Dauer der Übermittlung einer Speicheranfrage über das Kommunikationsmedium ist abhängig von der Anzahl möglicher Konkurrenten. Es muss damit gerechnet werden, dass die Ressource zum Zeitpunkt des Zugriffs bereits belegt ist. Um überhaupt eine Vorhersage über die Zugriffszeit treffen zu können, muss eine echtzeitfähige Arbitrierung auf dem Kommunikationsmedium stattfinden. Die obere Grenze der Zugriffszeit ($\Delta_{CacheMiss}$) für eine Speicheranfrage ist abhängig von jeweiligem Kommunikationsmedium und kann sich aus mehreren einzelnen Latenzen zusammensetzen. Im folgenden werden maximale Zugriffslatenzen für einen Bus, eine Crossbar und netzwerkbasierte Architekturen angegeben.

Busbasierte Systeme

Für einem echtzeitfähigen Systembus bietet sich die Verwendung einer *TDMA* (*Time Division Multiple Access*) Arbitrierung an. TDMA gilt als das Arbitrierungsverfahren, dass die beste zeitliche Vorhersagbarkeit gewährleistet [100][72]. Jedem Teilnehmer wird ein fester Zeitslot für einen exklusiven

Zugriff auf den Bus zugeordnet. Im Round-Robin Verfahren erhält dann jeder Rechenkern regelmäßig die Möglichkeit den Bus zu verwenden. Bei einer Länge des Zeitslots von $\Delta_{TDMA_{Slot}}$ und M Teilnehmern beträgt die obere Schranke für die Wartezeit für einen Buszugriff $(M - 1) * \Delta_{TDMA_{Slot}}$, zuzüglich der eigenen Zugriffszeit auf den Bus. Dies entspricht in etwa der Länge einer kompletten TDMA-Periode. In dieser Arbeit wird für die maximale Latenz für einen Cache-Miss vereinfacht die Länge einer TDMA-Periode Δ_{TDMA} angenommen.

$$\Delta_{CacheMiss_{Bus}} = \Delta_{TDMA} \quad (2.2)$$

Diese obere Grenze für die Zugriffslatenz entspricht der maximalen Wartezeit bei der TDMA Arbitrierung und ist eine sehr pessimistische Annahme. Wenngleich es zahlreiche Ansätze gibt, die Voraussage der Buslatenz zu optimieren [116][8][97][71], wird für die Veranschaulichung der Auswirkungen von Kohärenzoperationen in dieser Arbeit die obige Definition verwendet.

Crossbarbasierte Systeme

Auch in einer Crossbar kann eine TDMA-Arbitrierung verwendet werden, wie z.B. im AFDX Standard [111]. Die Teilnehmer des Systems geben ihre Nachricht in die Crossbar, wo sie bis zur vollständigen Übermittlung gepuffert werden. Die Arbitrierung findet dann separat für jedes Ziel am entsprechenden Ausgangsport statt. Ein Lesezugriff auf den gemeinsamen Speicher entspricht demnach einer ersten Nachricht, und die Übermittlung der Daten an den Rechenkern einer zweiten Nachricht. Folglich muss die Zugriffslatenz beider Teilnehmer in die maximale Latenz eines Cache-Misses einbezogen werden:

$$\Delta_{CacheMiss_{Crossbar}} = 2 * \Delta_{TDMA} \quad (2.3)$$

Netzwerkbasierte Systeme

Die maximale Latenz einer Nachrichtenübertragung zwischen zwei Teilnehmern in einem netzwerkbasierten System hängt von der konkreten Realisierung des Netzwerks ab [121]. Ausschlaggebend ist vor allem die Topologie der Netzwerks (Gitter, Torus, Baum, Ring) sowie das Routing der Nachrichten innerhalb des Netzwerks. Daher wird bei der Bewertung von Kohärenz-Protokollen die Latenz für die Übertragung von Nachrichten in *Hops* gemessen. Ein Hop entspricht einer direkten Nachrichtenübermittlung zwischen zwei Teilnehmern über das Netzwerk. Die maximale Dauer eines Hops, im Sinne einer Worst-Case Analyse, ist die maximale Dauer einer Nachrichtenübertragung im Kommunikationsnetz ($\Delta_{NetworkTraversal}$). In der Regel wird dieser Wert von der

Dauer der Übertragung zwischen den Teilnehmern mit der größten Distanz bestimmt. $\Delta_{NetworkTraversal}$ ist somit ein konstanter Wert für einen Hop und die Zugriffslatenz wird in der Anzahl von Hops gemessen, die für eine vollständige Durchführung des Zugriffs notwendig sind. Die maximale Latenz für einen Cache-Miss entspricht demnach:

$$\Delta_{CacheMiss_{Network}} = 2 * \Delta_{NetworkTraversal} \quad (2.4)$$

Dabei wird im ersten Hop die Anfrage an den Speicher übermittelt und im zweiten Hop das Datum an den Rechenkern transportiert. Obwohl in der Regel bei einem Cache-Miss die einzelnen Datenwörter einer Cache-Zeile in separaten Flits (*Flow Unit*) verschickt werden, fasst man diese der Einfachheit halber zu einem Hop zusammen. Es existieren zahlreiche Implementierungen echtzeitfähiger TDMA-basierter Kommunikationsnetze [125][121].

Kapitel 3

Cache-Kohärenz in Echtzeitsystemen

Cache-Kohärenzprotokolle in aktuellen Mehrkern-Architekturen entsprechen den Anforderungen von Systemen, die auf eine möglichst hohe Performanz hinzielen. Diese Anforderungen decken sich nicht mit den speziellen Anforderungen von Echtzeitsystemen. Kohärenzoperationen initiieren eine Kommunikation zwischen den Rechenkernen und beeinflussen dadurch das Zeitverhalten der beteiligten Systemkomponenten. Wird die zeitliche Vorhersagbarkeit von Datenzugriffen beeinträchtigt, so sind die Voraussetzungen für eine präzise Abschätzung der WCET nicht mehr gegeben.

In diesem Kapitel werden vorerst einige Voraussetzungen für echtzeitfähige Cache-Kohärenz definiert, die notwendig sind, um eine statische Laufzeitanalyse erfolgreich durchzuführen. Es wird ein Blick auf die in aktuellen Mehrkern-Architekturen verwendeten Kohärenzverfahren geworfen. Im Anschluss werden bekannte Cache-Kohärenzprotokolle dahingehend untersucht, ob sie die definierten Voraussetzungen erfüllen. Es wird aufgezeigt, inwieweit die Kohärenzoperationen in diesen Protokollen eine zeitliche Unvorhersehbarkeit verursachen und Zugriffslatenzen beeinträchtigen. Das Kapitel schließt mit einem Überblick über die aktuelle Forschung zu kohärenter Speicherhierarchie für Echtzeitsysteme.

3.1 Voraussetzungen für echtzeitfähige Cache-Kohärenz

Um in harten Echtzeitsystemen eingesetzt werden zu können, muss ein Cache-Kohärenzverfahren ein Maß an zeitlicher Vorhersagbarkeit aufweisen, welches für eine statische Laufzeitanalyse ausreichend ist. Diese Anforderung wird im Folgenden konkretisiert. Dabei wird vorausgesetzt, dass das Kohärenzverfahren in eine Systemarchitektur eingebettet ist, die echtzeitfähig ist. Die übrigen Komponenten des Mehrkern-Prozessors müssen ebenso auch ein zeitlich vorhersagbares Verhalten zeigen.

1. Keine Beeinträchtigung der Cacheanalyse

Die Cacheanalyse ist als Teil der statischen Laufzeitanalyse verantwortlich für die Vorhersage von Cache-Hits bzw. Cache-Misses. Es ist entscheidend für die Cacheanalyse, dass sie ein möglichst umfassendes Bild über den Zustand des Cache-Speichers hat. Ein echtzeitfähiges Kohärenzverfahren sollte die Cacheanalyse nicht negativ beeinträchtigen, indem es unvorhersagbare Zustandsänderungen hervorruft. Wird die Vorhersage von Zugriffslatenzen erschwert oder gar verhindert, so kann dies eine präzise Abschätzung der WCET vereiteln. Für den Fall, dass ein Kohärenzverfahren eigenständig Speicherzugriffe initiiert, sollten diese Zugriffe ebenfalls für die Cacheanalyse vorhersagbar sein.

2. Vermeidung von Deadlocks/Aushungern

Von einem Kohärenzverfahren ausgelöste Kommunikation zwischen den Cache-Speichern sollte nicht die Möglichkeit eines Deadlocks oder Aushungerns eines Rechenkerns bewirken. Durch Aushungern wird ein Rechenkern dauerhaft an der Fortsetzung seiner Ausführung gehindert. Kann so ein Fall bei der Ausführung eines Programms theoretisch auftreten, so muss eine statische Worst-Case Analyse diesen Fall auch in die Ermittlung der $WCET_{est}$ miteinbeziehen. Die Möglichkeit einer dauerhaften Unterbrechung der Ausführung macht es für die statische Analyse unmöglich, eine obere Schranke der Laufzeit zu ermitteln.

3. Vermeidung übermäßig erhöhter Zugriffslatenzen

Wenngleich die Cacheanalyse oft präzise Vorhersagen über Cache-Hits ermöglicht, können unverhältnismäßig hohe Zugriffslatenzen das Resultat einer WCET-Abschätzung unbrauchbar machen. Haben Kohärenzoperationen eine negative Auswirkung auf die Latenz eines Cache-Zugriffs, so sollte das Ausmaß begrenzt und vorhersagbar sein. Eine pauschale Erhöhung der Latenz jedes Cache-Hits bzw. Cache-Misses sollte vermieden werden.

Die zeitliche Vorhersagbarkeit von Speicherzugriffen stellt eine notwendige Bedingung für den Einsatz eines Cache-Kohärenzverfahrens in Echtzeitsystemen dar. Als Richtmaß für die Vorhersagbarkeit sollte stets das zeitliche Verhalten eines herkömmlichen, inkohärenten Caches dienen. Neben der Vorhersagbarkeit ist aber auch die Höhe der Zugriffslatenzen entscheidend. Übermäßig erhöhte Zugriffslatenzen können den Einsatz von Cache-Speichern in Echtzeitsystemen generell in Frage stellen.

3.2 Cache-Kohärenz in aktuellen Mehrkern-Prozessoren

Praktisch alle Hersteller von Prozessorsystemen haben mittlerweile Mehrkern-Prozessoren auf den Markt gebracht. Neben Produkten für den Heim und High-End Bereich mit 2 bis 8 Rechenkernen und höchster Taktfrequenz finden sich ebenso Architekturen für Mikrocontroller und eingebettete Systeme, welche auf Sicherheit und Energieeffizienz ausgerichtet sind. Darüber hinaus richtet sich das Augenmerk zunehmend auf Many-Core Systeme. Solche Architekturen mit dutzenden Rechenkernen auf einem Chip existieren häufig nur als Prototyp, einige Systeme sind jedoch bereits auf dem Markt (z.B. Cavium Octeon III [73]). Sämtliche Mehrkern-Prozessoren verfügen über eine Speicherhierarchie mit mindestens einer Cache-Ebene. Die Problematik der Cache-Kohärenz wird in diesen Systemen sehr unterschiedlich gelöst. Viele Chiphersteller geben nur sehr begrenzt Details über die Handhabung der Cache-Kohärenz preis. Dies mag zum Teil daran liegen, dass die verwendeten Mechanismen den heutigen Anforderungen in Bezug auf Echtzeitfähigkeit nicht immer gerecht werden.

Die Mehrzahl der eingesetzten Kohärenzprotokolle basiert auf dem klassischen MESI-Protokoll. In Mehrkern-Prozessorsystemen mit einer geringen Zahl an Rechenkernen handelt es sich dabei in der Regel um Snooping-Protokolle. So existieren busbasierte Systeme, wie die auf LEON3/4 Prozessoren basierenden Mehrkern-Architekturen von Aeroflex Gaisler [3], die ein Write-Invalidate Protokoll mit Write-Through Schreibstrategie einsetzen. Externe Invalidierungen sind ebenso Teil des Kohärenzprotokolls der QorIQ Architektur des Freescale P4080 [21]. Hierbei wird ein Snooping-Protokoll an einer Crossbar realisiert, das mit einer Write-Back Schreibstrategie arbeitet. Obwohl es sich dabei um ein Write-Invalidate Protokoll handelt, ist mittels der *Intervention* genannten Optimierung eine direkte Aktualisierung einer Cache-Zeile nach einem Cache-Miss möglich, ähnlich dem Bus-Snarfing.

Die Write-Invalidate Technik wird ebenso in verzeichnisbasierten Kohärenzprotokollen verwendet. Der Cavium Octeon [73] setzt ein Protokoll mit zentralem Kohärenzverzeichnis im geteilten L2-Cache seiner Crossbar-Architektur ein. Schreibzugriffe werden auch hier mittels Write-Through durchgeführt. Im

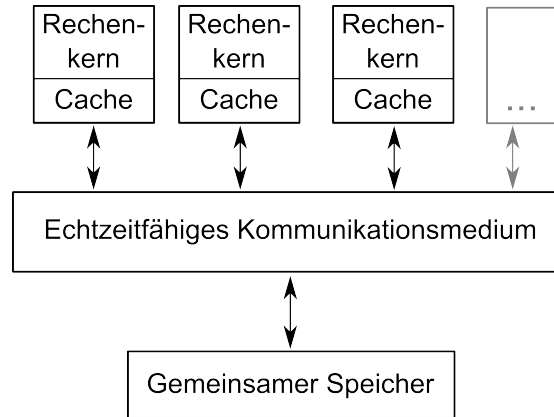


Abbildung 3.1: Aufbau der Basis-Architektur für die Cache-Kohärenz Analyse.

Dynamic Distributed Cache des Tiler TilePro [134] wird hingegen ein verteiltes Verzeichnis-Protokoll realisiert. Zugriffe auf ein gemeinsames Datum, die ebenso mittels Write-Through stattfinden, werden dabei über das Verzeichnis im Heimatknoten des Datum geleitet.

Die in aktuellen Architekturen eingesetzten Cache-Kohärenzverfahren beruhen auf den klassischen Kohärenzlösungen und zielen auf eine hohe Average-Case Performanz ab. Der Fokus dieser Systeme liegt nicht auf Echtzeitfähigkeit. Um dennoch einen Einsatz in Echtzeitsystemen zu ermöglichen, bietet jede dieser Architekturen einen Zugriff auf Daten unter Umgehung des Cache-Speichers an. Alternativ dazu wird mit Hilfe von speziellen Instruktionen, die eine Invalidierung des Caches bewirken, die Anwendung von Software-Kohärenztechniken ermöglicht.

Inzwischen bieten Chiphersteller auch Mehrkern-Architekturen an, die speziell für den Einsatz in sicherheitskritischen Systemen entwickelt wurden. Für solche Architekturen, zu denen der Kalray MPPA [39] sowie der Infineon Aurix [65] gehören, wurde gänzlich auf einen Hardware Cache-Kohärenzmechanismus verzichtet. Diese Systeme machen deutlich, dass für Echtzeitsysteme der Einsatz eines reinen Hardware Kohärenzverfahrens nicht in Frage kommt und lediglich einem überflüssigen Overhead an Logik bzw. Chipfläche entspricht. Stattdessen wird die Verantwortung für die Erhaltung kohärenter Zugriffe dem Benutzer bzw. der Software überlassen.

3.3 Analyse bekannter Cache-Kohärenzprotokolle

In diesem Kapitel werden gängige Ansätze zur Erhaltung der Cache-Kohärenz auf ihre Eignung für harte Echtzeitsysteme hin untersucht. Es wird analysiert, wie Kohärenzoperationen die zeitliche Vorhersagbarkeit eines Systems beeinflussen. Dafür wird zum einen der Einfluss auf die Cacheanalyse untersucht. Zum anderen wird detailliert betrachtet, wie sich Kohärenzoperationen auf die Zugriffslatenzen von Cache-Hits sowie Cache-Missen auswirken. Dabei werden Architekturen mit einem Bus, einer Crossbar, wie auch netzwerkbasierete Verbindungsarten berücksichtigt. Außerdem wird die besondere Rolle von Schreibzugriffen in Cache-Kohärenzverfahren untersucht.

Für diese Analyse wird davon ausgegangen, dass die Kohärenzverfahren in eine Architektur eingebettet sind, die den Anforderungen eines sicherheitskritischen Systems entspricht (vgl. Kapitel 2.5.4). Für einen Mehrkern-Prozessor bedeutet dies, dass die an einem Speicherzugriff beteiligten Systemkomponenten, wie das Kommunikationsmedium und der gemeinsame Speicher, ein vorhersagbares Zeitverhalten aufweisen. Abbildung 3.1 zeigt eine abstrahierte Systemarchitektur, die als Basis-Architektur für diese Analyse dient. Die Speicherhierarchie ist auf eine private Cache-Ebene zwischen Prozessorkern und Kommunikationsmedium beschränkt. Für die Darstellung der Auswirkungen unterschiedlicher Kohärenzoperationen ist ein privater Cache-Speicher pro Prozessorkern ausreichend. Zusätzliche Cache-Ebenen sowie gemeinsame Cache-Speicher erhöhen lediglich die Komplexität der Analyse von Zugriffslatenzen und sind für die generelle Betrachtung der Auswirkungen der Cache-Kohärenz in harten Echtzeitsystemen nicht nutzbringend.

Wie bereits in Kapitel 2.4 erläutert, existiert eine Vielzahl verschiedener Cache-Kohärenzprotokolle und deren Optimierungen. Eine detaillierte Untersuchung aller bekannten Protokolle ist im Rahmen dieser Arbeit nicht durchführbar und auch nicht das Ziel. Allerdings basieren gängige Kohärenzprotokolle auf einer überschaubaren Zahl verschiedener Kohärenzoperationen, die für die Erhaltung Cache-Kohärenz eingesetzt werden. Anhand dieser Kohärenzoperationen lassen sich die existierenden Verfahren klassifizieren und in ihren Auswirkungen verallgemeinern (z.B. Write-Invalidate Protokolle). Es werden daher die existierenden Klassen von Kohärenzprotokollen auf ihren Einfluss auf eine statischen Cacheanalyse hin untersucht.

3.3.1 Einfluss auf die Cacheanalyse

Die Cacheanalyse in aktuellen Laufzeitanalyse-Tools basiert auf dem Paradigma, dass Zustandsänderungen eines Cache-Speichers ausschließlich von dem dazugehörigen Rechenkern verursacht werden und somit isoliert vom übrigen System betrachtet werden können. Die Kohärenzoperationen in gängigen Cache-Kohärenzprotokollen machen diese Prämisse ungültig. Über Kohärenz-

Cache Block	Alter
m_1	2
m_2	0
m_3	3
m_4	T
m_5	1
m_6	T

Abbildung 3.2: Zustands eines Cache-Speichers mit Alterswerten für Cache-Blöcke.

nachrichten, die zwischen den Cache-Speichern eines Mehrkern-Prozessors ausgetauscht werden, übt ein Rechenkern direkten Einfluss auf den Inhalt des privaten Cache-Speichers eines anderen Rechenkerns aus. Diese externen Einflüsse auf einen Cache sind für eine Cacheanalyse, die auf die Betrachtung interner Cache-Zugriffe beschränkt ist, nicht vorhersehbar.

Eine besonders folgenreiche Zustandsänderung eines Cache-Speichers wird durch externe Invalidierungsnachrichten verursacht. Invalidierungsnachrichten zählen zu den Grundoperationen von Write-Invalidate Kohärenzprotokollen. Aber auch in anderen, auf dem MESI-Konzept basierenden Protokollen, werden sie eingesetzt. Sie finden sowohl in Snooping-Protokollen als auch in Verzeichnis-Protokollen Anwendung. Im Zuge einer Schreiboperation eines Rechenkerns auf ein geteiltes Datum werden alle in anderen Caches vorhandenen Kopien dieses Datums für ungültig erklärt, da sie nunmehr einen veralteten Wert tragen. Der Rechenkern versendet dazu eine Invalidierungsnachricht mit der entsprechenden Anweisung an die anderen Cache-Speicher. In einem Bus-system sind dies keine eigenständigen Invalidierungsnachrichten, sondern lediglich die Schreibzugriffe, die an den Bus gelegt werden. Aus Sicht der anderen Rechenkerne kann solch ein Schreibzugriff aber durchaus als Invalidierungsnachricht interpretiert werden. Die Invalidierungsnachrichten wirken sich direkt auf den Zustand der anderen Cache-Speicher aus, und damit auch auf deren Cacheanalyse. Da sie aus der Sicht des betroffenen Caches von einer externen Quelle ausgelöst werden, sind der Zeitpunkt der Invalidierung sowie der betroffene Cache-Block für die Cacheanalyse nicht vorherzusehen. Dies hat enorme Auswirkungen auf die Must-/May-Analyse eines Cache-Speichers.

Im Rahmen einer Must- und May-Analyse wird eine Abstraktion aller zu einem Zeitpunkt möglichen Cache-Zustände C vorgenommen. Diese Zustandsabstraktion ermöglicht es bei einem Cache-Zugriff, Aussagen über das Vorhandensein des gesuchten Datums zu treffen [59]. Ein konkreter Cache-Zustand c umfasst die Alterswerte aller möglichen Cache-Blöcke, die im Rahmen der

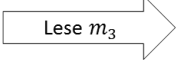
Cache Block	Alter		Cache Block	Alter
m_1	2		m_1	3
m_2	0		m_2	1
m_3	3		m_3	0
m_4	T		m_4	T
m_5	1		m_5	2
m_6	T		m_6	T

Abbildung 3.3: Zustandswechsel eines Cache-Speichers nach Zugriff auf einen Cache-Block unter Anwendung der LRU Ersetzungsstrategie.

verwendeten Ersetzungsstrategie verwaltet werden. Abbildung 3.2 zeigt einen Cache-Zustand mit Alterswerten für die Cache-Blöcke m_1 bis m_6 .

Wird ein Datum nach einem Cache-Miss neu in den Cache geladen, so erhält der entsprechende Cache-Block einen Wert, der das Alter des Blocks im Verhältnis zu den anderen Blöcken im selben Cache Satz angibt. Dieses Alter liegt im Intervall $\{0, \dots, A-1, T\}$, wobei $A-1$ dem maximal möglichen Alter eines gültigen Cache-Blocks entspricht und T einen Block kennzeichnet der sich nicht (mehr) im Cache befindet. Der Wert A ist identisch mit der Assoziativität des Cache-Speichers bzw. der maximalen Anzahl an Cache-Blöcken in einem Cache-Satz.

Im Folgenden wird die Must-/May-Analyse am Beispiel eines Caches mit Least-Recently-Used (LRU) Ersetzungsstrategie erläutert. Least-Recently-Used erzielt von allen Ersetzungsstrategien die bestmögliche Vorhersagbarkeit und sollte daher für ein echtzeitfähiges Cache-Kohärenzverfahren eingesetzt werden [113]. Wird beim Einsatz von LRU auf einen Cache-Block zugegriffen, so bekommt der Block das Alter 0 und ist damit der „jüngste“ Cache-Block, während das Alter der übrigen Blöcke entsprechend angepasst. Abbildung 3.3 zeigt ein Beispiel für einen Zustandswechsel beim Zugriff auf einen Cache-Block. Erreicht ein Cache-Block das maximale Alter $A-1$, so wird er zum Ziel der nächsten Ersetzung und erhält anschließend das Alter T . So ändert sich das Alter der Cache-Blöcke fortlaufend mit jedem Cache-Zugriff.

Must- und May-Analyse

Die Must-/May-Analyse ist eine Kombination zweier eigenständiger Analysen der möglichen Cache-Zustände [132]. Die Must-Analyse errechnet zu einem bestimmten Zeitpunkt einen abstrahierten Cache-Zustand C^u , der obere Schranken für das Alter aller möglichen Cache-Blöcke enthält. Das Alter eines Cache-Blocks in C^u entspricht dem Maximum aller Alterswerte des Blocks, in den

zu diesem Zeitpunkt möglichen Cache-Zuständen. Dabei gilt, dass $A - 1 < T$ ist, d.h. wenn ein Cache-Block in mindestens einem der möglichen Zustände nicht im Cache enthalten ist (besitzt ein Alter von T), so gilt er auch im abstrahierten Zustand C^u als nicht enthalten.

Das Resultat einer Must-Analyse ermöglicht die Vorhersage von Cache-Hits. Für einen Cache-Zugriff wird das Alter des gesuchten Cache-Blocks im abstrahierten Zustand C^u ermittelt. Ist das Alter ungleich T , so ist der Cache-Block in jedem der möglichen Zustände im Cache gespeichert und es kann mit Sicherheit ein Cache-Hit vorhergesagt werden.

Analog zur Must-Analyse errechnet die May-Analyse einen abstrahierten Cache-Zustand C^l , der untere Schranken für das Alter aller Cache-Blöcke enthält. Das Alter eines Cache-Blocks in C^l entspricht demnach dem Minimum aller Alterswerte des Blocks in allen möglichen Cache-Zuständen. Ist der Zustand eines Cache-Blocks in C^l gleich T , so ist der Cache-Block in keinem der möglichen Zustände im Cache enthalten. Das Resultat einer May-Analyse ermöglicht somit die sichere Vorhersage von Cache-Misses. Kombiniert man das Resultat der Must- und May-Analyse, so erhält man ein Intervall der möglichen Alterswerte für jeden Cache-Block.

Geht man davon aus, dass der Cache-Speicher zu Beginn einer Ausführung bzw. einer Cacheanalyse leer ist (*Cold-Cache*), so ist der Alterswert jedes Cache-Blocks T . Es besteht ein exaktes Wissen über den Cache-Zustand. Befindet sich das System jedoch im laufenden Betrieb, so ist in der Regel kein Wissen über den Zustand des Caches zu Beginn der Analyse vorhanden. Das Altersintervall jedes Cache-Blocks ist dann $\{0, T\}$. Im Laufe der Ausführung wird dieses Intervall enger eingegrenzt. Zugriffe des Rechenkerns auf den Cache resultieren in Änderungen der Alterswerte in den abstrahierten Zuständen C^u und C^l . Für einen Cache-Zugriff kann mit Hilfe des Altersintervalls unter Umständen ein garantierter Cache-Hit vorhergesagt werden. Die statische Analyse kann dann als maximale Latenz für den Cache-Zugriff die maximale Latenz eines Cache-Hits annehmen. Ist ein Cache-Hit nicht sicher vorhersagbar, so muss die Worst-Case Analyse von einem Cache-Miss ausgehen und entsprechend die maximale Cache-Miss Latenz für diesen Zugriff anrechnen.

Zustandsänderung bei externer Invalidierung

Wird eine Invalidierung eines Cache-Blocks durchgeführt, unabhängig davon ob sie intern oder extern ausgelöst wird, so hat dies einen direkten Einfluss auf den Zustand des Cache-Speichers. Die Invalidierung erzeugt einen zusätzlichen Zustandswechsel, in dem das Alter des invalidierten Cache-Blocks auf T gesetzt und das Alter der übrigen Cache-Blöcke angepasst wird. Abbildung 3.4 zeigt einen solchen Zustandswechsel. Wird die Invalidierung jedoch von einer externen Quelle ausgelöst, wie es z.B. bei Write-Invalidate Protokollen der

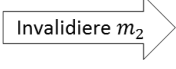
Cache Block	Alter		Cache Block	Alter
m_1	3		m_1	2
m_2	1		m_2	T
m_3	0		m_3	0
m_4	T		m_4	T
m_5	2		m_5	1
m_6	T		m_6	T

Abbildung 3.4: Zustandswechsel eines Cache-Speichers nach Invalidierung eines Cache-Blocks.

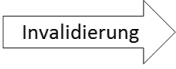
Cache Block	Alter		Cache Block	Alter
m_1	3		m_1	T
m_2	1		m_2	T
m_3	0		m_3	T
m_4	T		m_4	T
m_5	2		m_5	T
m_6	T		m_6	T

Abbildung 3.5: Änderung des abstrahierten Cache-Zustands C^u nach der Invalidierung eines unbekanntem Cache-Blocks.

Fall ist, so kann eine Cacheanalyse nicht vorhersehen, welcher Cache-Block von der Invalidierung betroffen ist. Die Adresse der zu invalidierenden Daten hängt vom jeweiligen Schreibzugriff ab, der die Invalidierung auslöst. Dieser Schreibzugriff wird von einem anderen Rechenkern durchgeführt und liegt nicht im Fokus der Cacheanalyse, die davon betroffen ist. Im Sinne einer Worst-Case Analyse, muss daher jeder möglicherweise im Cache befindliche Block in Betracht gezogen werden. Das heißt, die Menge I aller potentiellen Ziele einer Invalidierung ist definiert durch die Menge aller Cache-Blöcke m aus C^l für die gilt, das Alter von m ist ungleich T .

Die potentielle Invalidierung eines jeden der Cache-Blöcke aus I resultiert in einer Vielzahl neuer möglicher Zustände, die für die Errechnung eines Folgezustands C' abstrahiert werden müssen. Hinzugekommen sind Variationen aller bisherigen Zustände, in denen jeweils ein Cache-Block aus I das Alter T aufweist. Es ergibt sich dadurch ein spezieller Zustandswechsel für C^u bzw. C^l nach $C^{u'}$ bzw. $C^{l'}$.

Im abstrahierten Folgezustand $C^{u'}$, der obere Schranken für das Alter jedes

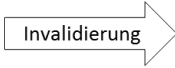
Cache Block	Alter		Cache Block	Alter
m_1	3		m_1	2
m_2	1		m_2	0
m_3	0		m_3	0
m_4	T		m_4	T
m_5	2		m_5	1
m_6	T		m_6	T

Abbildung 3.6: Änderung des abstrahierten Cache-Zustands C^l nach der Invalidierung eines unbekanntem Cache-Blocks.

Cache-Blocks festhält, muss aufgrund der externen Invalidierung das Alter aller Cache-Blöcke auf T gesetzt werden. Abbildung 3.5 zeigt einen solchen Zustandswechsel. Für die Cacheanalyse bedeutet das, dass für jeden Cache-Block angenommen werden muss, dass er nicht im Cache enthalten ist.

Der abstrahierte Folgezustand $C^{l'}$ enthält untere Schranken für das Alter der Cache-Blöcke, so dass hier kein Wert erhöht werden muss, da für jeden Cache-Block ein möglicher Zustand existiert, in dem er nicht invalidiert wurde. Dennoch sind Anpassungen notwendig. Für jeden Cache-Block, der im Ausgangszustand C^l ein Alter größer 0 und ungleich T aufweist, existiert ein möglicher Zustandswechsel, in dem ein „jüngerer“ Cache-Block invalidiert wurde. Bei diesem Zustandswechsel wurden entsprechend die Alterswerte der anderen Cache-Blöcke angepasst, d.h. um einen Wert reduziert. Für den abstrahierten Zustand $C^{l'}$ hat dies zur Folge, dass alle Alterswerte ungleich T um einen Wert reduziert werden, wobei Werte von 0 belassen werden. Ein Beispiel für einen solchen Zustandswechsel ist in Abbildung 3.6 dargestellt.

Diese, durch externe Invalidierungen ausgelösten, Zustandswechsel haben enorme Auswirkungen auf das Resultat der Must- und May-Analyse. Das Altersintervall jedes Cache-Blocks wird erweitert, was einer Vergrößerung des Zustandsraums und damit einer Steigerung der Ungewissheit über den Zustand des Caches gleichkommt. Das neue Altersintervall eines jeden Cache-Blocks enthält nun die obere Grenze T . Für den folgenden Zugriff ist es nun nicht mehr möglich, einen sicheren Cache-Hit vorherzusagen, gleich auf welche Adresse zugegriffen wird.

Ebenso wenig wie das Ziel einer Invalidierung, kann auch der Zeitpunkt einer Invalidierung von der Cacheanalyse vorhergesagt werden. In Anbetracht einer Worst-Case Analyse muss zu jedem theoretisch möglichen Zeitpunkt das Eintreffen einer Invalidierungsnachricht angenommen und in die Analyse einbezogen werden. Mögliche Zeitpunkte von Invalidierungen hängen davon ab, wie die Übertragung von Kohärenznachrichten im Kommunikationsmedium

realisiert ist. Ist es generell möglich, dass zwischen zwei aufeinanderfolgenden Cache-Zugriffen eines Rechenkerns eine externe Invalidierung auftritt, so kann für keinen Zugriff ein Cache-Hit vorhergesagt werden. Nach dem Laden eines Datums in den Cache muss davon ausgegangen werden, dass der entsprechende Cache-Block vor dem nächsten Zugriff durch eine Invalidierungsnachricht wieder aus dem Cache entfernt wird.

In einem echtzeitfähigen System kann das Auftreten möglicher Invalidierungen durch den Einsatz einer echtzeitfähigen Arbitrierung eingeschränkt werden. Da Invalidierungsnachrichten ebenso wie Speicherzugriffe über das Kommunikationsmedium übermittelt werden, unterliegen sie auch dem Zeitschema der Arbitrierung. So ist z.B. für einen Bus mit TDMA-Arbitrierung die Häufigkeit des Auftretens von externen Invalidierungen beschränkt. Unter der Voraussetzung, dass pro TDMA-Slot nur eine Invalidierungsnachricht versendet wird, können in einem Mehrkern-Prozessor mit N Rechenkernen, zwischen zwei Cache-Misses eines Rechenkerns maximal $N-1$ externe Invalidierungen auftreten. Diese Einschränkung verbessert allerdings nicht die Vorhersagbarkeit von Cache-Hits, da nach einem Caches Miss bereits eine mögliche Invalidierung ausreicht, um den zuvor geladenen Cache-Block wieder zu entfernen.

Ziel bei einer kombinierten Must- und May-Analyse ist es, im Laufe einer Ausführung das abstrahierte Altersintervall eines Cache-Blocks einzugrenzen und somit sichere Vorhersagen über Cache-Hits bzw. Cache-Misses treffen zu können. Externe Invalidierungen wirken diesem Prozess des Wissensgewinns entgegen, indem sie die Altersintervalle wieder ausweiten. Durch die Zustandswechsel, die externe Invalidierungen auslösen, tendieren die Intervalle zurück zu ihrem Anfangszustand $\{0, T\}$. Allein die Tatsache, dass ein einzige Invalidierung genügt, um die sichere Vorhersagbarkeit eines Cache-Hits für die folgenden Zugriffe zu unterbinden, macht eine statische Cacheanalyse mit Berücksichtigung von externen Invalidierungen unbrauchbar.

3.3.2 Einfluss auf die Latenz eine Cache-Zugriffs

Neben Invalidierungsnachrichten gibt es eine Reihe weiterer Kohärenznachrichten, die zum Austausch von Informationen zwischen den Rechenkernen versendet werden. Diese Kohärenztransaktionen wirken sich in vielfältiger Weise auf die Latenz von Cache-Zugriffen aus. Im Folgenden wird der Einfluss auf verschiedene Arten von Cache-Zugriffen untersucht.

Kohärenztransaktionen stellen, aus Sicht des Empfängers, externe Zugriffe auf den privaten Cache-Speicher dar. Diese Transaktionen können je nach angewendetem Kohärenzverfahren unterschiedliche Auswirkungen haben. Die im vorherigen Abschnitt erläuterten Invalidierungen sind ein Beispiel für mögliche Kohärenztransaktionen. Weiterhin gibt es Transaktionen, die den Kohärenzzustand eines bestimmten Cache-Blocks ändern, oder geteilte Daten in einer

Cache-Zeile aktualisieren. Außerdem können Invalidierungen einen Cache dazu veranlassen, eine modifizierte Cache-Zeile in den Speicher zurück zuschreiben.

Kohärenztransaktionen bewirken, dass der Inhalt eines Cache-Speichers nicht mehr nur vom zugehörigen Rechenkern, sondern ebenso von anderen Teilnehmern modifiziert werden kann. Das macht den Cache aus Sicht einer Cacheanalyse zu einer geteilten Ressource, um die man im Wettstreit mit anderen Teilnehmern steht. So kann der Zugriff eines Rechenkerns auf seinen Cache mit einem externen Zugriff auf diesen Cache kollidieren. Entscheidend für eine Cacheanalyse ist es daher, in wieweit ein interner Cache-Zugriff von einer externen Kohärenztransaktion beeinflusst werden kann.

Um eine gegenseitige Beeinflussung von internen und externen Zugriffen zu vermeiden, wäre die Verwendung eines Dual-Ported Caches denkbar [137]. Dadurch wäre es generell möglich interne und externe Zugriffe parallel auszuführen. Mit zwei verfügbaren Zugriffssports könnten interne und externe Zugriffe auf verschiedene Cache-Blöcke gleichzeitig oder überlappend stattfinden. Für den Fall, dass die internen und externen Zugriffe zeitgleich auf den selben Cache-Block hinzielen, befinden sich die Zugriffe jedoch wieder im Wettstreit. Bei Lesezugriffen wäre dies noch unproblematisch, bei Schreibzugriffen aber nicht mehr und es müssten Maßnahmen für eine Arbitrierung der Zugriffe getroffen werden. Wie bereits erläutert, ist für eine Cacheanalyse das Ziel und der Inhalt von externen Kohärenztransaktionen nicht bekannt. Im Sinne einer Worst-Case Analyse muss davon ausgegangen werden, dass ein Cache-Block beim Eintreffen eines internen Zugriffs bereits von einer externen Quelle verwendet wird. Es ist daher fraglich, ob eine Arbitrierung in einem Dual-Ported Cache für die Cacheanalyse hilfreich ist. In dieser Arbeit soll aber nicht weiter auf die Folgen einer Arbitrierung einem Dual-Ported Cache eingegangen werden. Stattdessen bezieht sich die Analyse auf die üblicherweise eingesetzten Single-Ported Caches.

Die Möglichkeit des Eintreffens einer Kohärenztransaktion hat zur Folge, dass für die Latenz eines internen Cache-Zugriffs ($\Delta_{CacheAccess}$) eine zusätzliche Wartezeit ($\Delta_{CoherenceTransaktion}$) miteinbezogen werden muss, in der der externe Zugriff behandelt wird.:

$$\Delta_{CacheAccess} = \Delta_{CoherenceTransaktion} + (\Delta_{CacheHit}/\Delta_{CacheMiss}) \quad (3.1)$$

Dies ist besonders für die Latenz eines Cache-Hits bedeutsam, da diese bei einer statischen Laufzeitanalyse eine fixe Größe ist, die keine Überabschätzung der WCET verursacht. Das Ausmaß von $\Delta_{CoherenceTransaktion}$ ist abhängig vom Typ der Kohärenztransaktion. Möglich ist eine zusätzliche Latenz von einem Cache-Zyklus bei Änderungen des Kohärenzzustands eines Cache-Blocks, z.B.

für den Wechsel einer Cache-Zeile in den *Shared* Zustand. Aber auch eine Vielzahl von Cache-Zyklen, nötig für das Zurückschreiben einer Cache-Zeile in den Hauptspeicher, ist denkbar. Eine pauschale Erhöhung der Latenz eines Cache-Hits um mehrere Zyklen sollte nicht vernachlässigt werden. Zwar ist die Latenz eines Cache-Misses um ein Vielfaches höher, dafür treten dieser sehr viel seltener auf. Legt man die Daten der Evaluation von Echtzeitsystemen von Ferdinand und Wilhelm [50] zugrunde, so werden bis zu 91% aller Cache-Zugriffe als Cache-Hit (Always Hit) klassifiziert. Cache-Hit Latenzen haben also einen signifikanten Anteil an der gesamten Laufzeit. Es ist daher davon auszugehen, dass sich eine pauschale Erhöhung der Cache-Hit Latenz spürbar auf die WCET-Abschätzung auswirkt.

3.3.3 Cache-Miss Latenz in Snooping-Protokollen

In Folge eines Cache-Misses wird das gesuchte Datum aus der nächsten Hierarchieebene in den Cache-Speicher geladen. Ein Kohärenzprotokoll hat Auswirkungen darauf, auf welchem Weg die angeforderten Daten in die Cache-Zeile gelangen. Ohne Einwirkung eines Kohärenzprotokolls, erfolgt lediglich ein Zugriff auf den gemeinsamen Speicher. Wird dagegen ein Cache-Kohärenzprotokoll eingesetzt, kann der Ablauf eines Cache-Misses davon beeinflusst werden, ob ein Lese- oder Schreibzugriff durchgeführt wird. Der Einfluss von Kohärenzprotokollen auf Schreibzugriffe soll in einem späteren Teil erläutert werden. Hier beschränkt sich die Analyse auf Cache-Misses bei Lesezugriffen, wobei zwischen Systemen mit Bus, Crossbar und netzwerkbasiertem Kommunikationsmedium unterschieden wird.

Zugriffslatenz in Bus-Systemen

Arbeitet ein Cache mit einer Write-Back Schreibstrategie, so werden Änderungen an Daten ausschließlich im Cache getätigt und nur bei Bedarf an den gemeinsamen Speicher weitergegeben. Solch ein Bedarf entsteht zum Beispiel, wenn ein anderer Rechenkern nach einem Cache-Miss dieses Datum laden möchte. Kohärenzprotokolle weisen den Cache, der die modifizierte Cache-Zeile gespeichert hat, im Folgenden *bereitstellender Cache* genannt, dazu an diese zurück in den gemeinsamen Speicher zu schreiben. Der Zugriff des *anfordernden Caches* wird zuvor abgebrochen und nach Vollendung des Zurückschreibens erneut ausgeführt. Die Latenz des Cache-Misses hängt also davon ab, auf welchem Weg das Zurückschreiben durchgeführt wird.

Im einem Bus-System mit TDMA-Arbitrierung muss das Zurückschreiben in einem der verfügbaren TDMA-Slots durchgeführt werden. Verschiedene Varianten sind denkbar:

1. **Zurückschreiben im TDMA-Slot des bereitstellenden Caches**

Nachdem der Zugriff des anfordernden Caches abgebrochen wurde, kann das Zurückschreiben der modifizierten Cache-Zeile in den gemeinsamen Speicher im nachfolgenden TDMA-Slot des bereitstellenden Caches stattfinden. Daraufhin kann der anfordernde Cache seinen Lesezugriff im nächsten eigenen TDMA-Slot erneut beginnen und die Daten aus dem gemeinsamen Speicher laden. Die maximale Wartezeit für einen Cache-Miss erhöht sich dabei, im Vergleich zu einem herkömmlichen Cache-Miss, um eine zusätzliche TDMA-Periode:

$$\Delta_{CacheMiss_{Bus}} = 2 * \Delta_{TDMA} \quad (3.2)$$

Jedoch ergeben sich bei dieser Varianten mehrere Konfliktmöglichkeiten. Nutzt der bereitstellende Cache seinen TDMA-Slot zum Zurückschreiben der modifizierten Cache-Zeile, so steht dieser Slot nicht mehr für eigene Zugriffe zur Verfügung. Der Konflikt könnte durch eine Priorisierung eines der beiden Zugriffe gelöst werden. Dadurch wäre es aber prinzipiell möglich, dass der andere Zugriff beliebig lang verzögert wird, was eine unbegrenzte maximale Latenz und damit eine undurchführbare Cacheanalyse zur Folge hätte. Eine faire, wechselnde Arbitrierung der konkurrierenden Zugriffe würde eine zeitliche Vorhersagbarkeit ermöglichen. Das Zurückschreiben würde so maximal um eine zusätzliche TDMA-Periode verzögert werden. Die maximale Wartezeit für einen Cache-Miss erhöht sich dabei wiederum um einen TDMA Durchlauf:

$$\Delta_{CacheMiss_{Bus}} = 3 * \Delta_{TDMA} \quad (3.3)$$

Diese Variante hat zum einen den Nachteil, dass der TDMA-Slot des anfordernden Caches lange Zeit ungenutzt bleibt. Zum anderen bleibt ein weiterer möglicher Konflikt bestehen. So ist es denkbar, dass nach dem Zurückschreiben der modifizierten Daten und vor dem erneuten Lesezugriff des anfordernden Caches, ein dritter Cache einen Cache-Miss auf die gleiche Cache-Zeile ausführt und schreibend auf die Daten zugreift. Dadurch wird der Lesezugriff des anfordernden Caches wiederum verhindert, da wieder eine modifizierte Kopie der Cache-Zeile in einem anderen Cache existiert. Die eben beschriebene Prozedur des Zurückschreibens muss nun erneut, diesmal mit dem erwähnten dritten Cache durchgeführt werden. Abbildung 3.7 veranschaulicht eine solche Situation. Der anfordernde Cache kann aber nicht beliebig oft durch eine solche Situation unterbrochen

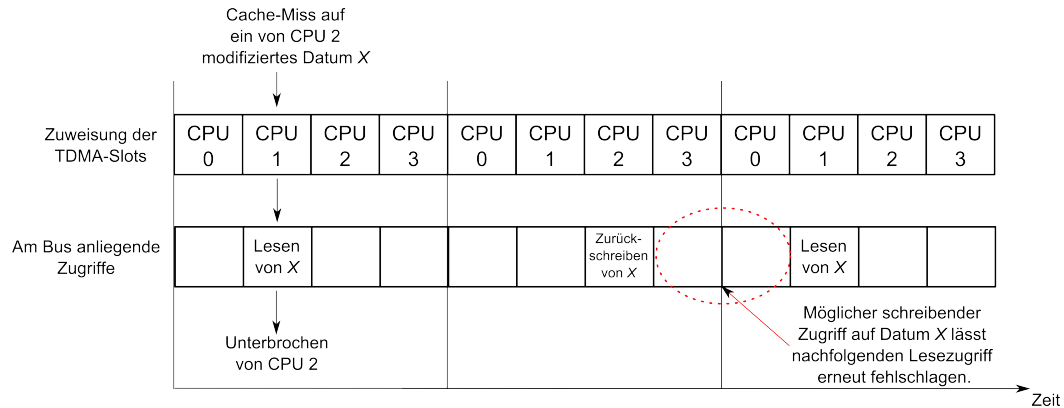


Abbildung 3.7: Veranschaulichung der Möglichkeit der Blockierung eines Rechenkerns bei Anwendung eines Snooping-Protokolls im TDMA-Bus, nach [137].

werden. Eine Unterbrechung ist jeweils nur einmal von jedem Rechenkern möglich, der dem Zugriff des anfordernden Caches mit seinem Zugriff zuvor kommen kann. Die Anzahl solcher Unterbrechungen ist demnach auf $N - 1$ bei N im System vorhandenen Rechenkernen begrenzt. Die maximale Latenz eines Cache-Misses für den anfordernden Rechenkern ist in diesem Fall:

$$\Delta_{CacheMiss_{Bus}} = ((N - 1) * 2) + 1) * \Delta_{TDMA} \quad (3.4)$$

2. Zurückschreiben im TDMA-Slot des anfordernden Caches

Um die zeitliche Vorhersagbarkeit der internen Zugriffe anderer Rechenkerns in ihren TDMA-Slots nicht zu beeinträchtigen, kann einem bereitstellenden Cache ermöglicht werden, einen fremden TDMA-Slot zum Zurückschreiben zu gebrauchen. Es wird dann der eigene Slot des anfordernden Caches verwendet. Nach der Unterbrechung des Lesezugriffs des anfordernden Caches, nutzt der bereitstellende Cache den nächsten TDMA-Slot des anfordernden Caches, um die modifizierte Daten zurückzuschreiben. Der anfordernde Cache kann seinen Lesezugriff dann im darauffolgenden eigenen TDMA-Slot durchführen. Es müssen also zwei zusätzliche TDMA Durchläufe in die maximale Latenz einbezogen werden.

$$\Delta_{CacheMiss_{Bus}} = 3 * \Delta_{TDMA} \quad (3.5)$$

Jedoch kann es bei dieser Variante zu einer Situation des Aushungerns für den anfordernden Rechenkern kommen. Greift zwischen dem Zurück-

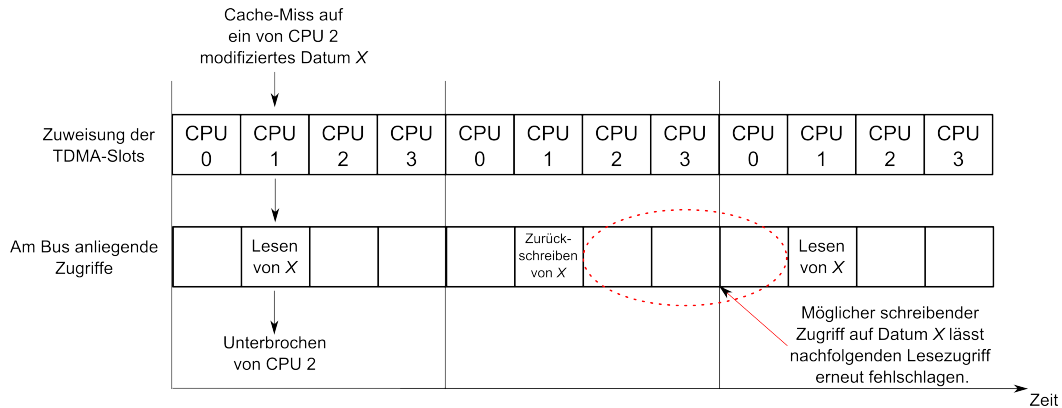


Abbildung 3.8: Veranschaulichung der Möglichkeit des Aushungerns eines Rechenkerns bei Anwendung eines Snooping-Protokolls im TDMA-Bus, nach [137].

schreiben der Daten und dem erneuten Lesezugriff, ein anderer Rechenkern schreibend auf den gleichen Cache-Block zu, so ist (genau wie im vorherigen Beispiel) ein erneutes Zurückschreiben der modifizierten Cache-Zeile notwendig. Doch im Gegensatz zum vorherigen Beispiel, ist es hier jedem Rechenkern möglich, den erneuten Zugriff des anfordernden Rechenkerns zu unterbrechen. Und da zwischen dem Zurückschreiben und dem erneuten Lesezugriff stets eine volle TDMA-Periode liegt, kann die Unterbrechung beliebig oft erfolgen (siehe Abbildung 3.8). Es ist nicht mehr möglich eine maximale Anzahl an Unterbrechungen anzugeben:

$$\Delta_{CacheMiss_{Bus}} = \infty \quad (3.6)$$

Die Möglichkeit des Aushungerns eines Rechenkerns macht eine die Ermittlung einer WCET-Abschätzung unmöglich (siehe Kapitel 3.1).

3. Zurückschreiben mit Update im TDMA-Slot des anfordernden Caches

Die zuvor beschriebenen Probleme, die beim Zurückschreiben modifizierter Daten auftreten, lassen sich beheben, indem die Broadcast-Funktionalität des Busses genutzt wird. Der bereitstellende Cache legt die Daten, beim Zurückschreiben in den gemeinsamen Speicher, für den anfordernden Cache sichtbar am Bus an. So können die Daten beim Zurückschreiben direkt in den anfordernden Cache geladen werden. Diese Möglichkeit wird z.B. von Bus-Snarfing Protokollen genutzt. Durch die direkte Übermittlung an den anfordernden Cache wird auch die Möglichkeit eines Aushungerns unterbunden. Geht man davon aus, dass das

Zurückschreiben nach der Unterbrechung des Lesezugriffs im nächsten eigenen TDMA-Slot des anfordernden Caches stattfindet, dann beträgt die maximale Latenz für einen Cache-Miss zwei TDMA-Perioden:

$$\Delta_{CacheMiss_{Bus}} = 2 * \Delta_{TDMA} \quad (3.7)$$

Die untersuchten Varianten für das Zurückschreiben modifizierter Cache-Zeilen in einem echtzeitfähigen Bus erhöhen teilweise massiv die Zugriffslatenz für einen Cache-Miss. Eine der Varianten macht eine statische WCET Analyse sogar undurchführbar. Eine präzise WCET-Abschätzung ist bei derart erhöhten Zugriffslatenzen nicht zu erreichen.

Zugriffslatenz in Crossbar-Systemen

Betrachtet man die zeitlichen Abläufe und Abhängigkeiten bei der TDMA-Arbitrierung, so gibt es viele Gemeinsamkeiten zwischen bei der Anwendung in einem Bus und einer Crossbar. In beiden Fällen werden die Zugriffe aller Rechenkerne auf den gemeinsamen Speicher sequentiell, in der vom TDMA-Schema bestimmten Reihenfolge, durchgeführt. Die zuvor erläuterten Auswirkungen auf die Zugriffslatenzen sind daher zum Teil auf Systeme mit Crossbar übertragbar.

Um ein Snooping-Protokoll in einem System mit einer Crossbar zu realisieren, müssen Kohärenznachrichten als Broadcast an die anderen Rechenkerne gesendet werden. Dabei muss beachtet werden, dass Rechenkerne in einer Crossbar ihre Zugriffe zeitgleich über verschiedene Kommunikationskanäle durchführen können. Es stellt sich die Frage, inwieweit diese Broadcasts innerhalb der Crossbar arbitriert werden können und welche Auswirkungen dies auf ein Kohärenzprotokoll hat. Das macht eine Bestimmung maximaler Zugriffslatenzen, mit Berücksichtigung der Interaktion der Teilnehmer, abhängig von zahlreichen Details der Implementierung der Crossbar. Es wäre hier nicht zielführend über mögliche Implementationen zu spekulieren. Nach bestem Wissen des Autors, ist die einzige Architektur mit einer Crossbar, die ein Snooping-Protokoll einsetzt, die QorIQ Architektur von Freescale [21]. Die darin eingesetzte CoreNet Fabric ermöglicht eine direkte Übermittlung modifizierter Cache-Zeilen, ähnlich dem Bus-Snarfing. In einer solchen Architektur wird bei einem Cache-Miss die Anfrage entweder vom gemeinsamen Speicher oder direkt vom bereitstellenden Cache beantwortet. Es ist nicht genauer spezifiziert, ob es bei diesen beiden Möglichkeiten einen Unterschied in der Zugriffslatenz gibt. In der Annahme, dass es für den anfordernden Rechenkern keine Rolle spielt, von welcher Quelle er die Daten erhält, hat das Zurückschreiben der

Daten in einem solchen Fall keine negative Auswirkung auf die Cache-Miss Latenz:

$$\Delta_{CacheMiss_{Crossbar}} = 2 * \Delta_{TDMA} \quad (3.8)$$

3.3.4 Cache-Miss Latenz in Verzeichnis-Protokollen

Die Klasse der verzeichnisbasierten Kohärenzprotokolle wird größtenteils in netzartigen Topologien, aber auch in Systemen mit Crossbar eingesetzt. Anstatt Kohärenznachrichten als Broadcast über das System zu schicken, werden diese gezielt nur an die betroffenen Teilnehmer gesendet. Dazu ist es nötig einen Umweg über den Träger des Kohärenzverzeichnisses zu nehmen. Dieser Umweg spiegelt sich natürlich auch in den maximalen Zugriffslatenzen wieder. Für die Nachricht an den Träger des Verzeichnisses wird ein erster Hop fällig (vgl. Kapitel 2.5.4). Zugriffe auf gemeinsame Daten können bei einem Verzeichnis-Protokoll eine Vielzahl einzelner Kohärenznachrichten auslösen, die zwischen den beteiligten Rechenkernen versendet werden. Eine bedeutende Rolle spielt dabei die Schreibstrategie des Cache-Speichers.

Latenz bei Write-Back Schreibstrategie

In einem Verzeichnis-Protokoll führt der Zugriff auf ein gemeinsames Datum zuerst zu einem Zugriff auf das Kohärenz-Verzeichnis. Dieses Verzeichnis liegt entweder im gemeinsamen Speicher, oder es ist auf die lokalen Speicher der einzelnen Rechenkerne verteilt [86]. Sofern der anfordernde Cache nicht am selben Knoten liegt wie das benötigte Verzeichnis, so wird für den Zugriff auf das Verzeichnis ein erster Hop durch das Netzwerk fällig. Im Falle eines Write-Back Caches befindet sich möglicherweise eine modifizierte Kopie des gesuchten Datums in einem anderen Cache. Das Kohärenz-Verzeichnis sendet daraufhin eine Nachricht zum entsprechenden Cache, mit der Aufforderung das Datum bereitzustellen. Das Senden dieser Aufforderung fügt einen zweiten Hop zur Gesamtlatenz für einen Cache-Miss hinzu. Die Bereitstellung der modifizierten Cache-Zeile kann auf zwei Arten geschehen [23]:

1. **Zurückschreiben der Cache-Zeile in den gemeinsamen Speicher**
Das Kohärenz-Verzeichnis kann den bereitstellenden Cache auffordern die modifizierte Cache-Zeile in den gemeinsamen Speicher zurückzuschreiben. Dieses Zurückschreiben erzeugt einen weiteren Hop. Da ein Kohärenz-Verzeichnis immer zusammen mit den Daten die es verwaltet gespeichert ist, wird gleichzeitig der entsprechende Eintrag für die Cache-Zeile aktualisiert. Daraufhin kann das Verzeichnis die Übermittlung der aktualisierten Cache-Zeile an den anfragenden Cache in Gang setzen.

Dafür wird wieder ein Hop benötigt. Insgesamt entspricht die maximale Latenz für einen Cache-Zugriff der Dauer von vier Nachrichtenübermittlungen:

$$\Delta_{CacheMiss_{Network}} = 4 * \Delta_{NetworkTraversal} \quad (3.9)$$

2. Übertragung der Cache-Zeile zum anfragenden Cache

Anstatt die modifizierte Cache-Zeile in den Speicher zurückzuschreiben, kann der bereitstellende Cache die Daten in einer direkten Nachricht an den anfordernden Cache übermitteln [23]. Gleichzeitig sendet er eine Nachricht an den Heimatknoten der Cache-Zeile, um das Kohärenzverzeichnis von der veränderten Situation in Kenntnis zu setzen. Auf diese Art fallen sind 3 Hops als maximale Cache-Miss Latenz für den anfordernden Rechenkern an:

$$\Delta_{CacheMiss_{Network}} = 3 * \Delta_{NetworkTraversal} \quad (3.10)$$

Latenz bei Write-Through Schreibstrategie

Eine Write-Through Strategie hat auch bei Verzeichnis-Protokollen den positiven Effekt, dass Cache-Speicher von der Aufgabe befreit sind, modifizierten Daten anderen Caches zur Verfügung zu stellen. Die angeforderte Cache-Zeile kann direkt als Antwort für die Anfrage an das Kohärenz-Verzeichnis zurückgesendet werden [115]. Dafür werden lediglich zwei Hops benötigt. Die maximale Latenz für einen Cache-Miss entspricht der Latenz in einem System ohne Kohärenzverfahren.

$$\Delta_{CacheMiss_{Network}} = 2 * \Delta_{NetworkTraversal} \quad (3.11)$$

3.3.5 Latenz bei Schreibzugriffen

Schreibzugriffe auf gemeinsame Daten haben bei Kohärenzprotokollen immer einen besonderen Stellenwert. Sind es doch diese Zugriffe, die eine Inkonsistenz der Daten im Speichersystem erzeugen. Im vorherigen Kapitel wurden die Latenzen für Cache-Hits und Cache-Misses behandelt, jedoch unter der Prämisse, dass es sich um Lesezugriffe handelt. In diesem Kapitel soll es um die Latenz gehen, die durch eine Kohärenzoperation zusätzlich erzeugt wird, wenn es sich um einen Schreibzugriff handelt.

In einem System ohne Kohärenzprotokoll gilt ein Schreibzugriff auf ein Datum im Cache als abgeschlossen, sobald das entsprechende Datum im Cache modifiziert wurde. Aus der Sicht eines Rechenkerns kann mit der Ausführung weiterer Instruktionen fortgefahren werden, sobald der Schreibzugriff an den Cache weitergegeben wurde. Wird jedoch mehr als nur ein einfacher Schreibzugriff auf den Cache durchgeführt, so kann auch die Ausführung nachfolgender Cache-Zugriffe verzögert werden. Arbeitet der Cache mit einer Write-Through Schreibstrategie, so wird ein zusätzlicher Zugriff auf das Kommunikationsmedium und den gemeinsamen Speicher durchgeführt. Im Falle einer Write-Allocate Strategie kann dazu noch ein Cache-Miss auftreten. Diese Verzögerung nachfolgender Zugriffe muss also in die maximale Latenz eines Schreibzugriffes mit einbezogen werden. Die Latenz bei Cache-Misses wurde bereits in den vorhergehenden Kapiteln behandelt. Hier soll es in erster Linie um die maximale Latenz für einen Schreibzugriff mit Cache-Hit gehen ($\Delta_{CacheWriteHit}$). In einem Bus-System mit inkohärentem Cache und Write-Through Strategie, muss die statische Cacheanalyse für den Cache-Hit nach Schreibzugriff in etwa die gleiche Latenz wie für einen Cache-Miss annehmen:

$$\Delta_{CacheWriteHit_{Bus}} = \Delta_{TDMA} \quad (3.12)$$

Latenz bei Write-Invalidate Protokollen

Bei Anwendung eines Write-Invalidate Kohärenzprotokolls müssen bei einem Schreibzugriff auf ein geteiltes Datum alle existierenden Kopien in anderen Caches invalidiert werden. Cache-Zeilen besitzen daher Zustandsinformationen die angeben, ob noch weitere Kopien derselben Zeile in anderen Caches vorhanden sind. Schreibzugriffe auf Daten, die als geteilt markiert sind, lösen zuvor eine Invalidierungsnachricht aus. Diese Invalidierungsnachricht wird über das Kommunikationsmedium an die betroffenen Caches versandt. Invalidierungsnachrichten sind also den selben Zugriffszeiten am Kommunikationsmedium unterworfen, wie reguläre Zugriffe. Es stellt sich die Frage, ob ein Cache, im Anschluss an die Versendung einer Invalidierungsnachricht, direkt mit dem Schreibzugriff beginnen kann oder ob er auf die Bestätigung der erfolgreichen Invalidierung in den anderen Caches warten muss.

Ein Schreibzugriff erzeugt kein vom Rechenkern benötigtes Ergebnis, so dass ein Cache im Normalfall nicht auf die vollständige Durchführung des Schreibzugriffes in tieferen Ebenen der Speicherhierarchie warten muss. Nachfolgende Cache-Zugriffe können unverzüglich behandelt werden. Für Invalidierungsnachrichten in MESI-basierten Protokollen gilt dies nicht, da diese eine Änderung im Cache der Empfänger auslösen, die für den Sender ausschlaggebend ist.

Angenommen zwei Rechenkerne besitzen jeweils eine Kopie des Datums x in ihrem Cache und die entsprechende Cache-Zeile ist in beiden Caches als geteilt markiert. Beide Rechenkerne möchten nun zeitgleich einen Schreibzugriff auf Datum x ausführen und versenden eine entsprechende Invalidierungsnachricht für den anderen Cache. Wenn anschließend beide Caches den Schreibzugriff durchführen ohne auf die Bestätigung der Invalidierung zu warten, so besäßen beide eine modifizierte Kopie der gleichen Cache-Zeile. Ein solcher Zustand ist in einem MESI-Protokoll nicht erlaubt, da nicht mehr sichergestellt werden kann, dass nachfolgende Zugriffe auf dieses Datum kohärent sind.

Um die Kohärenz von Speicherzugriffen zu garantieren, muss folgende Bedingung bei der Handhabung von Invalidierungsnachrichten erfüllt sein:

Versendet ein Cache im Zuge eines Schreibzugriffs eine Invalidierungsnachricht für eine Cache-Zeile, so darf der Schreibzugriff erst dann durchgeführt werden, wenn sichergestellt ist, dass die entsprechende Cache-Zeile bei allen Empfängern der Nachricht invalidiert wurde.

Ein Cache muss also auf die Beendigung seiner Invalidierungs-Phase, bestehend aus Übermittlung der Invalidierungsnachricht und Empfang aller Invalidierungs-Bestätigungen, warten. Desweiteren sollte vermieden werden, dass sich beide Caches zeitgleich in einer Invalidierungs-Phase für die gleiche Cache-Zeile befinden. Dies würde schlussendlich dazu führen, dass beide ihre Cache-Zeile invalidieren und einen Cache-Miss durchführen müssen.

Die Latenz der Invalidierungs-Phase $\Delta_{Invalidation}$ muss in die maximale Latenz für einen Schreibzugriff miteinbezogen werden. Im Falle eines Cache-Misses kann die Invalidierungsnachricht als Teil der Anforderung der Daten für die Cache-Zeile (Bus- und Verzeichnis basierte Protokolle) oder als Broadcast (Snooping Protokoll in Crossbar) an alle Teilnehmer versendet werden. Die Latenz der Invalidierungs-Phase kann unter Umständen ganz oder teilweise von der Latenz des dazugehörigen Cache-Misses verborgen werden:

$$\Delta_{CacheWriteMiss} = \max(\Delta_{CacheMiss}, \Delta_{Invalidation}) \quad (3.13)$$

Im Falle eines Cache-Hits in einem Write-Back Cache muss jedoch eine neue Invalidierungsnachricht erzeugt und versendet werden:

$$\Delta_{CacheWriteHit} = \Delta_{Invalidation} + \Delta_{CacheCycle} \quad (3.14)$$

Die genaue Latenz einer Invalidierungs-Phase hängt wiederum vom Kommunikationsmedium ab.

In busbasierten Systemen mit TDMA-Arbitrierung wird eine Invalidierungsnachricht als Broadcast im TDMA-Slot des Senders übertragen. Im Falle eines Cache-Misses wird sie zeitgleich mit dem Lesezugriff an den Bus gelegt. Man kann davon ausgehen, dass diese Invalidierung bei einem Snooping-Protokoll von allen Rechenkernen gelesen wird, und auf eine Bestätigung verzichtet werden kann. Anders gesagt wird eine Bestätigung der Invalidierung noch im selben TDMA-Slot impliziert, so dass die Latenz der eines Cache-Misses entspricht:

$$\Delta_{Invalidation_{Bus}} = \Delta_{TDMA} \quad (3.15)$$

In Systemen mit einer Crossbar wird die Invalidierungsnachricht als Broadcast zu allem anderen Caches gesendet. Von einer impliziten Bestätigung der Invalidierung kann hier nicht ausgegangen werden. Die Bestätigung der Invalidierung ist eine eigenständige Nachricht eines jeden Caches. Bis alle Bestätigungen beim Sender eintreffen vergeht also ein weiterer TDMA-Zyklus. Dies erhöht die Zugriffslatenz sowohl für Cache-Hits als auch für Cache-Misses:

$$\Delta_{Invalidation_{Crossbar}} = 2 * \Delta_{TDMA} \quad (3.16)$$

In einem Verzeichnis-Protokoll wird die Benachrichtigung für einen Speicher-Zugriff zuerst zum Verzeichnis gesendet. Dieses sendet dann gezielt Invalidierungsnachrichten an alle Caches die das entsprechende Datum in ihrem Cache haben. Teil dieser Invalidierungsnachricht ist der Identität des Senders, so dass die Bestätigungen direkt an ihn geschickt werden können. Der gesamte Vorgang umfasst damit 3 Hops:

$$\Delta_{Invalidation_{Network}} = 3 * \Delta_{NetworkTraversal} \quad (3.17)$$

Latenz bei Write-Update Protokollen

In Write-Update Protokollen wird anstatt einer Invalidierungsnachricht, eine Nachricht mit dem modifiziertem Datum an die beteiligten Caches gesendet. Doch ist nicht bei jedem Schreibzugriff ist eine Aktualisierung notwendig. Die Behandlung eines Schreibzugriffes hängt vom Zustand der modifizierten Cache-Zeile ab. Damit ergeben sich auch unterschiedliche maximale Latenzen für Schreibzugriffe.

Ist eine Cache-Zeile als einzige Kopie im System markiert (z.B. *Exclusive*), so können Schreibzugriffe völlig ohne die Versendung einer Kohärenz-Nachricht

stattfinden. Die Latenzen sind dann identisch mit Latenzen eines Cache-Hits in Systemen ohne Kohärenztechnik:

$$\begin{aligned}\Delta_{CacheWriteHit_{Bus}} &= \Delta_{CacheCycle} \\ \Delta_{CacheWriteHit_{Crossbar}} &= \Delta_{CacheCycle} \\ \Delta_{CacheWriteHit_{Network}} &= \Delta_{CacheCycle}\end{aligned}\tag{3.18}$$

Wenn eine Cache-Zeile als geteilt markiert ist, so müssen Änderungen der Daten in Folge eines Schreibzugriffs an alle Caches weitergegeben werden, die eine Kopie dieser Cache-Zeile besitzen. Solche Schreibzugriffe werden dann als direkte Übermittlung an die anderen Caches über das Kommunikationsmedium realisiert. Ein solcher Schreibzugriff gilt erst dann als abgeschlossen, wenn die Daten in allen Kopien entsprechend geändert wurden.

In einem Bussystem ist eine globale Reihenfolge solcher Schreibzugriffe gegeben. Der Schreibzugriff eines Rechenkerns auf ein geteiltes Datum wird an den Bus gelegt und von allen anderen Rechenkernen empfangen, bevor ein zweiter Rechenkern seinen Schreibzugriff durchführt. Ein Rechenkern muss daher, nach einem Schreibzugriff auf ein geteiltes Datum, so lange mit der Fortsetzung der Ausführung warten, bis der Schreibzugriff über das Kommunikationsmedium übermittelt wurde. Die maximale Latenz entspricht der eines Write-Through Zugriffs:

$$\Delta_{Update_{Bus}} = \Delta_{TDMA}\tag{3.19}$$

Write-Update Protokolle benötigen eine effiziente Umsetzung von Broadcast bzw. Multicasts um Aktualisierungen anderer Cache-Zeilen schnell durchführen zu können. Bussysteme sind dafür besonders gut geeignet. Bei Verbindungstopologien, in denen mehrere Rechenkerne gleichzeitig auf das Kommunikationsmedium zugreifen können (Crossbar, Mesh etc.), ist ein Write-Update Protokoll nicht ohne weiteres umsetzbar. Um eine Wettlaufsituation zu vermeiden, kann nur jeweils einem Cache das Recht gewährt werden, eine geteilte Cache-Zeile zu modifizieren. Nur dieser Besitzer (*Owner*) darf auf diese Cache-Zeile schreiben und die Änderungen an andere Caches weitergeben. Alle anderen Caches die eine Kopie dieser Cache-Zeile besitzen sind Teilhabende (*Sharer*). Jeder Schreibzugriff eines Owners einer Cache-Zeile erzeugt eine Aktualisierungsnachricht. Es spricht nichts dagegen, dass der Owner mit der Ausführung fortfährt ohne Bestätigungen der Sharer einzuholen. Die maximale Latenz Δ_{Update} entspricht dann einem einfachen Zugriff auf das Kommunikationsmedium:

$$\begin{aligned}\Delta_{Update_{Crossbar}} &= \Delta_{TDMA} \\ \Delta_{Update_{Network}} &= \Delta_{NetworkTraversal}\end{aligned}\tag{3.20}$$

Als Teilhaber/Sharer einer Cache-Zeile hat man noch nicht das Recht Schreibzugriffe darauf durchzuführen. Zuvor muss eine Besitzanforderung an den aktuellen Owner versendet werden. Eine Besitzanforderung entspricht einer Aktualisierungsnachricht mit Invalidierung der Cache-Zeile des bisherigen Besitzers. Für diese Nachricht muss eine Bestätigung abgewartet werden. Das führt zu Latenzen, die denen einer Invalidierungsnachricht entsprechen:

$$\begin{aligned}\Delta_{Update_{Crossbar}} &= 2 * \Delta_{TDMA} \\ \Delta_{Update_{Network}} &= 3 * \Delta_{NetworkTraversal}\end{aligned}\tag{3.21}$$

3.3.6 Vorhersagbarkeit von Schreiblatenzen

Die maximale Latenz eines Schreibzugriffs, die in die statische WCET-Analyse einbezogen wird, ist wie im vorherigen Kapitel dargestellt nicht nur davon abhängig ob es ein Cache-Hit oder Cache-Miss ist, sondern kann auch vom Kohärenz-Zustand der Cache-Zeile abhängen. Zugriffe auf Cache-Zeilen, die nicht von anderen Caches geteilt werden, weisen deutlich geringere Latenzen auf, als Zugriffe die eine Kohärenznachricht auslösen. Eine Cacheanalyse sollte daher eine Vorhersage dieser Kohärenz-Zustände mit in die Ermittlung der $WCET_{est}$ einbeziehen.

Die sichere Voraussage des Kohärenz-Zustands einer Cache-Zeile ist jedoch, ähnlich wie die Vorhersage von Cache-Hits, nicht so einfach möglich. Während einige Zustandsänderungen vom Rechenkern selbst erzeugt werden (z.B. *Invalid* -> *Exclusive* oder *Shared* -> *Modified*), werden andere von externen Kohärenznachrichten ausgelöst (z.B. *Exclusive* -> *Shared*). So kann z.B. eine Cache-Zeile, die als einzige Kopie im System markiert ist, mit der nächstmöglichen Kohärenz-Nachricht zu einer geteilten Cache-Zeile werden. Die Berücksichtigung von externen Änderungen des Kohärenzzustands, analog zur Berücksichtigung von externen Invalidierungen, führt dazu dass Cache-Zeilen im Worst-Case tendenziell immer als geteilt betrachtet werden müssen. Für die Ermittlung der maximalen Latenz eines Schreibzugriffs hat dies zur Folge, dass für jeden Zugriff eine zu bestätigende Versendung von Kohärenznachrichten miteinbezogen werden muss, mit entsprechenden Auswirkungen auf die Worst-Case Laufzeit.

3.3.7 Schlussfolgerungen

Aus der Analyse der Auswirkungen von Kohärenzoperationen auf die Cacheanalyse und damit auf die zeitliche Vorhersagbarkeit von Anwendungen, lassen sich Schlussfolgerungen ziehen, mit welchen Mitteln echtzeitfähige Cache-Kohärenz zu erreichen ist. Die Schlussfolgerungen beziehen sich in erster Linie auf Kohärenzoperationen, die vermieden werden sollten. Gemeint sind all

jene, die eine Vorhersagbarkeit von Cache-Hits beeinträchtigen und die Zugriffslatenzen auf den Cache unverhältnismäßig erhöhen. Im Folgenden sind die Anforderungen an eine echtzeitfähige Cache-Kohärenz zusammengefasst:

1. Vermeidung einer Verzögerung von Cache-Zugriffen

Der Beginn eines jeden Cache-Zugriffs ist die Überprüfung, ob das gefragte Datum im Cache vorhanden ist. Es sollte keine Verzögerung dieses Zugriffs durch externe Kohärenztransaktionen entstehen. Andernfalls müsste eine zusätzliche Latenz für jeden einzelnen Cache-Zugriff hinzugechnet werden, was eine signifikante Erhöhung der $WCET_{est}$ zur Folge haben würde.

2. Vermeidung von Invalidierungsnachrichten

Durch Invalidierungsnachrichten ausgelöste Invalidierungen von Cache-Zeilen richten den größten Schaden im Sinne einer Cacheanalyse an. Sie verhindern die Vorhersage von Cache-Hits und sind daher gänzlich zu vermeiden. Damit gilt die gesamte Klasse der Write-Invalidate Protokolle als nicht anwendbar.

3. Vermeidung einer Verzögerung von Cache-Hits

Eine sichere Vorhersage von Cache-Hits führt nur dann zu einer brauchbaren WCET-Abschätzung, wenn die maximale Latenz eines Cache-Hits nicht übermäßig erhöht ist. Eine kurze und konstante maximale Cache-Hit Latenz sorgt dafür, dass der Cache seinen Nutzen auch bei einer WCET-Analyse unter Beweis stellen kann. Kohärenzoperationen, die negative Auswirkungen auf die Latenz eines Cache-Hits haben, sind daher zu vermeiden.

4. Write-Through als Schreibstrategie

Eine Write-Through Schreibstrategie aktualisiert bei jedem Schreibzugriff neben dem privaten Cache auch den gemeinsamen Speicher. Im Vergleich zu einer Write-Back Strategie treten dadurch im System deutlich mehr Speicherzugriffe auf. Diesem Nachteil steht jedoch der Vorteil entgegen, dass sich der aktuellste Wert eines Datums immer im gemeinsamen Speicher befindet. Dies hat den Effekt, dass ein Cache-Miss immer vom Hauptspeicher bedient werden kann. Es entfällt das Zurückschreiben einer modifizierten Cache-Zeile eines anderen Caches. Eine statische Laufzeitanalyse profitiert sehr von diesem Effekt. Im Zusammenspiel mit einer echtzeitfähigen Arbitrierung ist die maximale Latenz eines Cache-Misses nicht mehr von anderen Rechenkernen abhängig. Sie entspricht der Latenz ohne Einwirkung eines Kohärenzprotokolls. Eine Write-Through Schreibstrategie ist also, zumindest für die Cache-Miss Latenz, eine optimale Wahl [36].

Keine der untersuchten Hardware Kohärenztechniken ist in der Lage, jede dieser Anforderungen zu erfüllen. Hardware Kohärenztechniken basieren auf der Interaktion zwischen Cache-Speichern und wenden daher immer Kohärenztransaktionen an. Die Verwendung eines Write-Update Protokolls mit Write-Through Schreibstrategie ist hinsichtlich zeitlicher Vorhersagbarkeit die bestmögliche Variante eines reinen Hardware-Protokolls [137]. Jedoch resultiert auch diese Variante in stark erhöhten Zugriffs latenzen in Folge der Handhabung von Schreibzugriffen. In Anbetracht der erhöhten Anzahl an Zugriffen auf den gemeinsamen Speicher ist mit einer massiv erhöhten WCET-Abschätzung zu rechnen.

Eine Kohärenztechnik, die den Anforderungen eines harten Echtzeitsystems gänzlich standhält, muss auf unvorhersehbare Interaktionen zwischen Caches verzichten. Eine Interaktion zwischen Cache-Speichern, die den Inhalt oder den Zustand einer Cache-Zeile beeinflusst, sollte nicht Teil eines Kohärenzprotokolls sein.

Mit einer reinen Hardware Kohärenztechnik ist dies nicht möglich. Ein Teil der Verantwortung muss von der Software übernommen werden. Ein echtzeitfähiges Cache-Kohärenzprotokoll sollte eine Kombination aus zeitlich vorhersagbaren Techniken der Hardware und Softwarekohärenz einsetzen. Dabei sollte der Fokus auf eine möglichst hohe Performanz unter Beibehaltung der zeitlichen Vorhersagbarkeit liegen.

3.4 Echtzeitfähige Speicherhierarchie in der Forschung

Der dringende Bedarf nach einer echtzeitfähigen Speicherhierarchie für (Mehrkern-)Prozessoren und der Mangel einer solchen in aktuellen Architekturen wurde von der Forschung wahrgenommen [132] [18][41][145]. Zahlreiche Projekte und Forschungsarbeiten widmen sich der Entwicklung einer Prozessorarchitektur für Echtzeitsysteme [6][17]. Im Folgenden werden einige Arbeiten vorgestellt, die interessante Ansätze für eine zeitlich vorhersagbare Speicherhierarchie beinhalten.

Unter dem Begriff *Precision Timed Machine* (PRET) stehen eine Vielzahl von Forschungsarbeiten (u.a. [88][89][22]), die das Ziel haben, zeitliche Vorhersagbarkeit und Determinismus in den Fokus einer Systementwicklung zu stellen. Dabei sollen sämtliche Entwicklungsphasen, software- wie hardwareseitig miteinbezogen werden. Allerdings beschränkt sich die zugrundeliegende Systemarchitektur auf einen mehrfädigen Einkern-Prozessor. Cache-Speicher werden bei PRET als zeitlich unzureichend vorhersehbar eingestuft und es wird gänzlich auf sie verzichtet. Anstelle eines Caches werden Scratchpad-Speicher verwendet, die schnelle und zeitlich vorhersagbare Zugriffe auf Programmcode und Daten bieten. Die Speicherhierarchie einer PRET-Architektur ist eine plausible Lösung für mehrfädige Einkern-Prozessoren. Es ist jedoch frag-

lich ob sich dieser Ansatz auf Mehrkern-Prozessoren ausweiten lässt. In einem Mehrkern-Prozessor ist der Zugriff auf gemeinsame Daten nur dann effizient möglich, wenn jeder Rechenkern schnellen Zugriff auf diese Daten hat. Verfügen die Rechenkern nur über private Scratchpads, so ist ein Datum immer nur für einen Rechenkern schnell zugreifbar.

Scratchpad-Speicher werden in zahlreichen weiteren Ansätzen als Ersatz für Cache-Speicher angeführt [55][144][140]. In der PROMPT Architektur (Predictability Of Multi-Processor Timing) [36] wird der Einsatz von Caches an die Bedingung geknüpft, dass diese hinreichend vorhersagbar sind. Es wird angeraten, Daten- und Instruktionscaches zu trennen, sowie eine LRU Ersetzungsstrategie und Write-Through Schreibstrategie zu verwenden. Auf das Problem der Cache-Kohärenz wird nicht eingegangen.

Eine Möglichkeit, Zugriffe auf einen Cache vorhersagbar zu gestalten, ist die zeitliche und räumliche Isolierung bestimmter Bereiche des Cache-Speichers. Dieser Ansatz wird in Forschungsarbeiten über *Partitioning* und *Locking* behandelt [142][128][87]. Dabei hat ein Core/Task (für eine gewisse Zeit) exklusiven und damit vorhersagbaren Zugriff auf einen Bereich des Caches. Entsprechende Ansätze findet man auch unter der Bezeichnung *Cache-Coloring* [143][77].

Zu dem Zweck die Analysierbarkeit von Cache-Zugriffen zu erhöhen, schlägt Schoeberl [119] eine Aufteilung in mehrere separate Caches vor (Instruktions-Cache, Stack-Cache, Static-Cache etc.) um schwierig und einfach zu analysierende Zugriffe zu trennen. So ist in dieser Speicherhierarchie der *Static-Cache* für den Zugriff auf statisch zugewiesene Daten zuständig. Da dazu auch gemeinsame Daten gehören, muss für diesen Cache ein Kohärenzprotokoll implementieren werden. Diese Aufteilung beschränkt die durch Kohärenzverfahren beeinträchtigte Vorhersagbarkeit auf Zugriffe auf bestimmte Caches wie den Static-Cache. Analog zum Stack-Cache ist ein separater Cache für den Heap vorgesehen. Da Zugriffe auf den Heap besonders schwer zu analysieren sind, wird hierfür ein Object-Cache [120] eingesetzt. Dieser Cache setzt ein softwarebasiertes Kohärenzverfahren [101] ein, welches Invalidierungen des gesamten Caches an Synchronisationspunkten und Lesezugriffen auf *Volatile*-Variablen durchführt. Dadurch werden kohärente Zugriffe auf gemeinsame Daten sichergestellt. Jedoch schränkt die wiederholte Invalidierung des gesamten Cache-Speichers eine effiziente Nutzung desselben sehr ein.

Bei Lundquist et al. [90] werden Zugriffe mit Hilfe des Compilers dahingehend bewertet, ob sie von einer Cacheanalyse einfach vorherzusagen sind. Besteht die Möglichkeit, dass die Zieladresse eines Zugriffs nicht ermittelt werden kann, so wird der Zugriff ohne Beteiligung des Caches durchgeführt. Weitere Ansätze einen (Instruktions-)Cache selektiv für klassifizierte Zugriffe zu verwenden, wurden in [148][57] vorgestellt.

Schoeberl et al. [123][122] stellen eine echtzeitfähige Variante von Trans-

aktional Memory vor. Es wird gezeigt, dass für periodische Tasks eine obere Schranke für die maximale Neuausführung einer Transaktion angegeben werden kann, und so die Abschätzung einer WCET möglich ist. Um die Überabschätzung der WCET zu verringern ist dieser Ansatz auf wenige Zugriffe pro Transaktionen beschränkt.

Kapitel 4

On-Demand Coherent Cache

Der On-Demand Coherent Cache (ODC²) [107] ist ein Verfahren zur Erhaltung der Cache-Kohärenz in Mehrkern-Prozessoren. Der ODC² verfolgt ein Konzept, das grundlegend auf die Anforderungen von harten Echtzeitsystemen zugeschnitten ist und sich dabei Techniken der Software-Kohärenz bedient. Die Intention dieses Verfahrens ist einerseits, einen zeitlich vorhersagbaren Zugriff auf private sowie gemeinsame Daten sicherzustellen. Die zeitliche Vorhersagbarkeit muss den Anforderungen einer statischen Laufzeitanalyse genügen, so dass der ODC² in harten Echtzeitsystemen einsetzbar ist. Aus der Analyse gängiger Kohärenzverfahren in Kapitel 3 lässt sich schließen, dass eine hinreichende zeitliche Vorhersagbarkeit mit einer rein hardwarebasierten Kohärenztechnik nicht zu erreichen ist. Der ODC² vermeidet daher Techniken, die eine Kommunikation zwischen den Cache-Speichern bedingen. Zum anderen ist der ODC² darauf ausgerichtet, eine möglichst effiziente Nutzung des Cache-Speichers zu ermöglichen. So ergibt sich eine signifikante Steigerung der Performanz im Vergleich zur Durchführung von Speicher-Zugriffen unter Umgehung des Caches. Dies wird erreicht, indem die Handhabung von Datenzugriffen weitestgehend der eines herkömmlichen inkohärenten Cache-Speichers entspricht, und Kohärenzoperationen nur dann angewendet werden, wenn sie tatsächlich benötigt werden. Mit Hilfe spezieller Instruktionen innerhalb des Programmcodes ist der ODC² in der Lage, solche Zugriffe zu identifizieren und entsprechende Maßnahmen zu ergreifen.

Das Verfahren zur Erhaltung der Kohärenz im ODC² wurde außerdem im Hinblick auf eine möglichst einfache Realisierbarkeit entwickelt. Die zusätzlich benötigte Logik in der Cache-Hardware ist gering und bringt keinen unverhältnismäßig erhöhten Overhead in Bezug auf Chipfläche und Kosten mit sich.

Auch die Anforderungen an eine Erweiterung der Software sind verhältnismäßig, so dass dieser zusätzliche Aufwand für die Anwendung des ODC² keine Hürde darstellen sollte.

Das Konzept des ODC² unterscheidet sich von den bekannten, auf Average-Case Performanz ausgelegten Kohärenzverfahren, hinsichtlich der Handhabung der Konsistenz geteilter Daten. Cache-Kohärenzprotokolle erlauben üblicherweise eine Inkonsistenz der Daten zwischen dem Hauptspeicher und den beteiligten Caches, bewahren jedoch mittels spezieller Kommunikation untereinander die Konsistenz der Daten zwischen allen Cache-Speichern. So bleiben die Zugriffe auf gemeinsame Daten über den gesamten Zeitraum der Ausführung eines Programms kohärent. Beim ODC² hingegen werden Maßnahmen zur Erhaltung der Kohärenz nur bei Bedarf durchgeführt. Zugriffe auf private Daten werden im ODC² stets als kohärent betrachtet und bedürfen keiner zusätzlichen Behandlung. Lediglich beim Zugriff auf gemeinsame Daten ist es notwendig, Kohärenz sicherzustellen. Der ODC² wird daher nur in zuvor festgelegten Programmabschnitten aktiviert, in denen auf gemeinsame Daten zugegriffen wird und somit Inkohärenz auftreten kann. Das Verfahren ist auf die Ausführung parallelisierter Applikationen zugeschnitten, die mittels Synchronisation einen geregelten Zugriff auf gemeinsame Daten sicherstellen. Damit ist gemeint, dass mittels Synchronisation eine Wettlaufsituation (*race condition*) beim Zugriff auf gemeinsame Daten vermieden wird. Der ODC² greift bei der Ausführung auf solche Synchronisationstechniken zurück. Generell ist die korrekte Synchronisation der Prozesse einer parallelisierten Applikation eine Voraussetzung für eine korrekte Funktion. Für Applikationen in harten Echtzeitsystemen sollte eine korrekte Synchronisation als gegeben angesehen werden, so dass sie keine wesentliche Einschränkung für den Einsatz des ODC² darstellt.

Programmabschnitte, in denen der ODC² die Kohärenz von Zugriffen auf gemeinsame Daten sicherstellen muss, sind durch die Synchronisationspunkte der parallelen Applikation definiert. Im Sinne des ODC² sind in diesen Abschnitten Kopien gemeinsamer Daten in mehreren Cache-Speichern nur dann erlaubt, wenn diese Daten (auch im Bezug zum Hauptspeicher) konsistent bleiben. Dies ist z.B. bei Zugriffen auf Nur-Lese (Read-Only) Daten der Fall. Modifiziert ein Rechenkern seine Kopie im Cache, so verlangt das Konzept des ODC², dass keine weitere Kopie des Datums in einem anderen Cache existiert. Durch gezielte Invalidierung gemeinsamer Daten im Cache und in Zusammenarbeit mit der im Programm eingesetzten Synchronisationstechnik erreicht der ODC² eine Konsistenz der Daten über alle Caches hinweg. Dadurch wird verhindert, dass die Kopie eines Datum durch die Modifikation eines anderen Rechenkerns veraltet bzw. ungültig wird. Somit entfällt die Notwendigkeit einer Kommunikation zwischen den Caches und damit auch die Hauptquelle für die beschränkte zeitliche Vorhersagbarkeit von Cache-Kohärenzprotokollen.

Im Folgenden wird die Funktionsweise des ODC² detailliert erläutert und darauf eingegangen, wie Applikationen für den Einsatz des ODC² erweitert werden müssen. Zuerst werden die Grundfunktionen beschrieben und zusätzliche Funktionalitäten erläutert, die unter bestimmten Voraussetzungen eingesetzt werden können. Es werden die unterschiedlichen Voraussetzungen an Hardware und Software diskutiert, die für eine Realisierung des ODC² nötig sind. Zuletzt wird auf den Einfluss der Kohärenzoperationen auf die zeitliche Vorhersagbarkeit des Systems eingegangen.

4.1 Funktionsweise

Der ODC² entspricht einem kooperativen Hardware/Software Kohärenzverfahren. Spezielle Kontrollanweisungen im Programmcode dienen dabei zur Steuerung des ODC² und lösen Kohärenzoperationen aus. Eine erweiterte Cache-Hardware setzt diese Kohärenzoperationen um. Die Kontrollanweisungen werden mit Rücksicht auf die angewendete Synchronisationstechnik platziert und ermöglichen eine Unterscheidung von Programmabschnitten bezüglich potentieller Inkohärenz. So werden Abschnitte, in denen ausschließlich auf private Daten zugegriffen wird, von Abschnitten unterschieden, die Zugriffe auf private und gemeinsame Daten beinhalten. Bei der Durchführung von Speicherzugriffen in privaten Abschnitten verhält sich der ODC² exakt wie ein herkömmlicher Cache ohne Kohärenzprotokoll. Nur in Abschnitten, in denen auch auf gemeinsame Daten zugegriffen wird, werden Kohärenzoperationen angewendet. Damit bietet der ODC² eine temporäre Kohärenz, die nur dann aktiviert wird wenn sie wirklich benötigt wird.

4.1.1 Private-Mode und Shared-Mode

Der ODC² operiert in zwei verschiedenen Modi (*Private-Mode* und *Shared-Mode*), die abhängig davon aktiv sind, ob auf gemeinsame Daten zugegriffen wird. Zu Beginn einer Ausführung befindet sich der ODC² im Private-Mode, welcher ausschließlich für den Zugriff auf private Daten gedacht ist. Dieser Modus entspricht einer vollständigen Inaktivität aller Kohärenzmechanismen, daher darf dabei nicht auf gemeinsame Daten zugegriffen werden. Es besteht generell keine Notwendigkeit für Kohärenzmaßnahmen, da sich in diesem Modus zu keiner Zeit gemeinsame Daten im Cache befinden. Die Funktionalität sowie das Zeitverhalten des ODC² unterscheiden sich in diesem Modus nicht von denen eines inkohärenten Caches.

Ausgelöst durch eine Kontrollanweisung im Programmcode wird der ODC² in den Shared-Mode versetzt. Dieser Modus ist für den Zugriff auf private und gemeinsame Daten gedacht und aktiviert die Kohärenzmechanismen des ODC². Die Dauer der Aktivierung des Shared-Mode ist immer auf eine be-

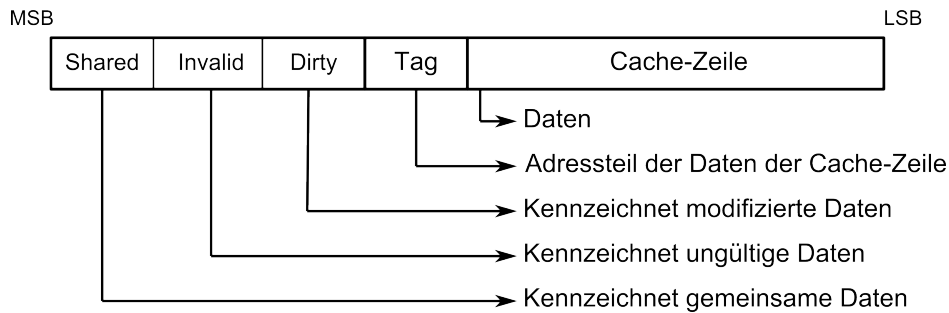


Abbildung 4.1: Aufbau eines ODC² Blockrahmens incl. Shared-Bit

stimmte Programmsequenz beschränkt, die Zugriffe auf gemeinsame Daten umfasst. Der Umfang dieser Sequenz ist nicht beschränkt, orientiert sich aber an der verwendeten Synchronisationstechnik. So kann die Sequenz einen kritischen Abschnitt umfassen, welcher über einen Mutex geschützt ist, oder auch durch Barrieren begrenzt sein.

Der Shared-Mode unterscheidet sich vom Private-Mode in der Handhabung von Daten, die neu in den Cache geladen werden. Tritt ein Cache-Miss bei einem Speicherzugriff während des Shared-Mode auf, so werden die entsprechenden Daten in die Cache-Zeile geladen und dort als „geteilt“ markiert. Die Markierung wird mittels eines zusätzlichen Status-Bits in den Kontrollinformationen des entsprechenden Cache-Blocks gesetzt. Abbildung 4.1 zeigt einen ODC² Blockrahmen mit Shared-Bit. Das Shared-Bit identifiziert den Cache-Block als potentieller Träger von gemeinsamen Daten, auf die Kohärenzmaßnahmen angewendet werden müssen. Generell wird im ODC² nicht zwischen privaten und gemeinsamen Daten unterschieden. Das heißt, der ODC² hat beim Laden eines neuen Datums im Shared-Mode keine Informationen darüber, ob es sich um ein privates oder gemeinsames Datum handelt. Eine solche Information ist in einem herkömmlichen Programmcode auch nicht gegeben. Daher werden alle Daten, die während des Shared-Mode neu in den Cache geladen werden, als „potenziell geteilte Daten“ angesehen und gleichermaßen behandelt.

Dies führt selbstverständlich dazu, dass auch private Daten, die während des Shared-Mode geladen werden, unnötigerweise als „geteilt“ markiert werden. Dies hat keinen negativen Einfluss auf die Funktionalität des ODC², es beeinträchtigt nicht die Kohärenz der Zugriffe. Es kann jedoch eine negative Auswirkung auf die Performanz bei der Ausführung der Applikation haben. Daher bietet der ODC² Möglichkeiten an, eine Markierung privater Daten zu verhindern.

Überprüfung der Zieladresse

Bei der Speicherabbildung in einem Mehrkern-Prozessor werden private und gemeinsame Daten häufig in separaten Speichersegmenten gehalten. Rechenkerns haben dadurch geschützten Zugriff auf ihre privaten Daten, während die gemeinsamen Daten von überall zugreifbar sind. Anhand der Zieladresse eines Speicherzugriffs ist es dann möglich zu erkennen, ob es sich um private oder gemeinsame Daten handelt. Im ODC² wird eine solche Überprüfung der Zieladresse eingesetzt. Dafür muss lediglich zu Beginn der Ausführung ein Adressbereich für gemeinsame Daten angegeben werden. Abhängig vom Resultat der Adressüberprüfung wird eine Cache-Zeile dann entweder als geteilt markiert oder unmarkiert belassen. Dadurch wird vermieden, dass nachfolgende Kohärenzoperationen fälschlicherweise auf private Daten angewendet werden. Eine Überprüfung der Zieladresse hat einen positiven Effekt auf die Laufzeit [103].

Ist die Sequenz von Zugriffen auf geteilte Daten beendet, wird der ODC² mittels einer weiteren Kontrollanweisung im Programmcode wieder in den Private-Mode versetzt. Infolge dieser zweiten Kontrollanweisung und dem Verlassen des Shared-Mode, wird die sogenannte *Restore-Procedure* ausgeführt.

4.1.2 Restore-Procedure

Die Restore-Procedure dient zum einen dem Zweck, sämtliche gemeinsame Daten aus dem Cache zu entfernen, und zum anderen, die Daten im Hauptspeicher zu aktualisieren. Mit Beginn der Restore-Procedure werden nachfolgende Zugriffe des Rechenkerns auf den Cache blockiert, was effektiv zu einer Unterbrechung des Programmflusses für die Dauer der Restore-Procedure führt. Mit Hilfe zweier Operationen werden geteilte Daten im Cache invalidiert und die Konsistenz der Daten im Hauptspeicher wiederhergestellt:

Invalidierung und Zurückschreiben

Zuerst werden alle Cache-Blöcke, die mit dem Shared-Bit markiert sind, als ungültig erklärt bzw. invalidiert. Dieser Vorgang entspricht einem Setzen des Invalid-Bits in jedem Cache-Block, in dem auch das Shared-Bit gesetzt ist. Nach Abschluss der Invalidierung befinden sich nur noch private Daten im Cache, alle gemeinsamen Daten sind nicht mehr zugreifbar.

Auf einige der Cache-Blöcke, die für eine Invalidierung vorgesehen sind, wurde möglicherweise im Verlauf des Shared-Mode schreibend zugegriffen. Sie beinhalten modifizierte Daten, die gegebenenfalls in den Hauptspeicher zurückgeschrieben werden müssen. Wird im Cache eine Write-Through Schreibstrategie eingesetzt, so sind die modifizierten Daten bereits konsistent zum gemeinsamen Speicher. Wird hingegen eine Write-Back Strategie eingesetzt, ist dies

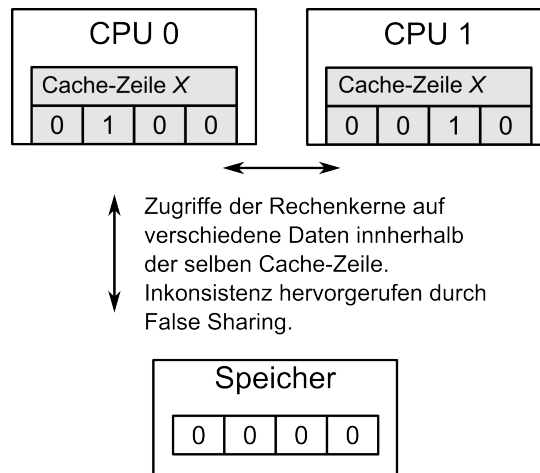


Abbildung 4.2: Veranschaulichung des False Sharing in einem Mehrkern-Prozessor mit privaten Cache-Speichern.

nicht der Fall und die modifizierten Daten müssen im Zuge der Invalidierung in den Hauptspeicher zurückgeschrieben werden. Dies ist ein Vorgang, der auch in einem inkohärenten Cache bei einer Ersetzung oder Invalidierung nötig ist und entspricht keiner speziellen Kohärenzoperation.

Für die Dauer des Zurückschreibens dürfen keine weiteren Zugriffe auf den Cache stattfinden. Die Ausführung des Rechenkern wird unter Umständen für eine längere Zeit unterbrochen. Nach Beendigung der Restore-Procedure befindet sich der Cache in einem ähnlichen Zustand wie zu Beginn des Shared-Mode. Sämtliche gemeinsame Daten, auf die während des Shared-Mode zugegriffen wurde, wurden aus dem Cache entfernt. Der gemeinsame Speicher ist, zumindest in Bezug auf die gemeinsamen Daten dieses Rechenkerns, in einem aktuellen Zustand. Diese Daten können im Folgenden von einem anderen Rechenkern gelesen und modifiziert werden, ohne dass eine Kommunikation mit dem ersten Rechenkern nötig ist.

4.1.3 Schreibstrategie des ODC²

Die Wahl der Schreibstrategie hat besondere Auswirkungen auf das ODC²-Verfahren. Generell unterstützt der ODC² sowohl die Write-Back als auch die Write-Through Schreibstrategie. Jedoch macht die besondere Handhabung der Zugriffe auf gemeinsame Daten im ODC² eine Anwendung der Write-Back Strategie nur eingeschränkt möglich.

In der Regel umfasst der Umfang der Daten, die in einen Cache-Block geladen werden, mehrere Datenwörter. Nur dadurch kann das Prinzip der räumlichen Lokalität genutzt werden (siehe Kapitel 2.1). Zusätzlich zum Datum, auf das zugegriffen werden soll, werden auch einige benachbarte Daten in eine

Cache-Zeile geladen. Dies hat jedoch einen Effekt zur Folge, der *False Sharing* genannt wird. False Sharing bezeichnet das unbeabsichtigte Laden eines Datums in eine Cache-Zeile, welches sich dann zusammen mit dem gewünschten Datum in der Cache-Zeile befindet (siehe Abbildung 4.2). Unbeabsichtigt ist es deshalb, weil es sich um ein Datum handeln kann, auf das der Rechenkern gar nicht zugreifen soll und vielleicht auch nicht darf. Wenngleich vom Rechenkern nicht direkt auf dieses mitgeladene Datum zugegriffen wird, muss die Konsistenz des Datums zu möglichen Kopien in anderen Caches bewahrt werden. In regulären Cache-Kohärenzmechanismen stellt dies kein Problem dar, da die Erhaltung der Kohärenz immer in Bezug auf eine gesamte Cache-Zeile gehandhabt wird. Kohärenznachrichten verwalten immer den Status einer gesamten Cache-Zeile. Dies trifft zwar ebenso auf die Kohärenzoperationen des ODC² zu, jedoch führt hier die fehlende Interaktion zwischen den Cache-Speichern zu einem Problem. Eine für den Einsatz des ODC² korrekt synchronisierte Applikation verhindert konkurrierenden Zugriff von mehreren Prozessorkernen auf ein gemeinsames Datum (mit Ausnahme von Read-Only Zugriffen). Gleichzeitiger Zugriff mehrerer Kerne auf disjunkte Daten ist ausdrücklich gestattet, da ein Kern dabei nicht die Daten eines anderen modifizieren kann. Im Falle von False Sharing, wenn disjunkte Daten unterschiedlicher Kerne in der gleichen Cache-Zeile liegen, ist diese Voraussetzung nicht mehr gegeben. Dann kann es zu inkohärenten Zugriffen kommen, wenn beim Zurückschreiben der Cache-Zeile die Änderungen eines anderen Rechenkerns überschrieben werden.

False Sharing führt nicht zwingend zu einem Fehlerfall. Da jedoch in der Regel nicht vorausgesehen werden kann, ob es in Folge von False Sharing zu fehlerhaften Zugriffen kommt, ist eine Anwendung der Write-Back Strategie für den ODC² nicht problemlos möglich. Durch die Anwendung der Write-Through Schreibstrategie kann die beschriebene Problematik vollständig vermieden werden. Da bei Write-Through Zugriffen die Daten im gemeinsamen Speicher direkt geändert werden, sind diese Daten immer konsistent zu den Daten im Cache. Es ist in der Restore-Procedure nicht mehr nötig, modifizierte Cache-Zeilen zurückzuschreiben und somit können auch keine Änderungen im gemeinsamen Speicher überschrieben werden. Eine Write-Through Strategie hat jedoch zur Folge, dass die Anzahl an Zugriffen auf den gemeinsamen Speicher und damit die Laufzeit möglicherweise erhöht wird. Die Write-Through Strategie wird nicht global für alle Zugriffe angewendet, sondern lediglich temporär für den Zugriff auf gemeinsame Daten während des Shared-Mode. Zugriffe auf private Daten werden weiterhin mittels Write-Back durchgeführt. Damit wird der Overhead an zusätzlichen Speicherzugriffen auf ein notwendiges Minimum reduziert.

Dies hat auch Auswirkungen auf die Dauer der Restore-Procedure. Da die modifizierten gemeinsamen Daten stets konsistent zum Hauptspeicher sind, werden die markierten Cache-Blöcke in der Restore-Procedure lediglich invali-

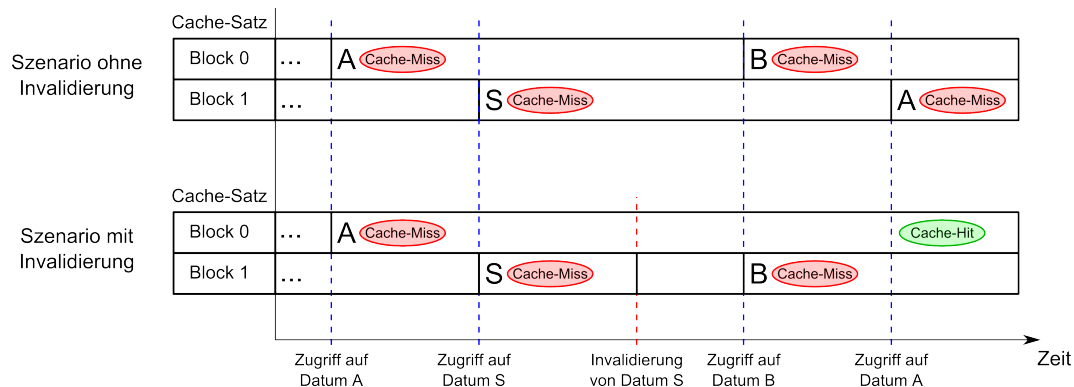


Abbildung 4.3: Veranschaulichung des positiven Effekts bei der Invalidation gemeinsamer Daten im ODC².

diert, ein Zurückschreiben ist nicht mehr nötig. Die potentiell erhöhte Laufzeit, die mit der Write-Through Strategie einhergeht, wird also zum Teil durch das hinfallige Zurückschreiben kompensiert.

4.1.4 Auswirkungen auf die Performanz

Wie bei jedem Cache-Kohärenzverfahren, haben auch die Operationen des ODC² Auswirkungen auf die Performanz der Ausführung. Wie im vorherigen Abschnitt erläutert, erzeugt der Einsatz der Write-Through Schreibstrategie auf gemeinsame Daten eine potentiell erhöhte Anzahl an Zugriffen auf den gemeinsamen Speicher. Je mehr Schreibzugriffe auf gemeinsame Daten in der Applikation stattfinden, desto mehr Zugriffe auf den gemeinsamen Speicher entstehen. Dies ist jedoch nicht die einzige Auswirkung. Die wiederholte Invalidation gemeinsamer Daten in der Restore-Procedure erhöht potentiell auch die Cache-Miss Rate und damit wiederum die Anzahl an Zugriffen auf den gemeinsamen Speicher. Das Ausmaß dieser negativen Auswirkungen hängt sehr stark vom Zugriffsmuster der Applikation ab und lässt sich nicht verallgemeinern. In Kapitel 5 werden diese Auswirkungen anhand einiger Benchmarks näher untersucht.

Die Invalidation gemeinsamer Daten kann allerdings auch einen positiven Effekt auf die Cache-Miss Rate haben. Außerhalb des Shared-Mode greift ein Rechenkern nicht auf gemeinsame Daten zu. Die Invalidation nach Beenden des Shared-Mode kann man also auch als Bereinigung des Caches von unnötigen Daten interpretieren. Der Rechenkern hat so im Private-Mode mehr freie Cache-Blöcke zur Verfügung und kann den Cache insgesamt effizienter nutzen. Konkret kann die Invalidation gemeinsamer Daten eine spätere Ersetzung einer Cache-Zeile mit privaten Daten verhindern. In Abbildung 4.3 ist ein solches Szenario dargestellt. Dieser positive Effekt auf die Performanz kann unter

<pre> - Zugriffe auf private Daten- lock(critical_section) enter_shared_mode - Zugriffe auf private und gemeinsame Daten - exit_shared_mode unlock(critical_section) - Zugriffe auf private Daten - </pre>	<pre> enter_shared_mode - Zugriffe auf private und gemeinsame Daten - exit_shared_mode barrier_wait(barrier_1) enter_shared_mode - Zugriffe auf private und gemeinsame Daten - exit_shared_mode barrier_wait(barrier_2) </pre>
---	--

Abbildung 4.4: Beispiel für die Anwendung von ODC² Kontrollanweisungen in einem kritischen Abschnitt (links) und bei Barrieren (rechts).

Umständen einen Teil der negativen Auswirkungen kompensieren.

4.2 Realisierung

Um ein Programm unter Anwendung des ODC² auszuführen, sind einige Modifikationen des Programmcodes und der Abbildung der Daten im Speicher nötig. Die Modifikationen werden direkt auf den Quellcode der Applikation angewendet und resultieren in einem für den ODC² angepassten Binärcode. Diese Modifikationen verlangen hinreichend genaue Kenntnisse über den Programmfluss, die Synchronisation einzelner Prozesse und die Zugriffsstruktur auf gemeinsame Daten. Sie müssen sorgfältig und korrekt durchgeführt werden, da eine fehlerfreie Erhaltung der Kohärenz von ihnen abhängt. Wenngleich die Modifikation des Programmcodes für diese Arbeit manuell durchgeführt wurde, ist auch eine automatisierte Durchführung nicht ausgeschlossen. Im Folgenden werden die nötigen Änderungen erläutert.

4.2.1 ODC² Kontrollanweisungen

Die Kohärenzoperationen des ODC² werden mit Hilfe zweier Kontrollanweisungen innerhalb des Programmcodes gesteuert. Diese Kontrollanweisungen, hier *enter_shared_mode* und *exit_shared_mode* genannt, aktivieren und deaktivieren den Shared-Mode und müssen, den Zugriffssequenzen auf gemeinsame Daten entsprechend, platziert werden. In einem korrekt parallelisierten Programmcode sind diese Zugriffssequenzen an den im Programm verwendeten Synchronisationsprimitiven ausgerichtet. Sie sind zum Beispiel in kritischen Abschnitten eingebettet oder von Barrieren umschlossen. Die Platzierung der ODC² Kontrollanweisungen kann ebenso an diesen Synchronisationsprimitiven ausgerichtet werden. Entscheidend dabei ist, dass sämtliche Zugriffe auf gemeinsame Daten eingerahmt von diesen Kontrollanweisungen stattfinden.

<pre> lock(addr){ while fetch-and-dec(addr) do end while enter_shared_mode } unlock(addr){ exit_shared_mode fetch-and-inc(addr) } </pre>	<pre> wait(barrier){ exit_shared_mode while !F&D(&barrier!waitlist_lock) do end while cur_runner = F&I(&barrier!runners); if (cur_runner barrier!needed - 1) then Wakes all waiting threads else F&I(&barrier! waitlist_lock) Suspend end if F&I(&barrier! waitlist_lock) enter_shared_mode } </pre>
---	--

Abbildung 4.5: Beispiel für die Einbettung von ODC² Kontrollanweisungen in die Synchronisationsprimitiven für Mutex (links) und Barrieren (rechts).

Zugriffe auf Synchronisationsprimitive hingegen müssen während des Private-Mode stattfinden. Die Abbildung 4.4 zeigt beispielhaft wie die Platzierung der Kontrollanweisungen bei der Verwendung eines kritischen Abschnitts und bei der Verwendung von Barrieren aussehen kann.

4.2.2 Synchronisationstechniken

Der ODC² benötigt für eine fehlerfreie Ausführung eine korrekt synchronisierte parallele Applikation, ist aber nicht auf die Verwendung einer speziellen Synchronisationstechnik festgelegt. Sämtliche Synchronisationstechniken, welche für den Einsatz in harten Echtzeitsystemen geeignet sind (siehe Gerdes [51]), können für den ODC² verwendet werden. Dazu zählen verschieden Arten von Locks wie Spin Locks, Mutex Locks oder Ticket Locks. Ebenso sind sperrfreie Synchronisationstechniken wie Barrieren einsetzbar. Generell ist der ODC² kompatibel zu jeder Synchronisationstechnik, die den Zugriff auf gemeinsame Daten korrekt synchronisiert und dabei folgende Bedingung erfüllt:

Greift ein Rechenkern in einem von Synchronisationsprimitiven definierten Programmabschnitt schreibend auf ein gemeinsames Datum zu, darf in dieser Zeit kein weiterer Rechenkern auf das selbe Datum zugreifen. Bei ausschließlich lesendem Zugriff sind zeitgleiche Zugriffe mehrerer Rechenkerne erlaubt.

Eine Ergänzung des Programmcodes mit ODC² Kontrollanweisungen kann manuell im Quellcode durchgeführt werden. Ebenso ist es möglich, Kontrollanweisungen direkt in Synchronisationsprimitiven einzubinden. So kann beim Betreten und Verlassen eines kritischen Abschnitts über *Lock*- und *Unlock*-Funktionen automatisch der Shared-Mode aktiviert bzw. deaktiviert werden.

Wartefunktionen von Barrieren wiederum können von Kontrollanweisungen umschlossen werden. In Abbildung 4.5 sind zwei Beispiele für eingebundene Kontrollanweisungen, basierend auf Implementierungen von Gerdes [51], dargestellt.

4.2.3 Abbildung privater und gemeinsamer Daten

Um unter Einsatz des Cache-Speichers auf private und geteilte Daten zuzugreifen, müssen diese selbstverständlich auf einem Speicherbereich abgebildet werden, der vom Cache-Speicher adressiert wird. Für den ODC² ist hier eine Einschränkung nötig. Da auf gemeinsame Variablen, welche für die Handhabung der Synchronisation zuständig sind, außerhalb des Shared-Mode zugegriffen wird, unterliegen diese nicht dem „Schutz“ des ODC². Auf diese Variablen darf nur unter Umgehung des Cache-Speichers zugegriffen werden. Daher müssen sie in einem Speicherbereich abgebildet sein, auf den ohne Einsatz des Cachespeichers zugegriffen wird. Diese Einschränkung macht es nötig, alle Definitionen der Synchronisationsprimitiven (Locks, Barrieren etc.) im Programmcode dahingehend zu überarbeiten.

Entscheidend für die korrekte Funktionsweise des ODC² ist es, dass eine Cache-Zeile ausschließlich für private oder gemeinsame Daten verwendet wird. Eine Cache-Zeile darf nicht private und gemeinsame Daten zugleich beinhalten. Andernfalls könnten durch *False Sharing* gemeinsame Daten außerhalb des Shared-Mode in den Cache geladen werden. Ein kohärenter Zugriff darauf könnte nicht mehr gewährleistet werden. Um dies zu verhindern sollten private und gemeinsame Daten in separaten Segmenten des Adressraums abgebildet werden. Ist dies nicht möglich, kann z.B. bei der Definition der Variablen sichergestellt werden, dass alle gemeinsamen Daten zusammen und an der Cache-Zeilengröße ausgerichtet im Speicher liegen.

4.2.4 Hardwareanforderungen

Eine Realisierung des ODC² Verfahrens setzt eine Erweiterung der Cache-Hardware voraus. Dabei ist das Verfahren flexibel in Bezug auf die Architektur in die es eingebettet ist. Gängige Hardware Cache-Kohärenzmechanismen sind in der Regel auf bestimmte Architekturtypen beschränkt. So sind Snooping-Protokolle auf ein Kommunikationsnetz angewiesen, das ein effizientes Broadcasting der Speicherzugriffe aller Teilnehmer ermöglicht. Verzeichnisbasierte Protokolle sind in dieser Hinsicht flexibler, benötigen jedoch zusätzlichen Speicher für die Sicherung der Verzeichnisse. Im Folgenden wird aufgeführt, inwieweit der ODC² verschiedene Architekturkonfigurationen unterstützt:

- **Kommunikationsmedium**

Das Kommunikationsmedium hat keinen Einfluss auf die Kohärenzope-

rationen des ODC². Es kann sowohl ein Bus, eine Crossbar als auch netzwerkbasierte Verbindungsarten verwendet werden. Im Gegensatz zu gängigen Hardware-Protokollen werden keine Kohärenznachrichten über das Kommunikationsmedium versendet. Die Operationen finden für jeden Rechenkern lokal statt. Damit sind die Anforderungen an zusätzlicher Hardware auf den Cache selbst beschränkt.

- **Speicherhierarchie**

Das ODC² Verfahren ist für hart echtzeitfähige Systeme konzipiert, und ist daher für den Einsatz in einem First-Level Cache und einem gemeinsamen Speicher in der darauffolgenden Hierarchieebene konzipiert. Eine Erweiterung der Kohärenzoperationen ist prinzipiell auch auf weitere Cache-Ebenen möglich. Jedoch beeinträchtigen weitere Cache-Ebenen die zeitliche Vorhersagbarkeit von Speicherzugriffen und sind deshalb für harte Echtzeitsysteme keine gute Wahl.

- **Cache-Organisation**

Der ODC² setzt keine spezielle Cache-Organisation voraus. Direkt abgebildete, satz- wie auch voll-assoziative Caches sind möglich. Hinsichtlich einer Cacheanalyse ist eher ein satz-assoziativer Cache vorzuziehen [18]. In Bezug auf die Schreibstrategie ist die in Kapitel 4.1.3 beschriebene Einschränkung zu beachten. Generell kann aber Write-Back wie auch Write-Through verwendet werden. Im Falle von Write-Through können Schreibzugriffe als *Write-Allocate* und als *Non-Write-Allocate* durchgeführt werden. Ebenso kann die Ersetzungsstrategie frei gewählt werden, wenngleich hinsichtlich der gewünschten Echtzeitfähigkeit die Verwendung von *Least-Recently-Used* (LRU) empfohlen wird. Die Ersetzungsstrategie LRU gilt in Hinsicht auf Analysierbarkeit als die beste Wahl [113].

Die hohe Flexibilität des ODC² Verfahrens hinsichtlich der Systemarchitektur macht auch einen Einsatz in heterogenen Systemen problemlos möglich. So können die verwendeten Rechenkerne gänzlich unterschiedliche Mikroarchitekturen und Cache-Konfigurationen aufweisen. Auch ein Zusammenspiel unterschiedlicher Kommunikationsmedien ist denkbar.

ODC² Kontrollinformationen

Basierend auf der gewählten Cache-Architektur ist eine Erweiterung des Kontrollinformationen eines Cache-Blocks nötig. Diese Erweiterung ist relativ gering im Vergleich zu MESI-basierten Kohärenzprotokollen. Zusätzlich zu den in jedem Cache vorhandenen Status-Bits *Valid*, *Dirty/Modified* etc. wird lediglich das Shared-Bit benötigt. Dies entspricht einer Vergrößerung des benötigten Speicherplatzes für Zustandsinformationen im Cache um 1 Bit pro

Cache-Block, was z.B. bei einem 32 kB großen Cache-Speicher und Cache-Zeilen von 16 Byte Länge einem zusätzlichen Bedarf von 256 Byte bzw. 0,8% der Speicherkapazität entspricht.

Cache-Controller

Die Kohärenzoperationen des ODC² (Markierung, Invalidierung etc.) benötigen zusätzliche Logik im Cache-Controller. Die Operationen nutzen zum überwiegenden Teil die bereits vorhandene Funktionalität, so dass der Bedarf an zusätzlicher Hardware gering ist. Die Funktionalität, über eine spezielle Instruktion den kompletten Inhalt des Caches zu invalidieren und wenn nötig zurückzuschreiben, ist in der Regel in einem Cache bereits vorhanden. Die in der Restore-Procedure eingesetzte Invalidierung entspricht einem globalen Setzen des Invalid-Bit in Abhängigkeit des Shared-Bit und kann im Prinzip für alle Cache-Blöcke parallel ausgeführt werden. Das Zurückschreiben modifizierter Cache-Zeilen bei einer Invalidierung ist Teil der Grundfunktionalität eines Caches.

Insgesamt sind die Anforderungen an zusätzlicher Hardware gering im Vergleich zu gängigen Hardware-Kohärenztechniken, welche komplexe Logik für die Übermittlung der Kohärenznachrichten benötigen oder einen hohen Speicherbedarf zur Sicherung von Kohärenzinformationen haben. Mit dem geringen Bedarf an zusätzlicher Hardware ist auch ein geringer Verbrauch an Energie und Chipfläche zu erwarten, was die Eignung des ODC² für eingebettete Echtzeitsysteme unterstreicht.

4.3 Echtzeitfähigkeit

Die in Kapitel 3.3 beschriebenen Auswirkungen von Kohärenznachrichten auf die zeitliche Vorhersagbarkeit treten beim ODC² nicht auf. Der Verzicht auf eine Interaktion der Cache-Speicher zur Erhaltung kohärenter Zugriffe eliminiert die Hauptursachen für unverhältnismäßig erhöhte Zugriffslatenzen. Es bleibt jedoch zu untersuchen, wie sich die im ODC² angewendeten Kohärenzoperationen auf eine Cacheanalyse auswirken. In diesem Kapitel werden sämtliche Operationen des ODC² detailliert auf ihre Auswirkungen auf eine Cacheanalyse hin untersucht und deren zeitliche Vorhersagbarkeit aufgezeigt:

4.3.1 Ausführung der Kontrollanweisungen

Die Kontrollanweisungen zur Aktivierung und Deaktivierung des Shared-Mode können als einfache Instruktionen implementiert werden, die einen Zugriff auf ein Kontrollregister des Cache-Speichers durchführen. Die Latenz $\Delta_{EnterShared}$ für einen Wechsel vom Private-Mode in den Shared-Mode ist mit der Latenz eines Cache-Hits bei einem Schreibzugriff vergleichbar. Der Prozessor kann im

direkten Anschluss an die Ausführung der Kontrollanweisung mit der Ausführung weiterer Instruktionen fortfahren.

$$\Delta_{EnterShared} = \Delta_{CacheCycle} \quad (4.1)$$

Diese Latenz ist konstant und der Zeitpunkt der Ausführung ist von der Cacheanalyse exakt vorhersehbar. Für den Zustandswechsel vom Shared-Mode in den Private-Mode gilt im Grunde die gleiche Latenz. Jedoch folgt auf diesen Zustandswechsel die Restore-Procedure und die Ausführung nachfolgender Instruktionen wird bis zum Ende der Restore-Procedure verzögert. Strenggenommen muss also die Latenz für die Restore-Procedure ($\Delta_{RestoreProcedure}$) mit zur der Latenz für die Kontrollanweisung hinzugerechnet werden:

$$\Delta_{ExitShared} = \Delta_{RestoreProcedure} + \Delta_{CacheCycle} \quad (4.2)$$

Die Auswirkungen und Latenz der Restore-Procedure werden noch separat erläutert.

4.3.2 Markierung der Cache-Zeilen

Die Markierung einer Cache-Zeile mit dem Shared-Bit in den Kontrollinformationen eines Cache-Blocks verursacht keine zusätzliche Latenz. Da jedoch die markierten Cache-Zeilen in der Restore-Procedure invalidiert werden, hat die Markierung einen Einfluss auf den Zustand des Caches und damit auf die Cacheanalyse. Es ist also notwendig die Markierung in die Must-/May-Analyse miteinzubeziehen [109]. Die Cacheanalyse wird um eine Shared-Analyse erweitert, die für jeden Speicherzugriff angibt, ob sich der Cache im Shared-Mode befindet. Die Shared-Analyse muss sowohl Teil der Must- wie auch der May-Analyse sein und impliziert die neuen Zustände für Speicherzugriffe. So wird ein Zugriff als *Always-Shared*, *Sometimes-Shared* oder *Never-Shared* klassifiziert. Es sind Szenarios denkbar, in denen es vom Programmfluss abhängt, ob ein Zugriff im Shared-Mode ausgeführt wird. In diesem Fall wird der Zugriff als *Sometimes-Shared* klassifiziert. Üblicherweise ist aber ist eine sichere Klassifizierung in *Always-Shared* und *Never-Shared* möglich. Abhängig von der Klassifizierung wird dann z.B. ein Write-Back oder Write-Through Schreibzugriff angenommen.

Die Cacheanalyse kann mit Hilfe der Shared-Analyse eine maximale Menge zu markierender Cache-Zeilen ermitteln. Es werden genau die Cache-Zeilen als markiert angenommen, die in Folge eines Zugriffs in den Cache geladen wurden, der als *Always-Shared* oder *Sometimes-Shared* klassifiziert wurde. Somit

ist die Menge von Cache-Zeilen, die von der Cacheanalyse als markiert vorhergesehen werden, eine Obermenge der tatsächlich markierten Cache-Zeilen. Die Vorhersage von zu markierenden Cache-Zeilen kann daher als sicher betrachtet werden.

4.3.3 Restore-Procedure

Die Restore-Procedure wirkt sich zweifach auf die Vorhersagbarkeit der Ausführung aus. Zum einen verursacht ein potientiell zurückschreiben von modifizierten Cache-Zeilen eine zusätzliche Latenz. Zum anderen wird durch die Invalidierungen der Zustand des Caches verändert, was sich auf die Must-/May-Analyse auswirkt.

Die Invalidierung aller markierten Cache-Zeilen kann so implementiert werden, dass sie parallel an allen betroffenen Cache-Zeilen durchgeführt wird (siehe Kapitel 4.2.4). Werden Schreibzugriffe auf markierte Cache-Zeilen mittels Write-Through durchgeführt, so ist es nicht nötig, modifizierte Daten bei der Invalidierung zurückzuschreiben. Es kann davon ausgegangen werden, dass die Invalidierung für alle betroffenen Cache-Zeilen parallel, womöglich in sogar in einem Zyklus, durchführbar ist. Dadurch ist lediglich eine konstante Latenz für die Restore-Procedure einzubeziehen. :

$$\Delta_{RestoreProcedure(Write-Through)} = \Delta_{CacheCycle} \quad (4.3)$$

Ist der Cache-Speicher aus irgendeinem Grund nicht in der Lage, die Invalidierung parallel auszuführen, ist die Latenz der Restore-Procedure dennoch durch die Anzahl markierter Cache-Zeilen begrenzt und vorhersehbar. Bei M markierten Zeilen und einer Latenz für die Invalidierung von einem Zyklus pro Cache-Zeile ergibt sich folgende maximale Latenz:

$$\Delta_{RestoreProcedure(Write-Through)} = M * \Delta_{CacheCycle} \quad (4.4)$$

Kann sichergestellt werden, dass das Auftreten von *False Sharing* bei der Ausführung zu keinem Fehler führen kann, so sind Schreibzugriffe im Shared-Mode mittels Write-Back möglich. Markierte Cache-Zeilen werden dann in der Restore-Procedure in den gemeinsamen Speicher zurückgeschrieben. Dadurch entsteht eine zusätzliche Latenz, die begrenzt ist durch das Produkt der maximalen Latenz $\Delta_{CacheLineWrite}$ eines Schreibzugriffes auf den gemeinsamen Speicher und der Anzahl M an Cache-Zeilen die im Verlauf des Shared-Mode markiert und modifiziert wurden:

$$\Delta_{RestoreProcedure(Write-Back)} = M * \Delta_{CacheLineWrite} \quad (4.5)$$

4.3.4 Vorhersage markierter Cache-Zeilen

Neben diesen direkten Auswirkungen der Kohärenzoperationen auf die Laufzeit treten auch indirekte Auswirkungen auf. Die wiederholten Invalidierungen ändern den Zustand des Cache-Speichers. Wie zuvor angemerkt, ist eine Überabschätzung der als geteilt markierten Cache-Zeilen möglich. Jede Invalidierung einer solchen Cache-Zeile kann zu einem potentiellen Cache-Miss in der weiteren Ausführung führen. Eine Überabschätzung entspricht also einer potentiellen Erhöhung der Cache-Miss Rate und damit der maximalen Ausführungszeit. Das mögliche Ausmaß dieser Überabschätzung hängt von mehreren Faktoren ab.

Findet eine Überprüfung der Zieladresse im Shared-Mode statt, werden nur die Cache-Zeilen als geteilt markiert, deren Daten aus dem Speichersegment der gemeinsamen Daten stammen. Entsprechend der Funktionalität des ODC², wird außerhalb eines Shared-Mode nicht auf diese Daten zugegriffen. Damit sind zu Beginn des Shared-Mode alle Cache-Blöcke unmarkiert und die nachfolgende Markierung ist unabhängig vom aktuellen Zustand des Caches. Sie wird nur von der späteren Zugriffsfolge bestimmt. Vorausgesetzt, es kann für alle Zugriffe eine Zieladresse bestimmt werden, so ist bei der Cacheanalyse gar kein Spielraum für eine Überabschätzung markierter Cache-Zeilen gegeben. Das vorhergesagte Ausmaß der Invalidierung in der Restore-Procedure entspricht exakt dem tatsächlichen Ausmaß. Findet keine Überprüfung der Zieladresse statt, so werden für alle während des Shared-Modus angenommenen Cache-Misses die entsprechenden Cache-Zeilen markiert. Die dabei entstehende Überabschätzung ist durch die Anzahl der von der Cacheanalyse vorhergesagten Cache-Misses begrenzt.

Ist das Ziel eines Speicherzugriffs nicht vorhersehbar (z.B. bei indirekter Adressierung), so muss die Cacheanalyse einen Zugriff auf jeden der möglichen Cache-Blöcke annehmen. Entsprechend steigt die Anzahl potentiell markierter Cache-Blöcke auf alle Cache-Blöcke, die zu diesem Zeitpunkt in den Cache geladen werden können. Die zuvor mit Sicherheit bereits im Cache enthaltenen Blöcke sind davon nicht betroffen. Die betroffene Menge ist also definiert durch die Menge von Cache-Blöcken, die im abstrahierten Zustand C^u einen Alterswert von T besitzen. Ein Zugriff mit unvorhersehbarer Zieladresse ist für eine Cacheanalyse problematisch, da das sichere Wissen über den Zustand des Caches stark verringert wird. Für die Latenz der Restore-Prozedur muss dies keine negative Auswirkung haben. Die Anzahl während des Shared-Mode markierter Cache-Zeilen muss sich nicht zwingend mit der Anzahl an Cache-Zeilen decken, die im Cache-Zustand als geteilt markiert sind. Entscheidend für die Latenz ist die Anzahl der Zugriffe auf markierte Daten während des Shared-Mode. Und diese Anzahl wird auch bei einem Zugriff mit unbekannter Zieladresse lediglich um einem Wert erhöht.

4.4 Bezug zu verwandten Arbeiten

Das Verfahren des On-Demand Coherent Cache steht im Gegensatz zu gängigen Hardware Kohärenzprotokollen, da es auf Interaktion zwischen Caches zur Erhaltung der Kohärenz verzichtet. Cache-Kohärenzverfahren, die eine zeitliche Vorhersagbarkeit der Zugriffe zum Ziel haben, sind nicht sehr verbreitet. Der ODC² greift auf einige bekannte Konzepte der Software-Kohärenz zurück, die aber nie in Zusammenhang mit zeitlicher Vorhersagbarkeit betrachtet wurden.

Die Idee, an definierten Programmpunkten Teile des Cache-Speichers zu invalidieren, wurde bereits unter dem Begriff Selective-Invalidation angewendet. Dabei wird, ähnlich dem ODC², der Programmcode mit Invalidierungsanweisungen erweitert. Diese Erweiterungen beruhen zum Teil auf einer aufwändigen Compileranalyse [141]. Im Ansatz von Sandhu et al. [118] wurden bereits Synchronisationstechniken für die Invalidierung des Cache-Speicher verwendet. Eine Synchronisation der Speicherzugriffe wird dabei nicht über Codesequenzen, sondern über definierte Speichersegmente, sogenannte Shared Regions verwaltet. Dies erfordert jedoch ein verändertes Programmiermodell. Andere Ansätze [12] versuchen, die Menge der Cache-Zeilen, die es zu invalidieren gilt, mittels komplexer Algorithmen zu approximieren. Dabei werden aber Zugriffsinformationen zwischen den Rechenkernen ausgetauscht, die einer statischen Analyse nicht zugänglich ist.

Ein auf Invalidierungen basierendes Cache-Kohärenzverfahren mit dem Ziel zeitlicher Vorhersagbarkeit wurde von Puffitsch et al. [101] vorgestellt. Die Invalidierung wird allerdings auf den gesamten Cache angewendet und setzt ein Write-Through Cache voraus.

Kapitel 5

Evaluation

In Kapitel 4 wurden die Auswirkungen des ODC² auf die Performanz und die zeitliche Vorhersagbarkeit bei der Ausführung paralleler Anwendungen hin beschrieben. Es wurde gezeigt, dass ein möglicher Overhead an Ausführungszeit aufgrund der temporären Write-Through Strategie und der wiederholten Invalidierung gemeinsamer Daten zu erwarten ist. Das Ausmaß dieses Overheads ist begrenzt und für eine statische Laufzeitanalyse vorhersehbar. Es bleibt die Frage, in wieweit dieser Overhead die Anwendbarkeit des ODC² Verfahrens beeinträchtigt. Ein Cache-Kohärenzverfahren ist nur dann sinnvoll einsetzbar, wenn eine signifikante Performanzsteigerung im Vergleich zu einem Zugriff auf gemeinsame Daten ohne Hilfe des Caches erreicht wird. Bei Echtzeitsystemen ist dabei die Worst-Case Performanz ausschlaggebend. In diesem Kapitel sollen daher die konkreten Auswirkungen des ODC² Verfahrens auf die Ausführung parallelisierter Benchmarks evaluiert und mit anderen Kohärenzverfahren verglichen werden.

Ziel dieser Evaluation ist es, zu ergründen, inwieweit der Einsatz des ODC² eine Verbesserung zu aktuell angewendeten Verfahren für kohärenten Zugriff auf gemeinsame Daten darstellt. Im ersten Teil der Evaluation wird der ODC² mit dem MESI Kohärenzprotokoll verglichen. Ziel ist es, die Auswirkungen beider Kohärenzverfahren hinsichtlich der Average-Case Performanz zu untersuchen. Der Schwerpunkt der Evaluation liegt jedoch auf der Anwendbarkeit des ODC² in einem Echtzeitsystem. Dazu wird der ODC² mit anderen echtzeitfähigen Lösungen zur Erhaltung kohärenter Zugriffe verglichen. Zum einen wird auf die Ergebnisse einer simulierten Ausführung auf einer echtzeitfähigen Mehrkern-Architektur zurückgegriffen. Zum anderen wird eine statische WCET-Analyse durchgeführt. Anhand der in der Simulation ermittelten Lauf-

zeit, sowie der errechneten oberen Schranke für die maximale Ausführungszeit, werden die verschiedenen Kohärenzverfahren bei Ausführung paralleler Benchmarks verglichen. Weiterhin wird das Verhalten des Cache-Speichers in den untersuchten Plattformen betrachtet. Es wird die Häufigkeit verschiedener Zugriffe auf den Cache und den gemeinsamen Speicher untersucht. Außerdem stehen die Auswirkungen der untersuchten Cache-Kohärenzverfahren auf den Zustand und den Inhalt des Cache-Speichers im Fokus.

Das Kapitel ist wie folgt unterteilt: Kapitel 5.1 bespricht vorausgehende Studien des ODC². In Kapitel 5.2 werden die Zielarchitektur sowie die einzelnen Plattformen vorgestellt, die in der Evaluation eingesetzt werden. Es wird der Aufbau der echtzeitfähigen Mehrkern-Architektur erläutert, in die der ODC² und die Vergleichstechniken eingebettet sind. Desweiteren werden die angewendeten Evaluationstools vorgestellt. Kapitel 5.4 geht näher auf die parallelen Benchmarks ein, die in der Evaluation ausgeführt werden und in Kapitel 5.5 werden die Ergebnisse der Evaluation diskutiert.

5.1 Vorangehende Studien

Die Evaluation in dieser Arbeit baut auf Untersuchungen des ODC² hinsichtlich der Laufzeit und WCET auf, die zum Teil im Rahmen des *par-MERASA* Projekts durchgeführt wurden. Es wurden Laufzeitsimulationen mit einigen Mikro-Benchmarks auf einem busbasierten System durchgeführt [106][103]. Ein Vergleich mit Implementierungen der MESI und MOESI Cache-Kohärenzprotokolle zeigte, dass der ODC² eine vergleichbar gute Performanz erzielt und dabei eine deutlich reduzierte Auslastung des Kommunikationsbusses erreicht. Spätere Untersuchungen [104] zeigten, dass ein Performanzgewinn auch in einem netzwerkbasierten Kommunikationsmedium erzielt werden kann. Detailliertere Analysen der ODC² Kohärenzoperationen weisen darauf hin, dass der Overhead, der durch die ODC² Kohärenzoperationen entsteht, gering ist.

In [109] wurden erste statische WCET-Analysen mit verschiedenen Mikro-Benchmarks präsentiert. Es konnte eine signifikant verringerte WCET-Abschätzung bei der Anwendung des ODC², im Vergleich zu gängigen Verfahren für den Zugriff auf gemeinsame Daten, festgestellt werden. Der Overhead an zusätzlichen Zugriffen auf den gemeinsamen Speicher hängt dabei sehr vom Zugriffsmuster der Anwendung ab.

Außerdem zeigen vorangehende Fallstudien zur Ausführung der 3D Path Planning Applikation [105][108], dass der ODC² auch bei einer hoch-parallelierten industriellen Applikation einen effizienten sowie zeitlich vorhersagbaren Zugriff auf gemeinsame Daten ermöglicht.

In dieser Evaluation sollen die bisher erzielten Resultate bestätigt und ergänzt werden. Ziel ist eine umfassende Untersuchung der Ausführung der zur

Verfügung stehenden Benchmarks unter Anwendung des ODC² und gängiger Alternativ-Verfahren. Es wird eine vertiefte Analyse der Kohärenzoperationen durchgeführt, um die Vor- und Nachteile der verschiedenen Verfahren zu beleuchten.

5.2 Evaluationsplattform

Das Hauptaugenmerk dieser Evaluation liegt in der Ausführung des ODC² in einer echtzeitfähigen Systemarchitektur. Für den Vergleich des ODC² mit dem MESI-Protokoll wird jedoch eine auf Average-Case Performanz gerichtete Architektur verwendet. Das System ist also nicht im Hinblick auf zeitliche Vorhersagbarkeit aufgebaut. Die Architektur verwendet einen Bus als Kommunikationsmedium und ermöglicht so den Einsatz eines Snooping-Kohärenzverfahrens. Es wird eine Round-Robin Arbitrierung der Buszugriffe angewendet. Auf PowerPC 405 Prozessoren basierende Rechenkerne und ein gemeinsamer Speicher, der Programmcode und Daten zur Verfügung stellt, sind an den Bus angeschlossen. Die Rechenkerne besitzen je einen separaten Instruktions- und Datencache. Diese privaten Cache-Speicher sind jeweils 16 kB groß und verfügt über 512 Cache-Sätze, die 2-fach assoziative Cache-Blöcke zu je 16 Byte enthalten. Die Schreibstrategie des Caches ist standardmäßig Write-Back. Als Kohärenz-Protokoll wendet der Cache den ODC² oder MESI an.

Die weiteren Kohärenzverfahren, die in dieser Evaluation untersucht werden, sind hinsichtlich ihrer zeitlichen Vorhersagbarkeit für den Einsatz in harten Echtzeitsystemen geeignet. Um die Auswirkungen der Verfahren in einem adäquaten Umfeld zu evaluieren, müssen diese in eine ebenso echtzeitfähige Systemarchitektur eingebettet werden. Eine für diese Evaluation geeignete Zielarchitektur muss also, ebenso wie der Cache, ein zeitlich vorhersagbares Verhalten ausweisen und den in Kapitel 2.5.4 angeführten Anforderungen entsprechen.

Daher wird für diese Evaluation eine echtzeitfähige Mehrkern-Prozessorarchitektur eingesetzt, die im Rahmen des FP7 EU-Projekts parMERASA [138] entwickelt wurde. Diese Architektur ist im Hinblick auf eine effiziente Ausführung parallelisierter Applikationen, unter Beachtung der speziellen Anforderungen sicherheitskritischer Echtzeitsysteme, konzipiert [110][136]. Sie erfüllt die Ansprüche an zeitlicher Vorhersagbarkeit beim Zugriff auf einzelne Komponenten wie Netzwerk, Speicher und I/O, sowie die Möglichkeit einer isolierten Betrachtung einzelner Anwendungen im Gesamtsystem (*freedom from interference*). Die parMERASA Architektur stellt ein adäquates Beispiel eines zukünftigen Mehrkern-Prozessors für Echtzeitsysteme dar und ist damit eine geeignete Plattform für diese Evaluation.

Die Evaluationsplattform beinhaltet eine Mehrkern-Architektur, die aus einfachen, auf PowerPC 405 Prozessoren basierenden Rechenkernen besteht,

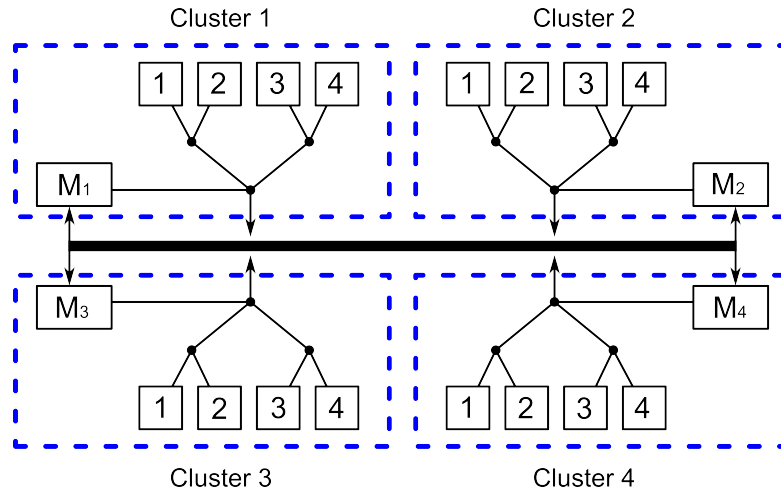


Abbildung 5.1: Abstrahierter Aufbau der parMERASA Architektur, nach [110].

die über ein netzwerkbasirtes Kommunikationsmedium verbunden sind. Die Topologie des Systems ist in Cluster unterteilt. Innerhalb eines Clusters sind die Rechenkerne in einer Baum-Struktur miteinander verknüpft (siehe Abbildung 5.1). Zusätzlich beinhaltet jedes Cluster einen gemeinsamen Speicher. Die Rechenkerne repräsentieren eine RISC Architektur mit einer 5-stufigen Pipeline. Auf Techniken, die die zeitliche Vorhersagbarkeit beeinträchtigen, wie z.B. dynamische Sprungvorhersage, wird verzichtet. Die einzelnen Rechenkerne eines Clusters verfügen über einen privaten L1-Cache und einen privaten Scratchpad-Speicher, und teilen sich einen gemeinsamen On-Chip Speicher, in dem Daten und Instruktionen abgelegt sind. Die Speicherabbildung der Architektur ermöglicht es, über separate Speichersegmente auf private und gemeinsame Daten zuzugreifen. Während der private Cache-Speicher für Programmcode zur Verfügung steht, ermöglicht der Scratchpad Speicher den schnellen Zugriff auf den Programmcode.

In der Evaluation werden unterschiedliche Cache-Organisationen betrachtet. Die Basis-Konfiguration stellt ein 16 kB großer Datencache mit 2-facher Assoziativität und einer Cache-Zeilengröße von 16 Byte dar. Es wird die Least-Recently-Used Ersetzungsstrategie verwendet und die reguläre Schreibstrategie ist Write-Back.

Da bei der Betrachtung der Kohärenz nur der Datencache eine Rolle spielt, wird für den Zugriff auf den Programmcode ein Scratchpad eingesetzt. Das Scratchpad ist für diese Evaluation so konfiguriert, dass es zu Beginn der Ausführung bereits über den vollständigen Programmcode verfügt. Sämtliche Zugriffe auf den Programmcode werden so mit einer minimalen Latenz, analog zu einem Cache-Hit beantwortet. Zugriffe auf den gemeinsamen Speicher zum Laden von Instruktionen finden nicht statt. Dadurch wird erreicht, dass die

Auswirkungen der Datenzugriffe ohne eine Beeinflussung durch Zugriffe auf Instruktionen untersucht werden können.

Die Unterteilung der Prozessorarchitektur in Cluster ermöglicht eine parallele, aber voneinander getrennte Ausführung verschiedener Applikationen, ohne unvorhersehbare gegenseitige Beeinflussung. In dieser Evaluation wird in jedem Durchlauf jeweils nur ein Benchmark ausgeführt, so dass sich der Aufbau des Systems auf einen Cluster beschränkt. Die Größe des Clusters hängt von der Anzahl an der Ausführung beteiligter Rechenkerne ab. Eine Erhöhung der Anzahl der Rechenkerne vergrößert auch die Baumstruktur des Kommunikationsmediums und schlägt sich auf die Zugriffslatenz für den gemeinsamen Speicher nieder. Dies wird für die Ermittlung der maximale Zugriffslatenz bei der statischen Analyse berücksichtigt.

Die verschiedenen Verfahren kohärenten Zugriff auf gemeinsame Daten zu realisieren werden jeweils anhand einer eigenen Instanz der jeweiligen Architektur untersucht. Diese insgesamt 5 Instanzen, im Folgenden Plattformen genannt, werden kurz erläutert:

- **ODC²**

Der in der ODC² Plattform eingesetzte Cache entspricht dem in Kapitel 4 beschriebenen ODC² Kohärenzverfahren. Die Speicherabbildung der Architektur ermöglicht es, anhand einer Überprüfung der Zieladresse eines Zugriffs, eine unnötige Markierung privater Daten während des Shared-Mode zu vermeiden (siehe Kapitel 4.1.1). Um inkohärente Zugriffe in Folge von False Sharing zu unterbinden, wendet der ODC² die Write-Through Schreibstrategie bei Zugriffen auf gemeinsame Daten an (siehe Kapitel 4.1.3). Die übrigen Schreibzugriffe werden mittels Write-Back durchgeführt.

- ***Uncached***

Die *Uncached* Plattform ist die erste Vergleichsplattform für den ODC². Diese Plattform verwendet einen herkömmlichen Cache ohne jeglichen Mechanismus zur Erhaltung der Cache-Kohärenz. Dies macht es notwendig, Zugriffe auf gemeinsame Daten unter Umgehung des Caches zu realisieren. Dazu werden sämtliche gemeinsame Daten auf einen nicht gecachten Speicherbereich abgebildet. Zugriffe auf diese Daten resultieren somit immer in Zugriffen auf den gemeinsamen Speicher. Für private Daten hingegen steht der Datencache ohne Einschränkungen zur Verfügung. Diese Methode ist die einfachste Art kohärente Zugriffe auf gemeinsame Daten zu ermöglichen und wird in nahezu sämtlichen Mehrkern-Prozessoren ermöglicht. Sie bietet die bestmögliche zeitliche Vorhersagbarkeit, da Cache-Kohärenz hier kein Problem darstellt. Aus diesem Grund ist sie oft die erste Wahl für sicherheitskritische Systeme. Zweifelsohne verzich-

tet diese Methode auf den Vorteil kürzerer Zugriffslatenzen, die Cache-Speicher mitbringen.

- **MESI**

Das MESI-Protokoll dient als Vergleichsplattform für die Auswirkungen der Kohärenzoperationen auf die Average-Case Performanz im Bussystem. Es realisiert ein Snooping-Verfahren mit Write-Invalidate Strategie, entsprechend dem Illinois Protokoll [98]. Jeder Rechenkern „belauscht“ die Speicherzugriffe der anderen Rechenkerne und ändert bei Bedarf den Zustand der eigenen Cache-Blöcke (siehe Zustandsdiagramm in Abbildung 2.11). Invalidierungsnachrichten werden eingesetzt um Kopien modifizierter Cache-Zeilen in anderen Cache-Speichern zu invalidieren.

- **Cache Flush**

Die Vergleichsplattform *Cache Flush* stellt eine einfache Variante der Software-Kohärenz dar. Der eingesetzte Cache-Speicher implementiert keine spezielle Kohärenztechnik, es wird lediglich die im Instruktionssatz zur Verfügung stehende Cache-Flush Instruktion verwendet. Zugriffe auf private sowie gemeinsame Daten werden ohne Einschränkungen mit Hilfe des Cache-Speichers durchgeführt. Lediglich auf Synchronisationsvariablen muss unter Umgehung des Caches zugegriffen werden. An Synchronisationspunkten, d.h. Punkten im Programmcode, bei denen eine Synchronisation der parallel arbeitenden Rechenkerne vollzogen wird, wird die Cache-Flush Instruktion ausgeführt. Diese Instruktion löst eine Invalidierung des gesamten Cache-Speichers aus. Bei Ausführung eines korrekt synchronisierten Programms, wird so die Kohärenz nachfolgender Zugriffe sichergestellt. Diese Methode realisiert eine echtzeitfähige Kohärenzlösung, die ohne zusätzliche Logik für die Cache-Hardware auskommt. Sie benötigt lediglich eine Erweiterung des Programmcodes mit korrekt platzierten Aufrufen der Cache-Flush Instruktion. Aktuelle Mehrkern-Prozessoren bieten diese Instruktion als Grundlage für Software-Kohärenzverfahren an. Wie beim ODC² kann auch bei der *Cache Flush* Methode das Problem des *False Sharing* (siehe Kapitel 4.1.3) auftreten. Um dadurch entstehende Inkohärenz zu vermeiden, müssen Schreibzugriffe bei der *Cache Flush* Plattform jederzeit mittels der Write-Through Strategie durchgeführt werden.

- **Magic**

Diese Vergleichsplattform entspricht keiner, in der Realität existierenden Kohärenztechnik. Die *Magic* Plattform stellt eine hypothetische Lösung dar und dient lediglich als Referenz für die anderen Verfahren. Zugriffe auf private und gemeinsame Daten werden mit Hilfe des Caches durchgeführt und sämtliche Zugriffe bleiben kohärent. Lesezugriffe werden im-

	Simulation	WCET-Analyse
Scratchpad	1 Zyklus	1 Zyklus
Cache (Hit)	1 Zyklus	1 Zyklus
Gemeinsamer Speicher	10 Zyklen	10 Zyklen
Netzwerk	<i>simuliert</i>	12 - 86 Zyklen je nach Anzahl beteiligter Rechenkerne

Tabelle 5.1: Übersicht über in der Evaluation verwendete Zugriffslatenzen bei der Simulation und WCET-Analyse.

mer mit den zuletzt geänderten Wert beantwortet und Schreibzugriffe sind augenblicklich für alle Rechenkerne sichtbar. Dabei bedarf es keiner Maßnahme oder Operation um diese Kohärenz zu ermöglichen, sie wird auf „magische“ Weise als gegeben angesehen. Das bedeutet insbesondere, dass sie ohne jeden Overhead oder Einfluss auf die Ausführung ermöglicht wird. Es findet weder eine Interaktion zwischen Rechenkernen statt, noch treten zusätzliche Invalidierungen auf. Das zeitliche Verhalten entspricht exakt dem eines inkohärenten Caches. Sowohl der Simulator als auch das WCET-Analysetool ermöglichen die Implementation eines solchen Cache-Speichers. Die *Magic* Plattform dient damit als Referenz für die theoretisch bestmögliche Ausführungszeit, bzw. als minimale untere Schranke für die zu erreichende Ausführungszeit bei Anwendung einer Cache-Kohärenztechnik. Die Referenz ermöglicht Rückschlüsse über Ausmaß des Overheads der übrigen Kohärenztechniken.

Als zusätzliche Referenzplattform wird eine Konfiguration der Architektur mit einem einzigen Rechenkern eingesetzt. Diese Einkern-Architektur soll als Plattform für eine sequentielle Ausführung dienen. Zwischen privaten und gemeinsamen Daten wird nicht mehr unterschieden, so dass jeder Zugriff mit Hilfe des Cache-Speichers vollzogen wird und das Problem der Cache-Kohärenz entfällt. Diese Plattform wird in der Evaluation nicht gesondert betrachtet sondern dient als Referenz für die Laufzeitverbesserung durch Parallelisierung in den übrigen Plattformen.

5.2.1 Zugriffslatenzen in der Evaluation

Das Resultat der WCET-Analyse einer parallelen Ausführung hängt stark von der Zugriffslatenz auf gemeinsame Ressourcen ab. Gerade in Mehrkern-Architekturen, in denen Zugriffe auf gemeinsame Ressourcen von anderen Rechenkernen verzögert werden können, machen diese Latenzen einen großen Teil der Ausführungszeit aus. Dabei ist die maximale Latenz abhängig von der Zugriffslatenz der Ressource selbst, aber auch von der maximalen Übertragungszeit im Kommunikationsmedium. Diese Größen müssen im Vorhinein für das

Kommunikationsmedium definiert werden. In Tabelle 5.1 sind die Latenzen angegeben, die für den Zugriff auf die in der echtzeitfähigen Evaluationsplattform verwendeten Speicher sowie das Kommunikationsnetz definiert sind.

5.3 Tools der Evaluation

Bei der Durchführung der Evaluation wird auf Tools zurückgegriffen, die eine Laufzeitsimulation und statische Laufzeitanalyse ermöglichen. Diese Tools sind frei verfügbar und wurden im Rahmen des parMERASA Projekts intensiv erweitert und modifiziert. Die Evaluation in dieser Arbeit greift auf diese erweiterten Tools zurück.

5.3.1 Ausführungssimulation

Die Ermittlung der Ausführungszeit der Benchmarks wird mit Hilfe eines Laufzeit-Simulators durchgeführt. Der Simulator basiert auf der Open-Source Plattform SoCLib [131]. SoCLib ist eine Sammlung von Systemmodulen für die Simulation von Multi-Prozessorsystemen und baut auf SystemC auf, einer Erweiterung der Programmiersprache C für das Hardware/Software Co-Design. SoCLib ermöglicht die taktgenaue Simulation eines Mehrkern-Prozessorsystems inklusive Kommunikationsmedium, Speicher, I/O-Module etc. bis hin zum Betriebssystem. Für das parMERASA Projekt wurden verschiedene SoCLib Systemmodule neu- bzw. weiterentwickelt um die im Projekt entwickelte echtzeitfähige Mehrkern-Architektur in die SoCLib Plattform zu integrieren.

Die im Simulator implementierte Prozessorarchitektur wird für die Evaluation so konfiguriert, dass sie jeweils eine der fünf untersuchten Plattformen realisiert. Desweiteren wird die Anzahl der an der parallelen Ausführung beteiligten Rechenkerne variiert und damit die Größe Systems oder des Cluster für jeden Durchgang angepasst. Die Benchmarks werden unmittelbar und ohne den Einsatz eines Betriebssystems ausgeführt. Das Scheduling und die Synchronisation der parallelen Task wird implizit im Programmcode der Benchmarks geregelt.

5.3.2 Statische WCET-Analyse

Für die statische Laufzeitanalyse, wird die OTAWA Toolbox [24][14] eingesetzt. OTAWA ist ein Framework für statische WCET-Analyse und wird im Institute de Recherche en Informatique (IRIT) in Toulouse entwickelt. OTAWA wurde im Rahmen des parMERASA Projekts als statisches Laufzeitanalysetool verwendet und ermöglicht die Abschätzung der WCET bei paralleler Ausführung einer Applikation in einer Mehrkern-Architektur. Neben der maximalen

Ausführungszeit eines Tasks mittels Pipeline- und Cacheanalyse, wird die maximale Wartezeit aller Tasks an Synchronisationspunkten ermittelt. Daraus ergibt sich abschließend eine WCET-Abschätzung für die gesamte parallele Ausführung.

Ein Modell der echtzeitfähigen Mehrkern-Architektur, wie sie auch im Simulator verwendet wird, ist in das OTAWA Framework integriert. Dieses Modell beinhaltet auch die verschiedenen Cache-Typen, die in der Evaluation untersucht werden. Eine Cacheanalyse auf Basis des ODC² bedarf einer Erweiterung des Analysemodells. So ist eine explizite *Shared Analyse* notwendig (siehe Kapitel 4.3), um den zusätzlichen Zustand einer Cache-Zeile (Shared-Zustand) in die Must/May-Analyse miteinzubeziehen (siehe [109]).

Um eine WCET-Analyse der parallelen Benchmarks durchzuführen, benötigt das Analysetool zusätzliche Informationen über den Programmfluss und die angewendeten Synchronisationsprimitiven. Es handelt sich dabei zum einen um sogenannte *Flowfacts*, die den Programmfluss darstellen und Annotationen zur Häufigkeit von Schleifendurchläufen beinhalten. Zum anderen müssen die Synchronisationspunkte im Benchmark explizit definiert und an der Synchronisation beteiligte Tasks angegeben werden. Diese Informationen wurden für die Benchmarks teilweise automatisiert erstellt und gehen als zusätzliche Eingaben in die Analyse ein.

5.4 Benchmarks

Eine Evaluation von Cache-Kohärenz Verfahren setzt den Einsatz von Benchmarks voraus, in denen von mehreren Rechenkernen auf gemeinsame Daten zugegriffen wird. Solche Benchmarks implementieren parallelisierte Algorithmen, die die Bearbeitung eines Problems auf mehrere Rechenkerne aufteilen. Mittels Synchronisationsprimitiven wie Mutex oder Barrieren wird das korrekte Zusammenspiel der Rechenkerne sichergestellt. Einer Performanzsteigerung durch die parallelisierte Ausführung steht der Overhead entgegen, der beim Zugriff auf gemeinsame Ressourcen entsteht. Die Anwendbarkeit eines Cache-Kohärenzverfahrens lässt sich daher auch dadurch beurteilen, in wieweit er eine Verbesserung zu einer sequentiellen Ausführung ermöglicht.

In der Evaluation werden 5 parallele Mikro-Benchmarks sowie eine parallele Applikations-Benchmark eingesetzt. Die Mikro-Benchmarks enthalten parallelisierte Algorithmen aus der Signal- und Bildverarbeitung, während die Applikations-Benchmark aus dem Bereich der Avionik stammt. In der Avionik werden besonders hohe Anforderungen an die logische und zeitliche Korrektheit bei der Ausführung von Applikationen gestellt. So verlangt die Norm DO-178B, die Richtlinien zur Zertifizierung von Avionik-Software angibt, eine WCET-Analyse für neu entwickelte Applikationen. Die Einführung des *Integrated Modular Avionics* (IMA) Konzepts, das es erlaubt mehrere Funkionali-

Benchmark	Code	Private/ Gemeinsame Daten	Synchronisation	Parallelität
Matrix	~19 kB	~1 kB / ~32 kB	Mutex	Daten
FFT	~32 kB	~1 kB / ~16 kB	Mutex/Barrieren	Daten
Dijkstra	~21 kB	~1 kB / ~41 kB	Barrieren	Daten
Smoothing	~27 kB	~1 kB / ~22 kB	Barrieren	Daten
Edges	~43 kB	~1 kB / ~22 kB	Barrieren	Daten
3DPP	~62 kB	~1 kB / ~28 kB	Barrieren	Daten & Task

Tabelle 5.2: Details zu den in der Evaluation verwendeten parallelen Benchmarks.

täten auf einer Recheneinheit zu vereinen, öffnete die Tür für den Einsatz von Mehr(kern)-Prozessorsystemen. Für die aktuelle Forschung im Bereich harter Echtzeitsysteme stellen Benchmarks aus dem Bereich der Avionik daher eine ideale Plattform dar [6][95][64][111]. Im Folgenden werden die Benchmarks kurz beschrieben:

- **Matrix Multiplikation (Matrix)**
Matrix [75] implementiert die Multiplikation zweier Matrizen mit 40×76 Integer Elementen. Die Multiplikation wird von allen beteiligten Rechenkernen parallel ausgeführt, wobei ein Kern jeweils eine Zeile der Zielmatrix exklusiv berechnet.
- **Fast Fourier Transformation (FFT)**
FFT [75] implementiert die Anwendung der Fast Fourier Transformation auf ein Array von 512 Double-Precision Gleitkommawerten. Die Berechnung wird parallelisiert auf disjunkte Bereiche der Arrays durchgeführt und über Mutex synchronisiert.
- **Dijkstra (Dijkstra)**
Dijkstra [67] berechnet den kürzesten Pfad zwischen zwei Knoten in einem gewichteten Netzwerk aus insgesamt 100 Knoten. Die Berechnung wird auf den beteiligten Rechenkernen parallel ausgeführt und über Barrieren synchronisiert.
- **SUSAN Smoothing (Smoothing)**
Als Teil der SUSAN Kollektion für Bilderkennungsalgorithmen [67] ist Smoothing verantwortlich für Rauschunterdrückung eines Bildsignals. Eine 30×38 Pixel große Bilddatei wird mittels des *gradient inverse weighted* Verfahrens gefiltert.
- **SUSAN Edges (Edges)**
Edges ist ebenso Teil der SUSAN Kollektion und realisiert die Erkennung und Markierung von Kanten im Bild.
- **3D Path Planing (3DPP)**
3D Path Planing ist die parallelisierte Implementation einer Applikation

aus dem Bereich der Navigation unbemannter Flugobjekte. Der Benchmark wurde von Honeywell International im Rahmen des parMERASA Projekts erstellt und dient dort als Anwendungsbeispiel für die parallelisierte Ausführung industrieller Applikationen. 3DPP errechnet einen möglichen Pfad eines Flugobjekts, hin zu einer Reihe von Zielpunkten unter Berücksichtigung von Hindernissen, die auf dem Weg liegen. Die Berechnung arbeitet auf Grundlage einer 3D Umgebung und erzeugt Angaben bezüglich erforderlicher Richtung und Geschwindigkeit. Der Algorithmus ist in Form einer Pipeline realisiert und bietet daher Möglichkeiten für Task-Parallelität. Zusätzlich kann die 3D Umgebung in Segmente aufgeteilt werden und mittels Daten-Parallelität von mehreren Rechenkernen simultan bearbeitet werden. Barrieren sorgen für die Synchronisation der einzelnen Berechnungsphasen. Detailliertere Informationen zu 3DPP sind bei Jahr et al. [68] zu finden.

Sämtliche Benchmarks sind mit den zusätzlichen Instruktionen ausgestattet, die notwendig sind um mit der *Cache Flush* und der ODC² Plattform ausgeführt zu werden. Die damit einhergehende Erhöhung der Codegröße ist für beide Kohärenztechniken gering. Die Ergänzung um *Cache Flush* Anweisungen vergrößert den Code der Benchmarks um durchschnittlich 0,07% und der ODC² bringt im Durchschnitt 0,29% mehr Code mit sich.

Zusätzlich zu den parallelisierten Implementationen werden die Benchmarks jeweils einmal in einer sequentiellen Variante auf einem Rechenkern ausgeführt. Diese Ausführung dient als Referenz für die Effizienz der Parallelisierung beim Einsatz der verschiedenen Kohärenztechniken. Laufzeitwerte der untersuchten Plattformen werden in der Auswertung immer relativ zur Laufzeit der sequentiellen Ausführung angegeben.

5.5 Auswertung

Wenngleich der ODC² für den Einsatz in harten Echtzeitsystemen konzipiert ist, ist es dennoch von Interesse, wie das Verfahren im Vergleich zu einem auf Average-Case Performanz orientierten Kohärenzprotokoll abschneidet. Der ODC² wird hierfür mit dem MESI-Protokoll verglichen. Als zusätzliche Referenz dient ein Zugriff auf gemeinsame Daten unter Umgehung des Cache-Speichers. In Abbildung 5.2 ist die simulierte Laufzeit einiger Mikro-Benchmarks und des 3D Path Planing Benchmarks bei paralleler Ausführung mit 4 Rechenkernen dargestellt. Die Werte sind jeweils auf die Laufzeit normiert, die bei einer sequentiellen Ausführung der Benchmarks auftritt. Für alle untersuchten Benchmarks erreicht die *Uncached* Plattform die jeweils höchste, und die *MESI* Plattform die niedrigste Ausführungszeit. Die Ausführungszeit

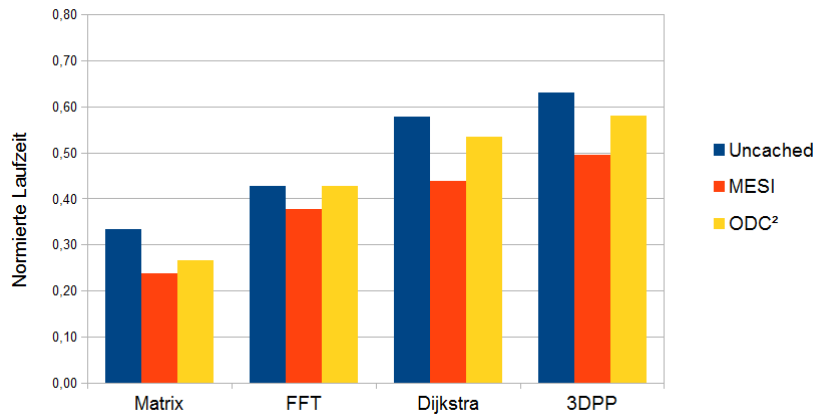


Abbildung 5.2: Auf sequentielle Ausführung normierte Laufzeit bei paralleler Ausführung mit 4 Rechenkernen im Bussystem.

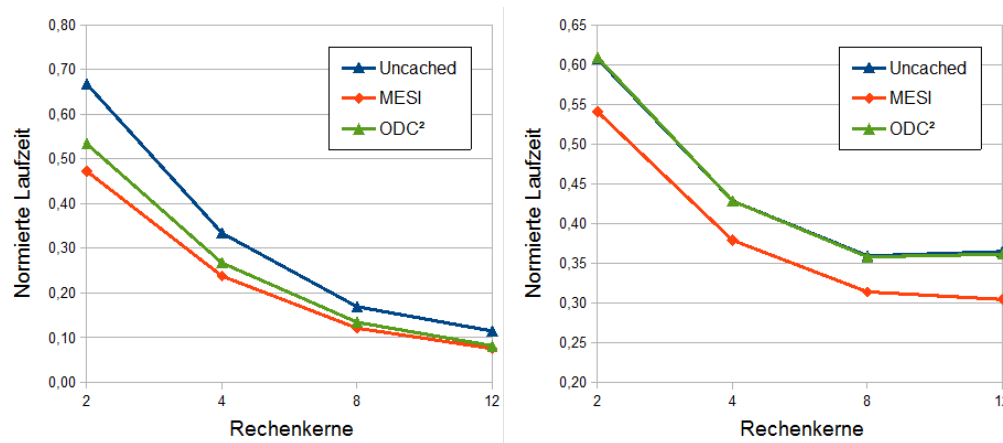


Abbildung 5.3: Auf sequentielle Ausführung normierte Skalierung der Laufzeit bei Ausführung von Matrix und FFT im Bussystem.

des ODC² liegt jeweils zwischen der Ausführungszeit der Plattformen *Uncached* und *MESI*.

Die verwendeten Benchmarks beinhalten verschiedene Zugriffsmuster auf gemeinsame Daten. Je nach Benchmark schneidet der ODC² im Vergleich zu den beiden anderen Plattformen unterschiedlich gut ab. Der ODC² erreicht eine Verbesserung der Laufzeit um bis zu 20,0% verglichen mit *Uncached*. Zugleich ist die Laufzeit jedoch um bis zu 21,7% höher als bei Ausführung mit dem MESI Protokoll.

Der Unterschied in der Performanz zwischen ODC² und *MESI* wird besonders beim Vergleich der Resultate der Benchmarks Matrix und FFT deut-

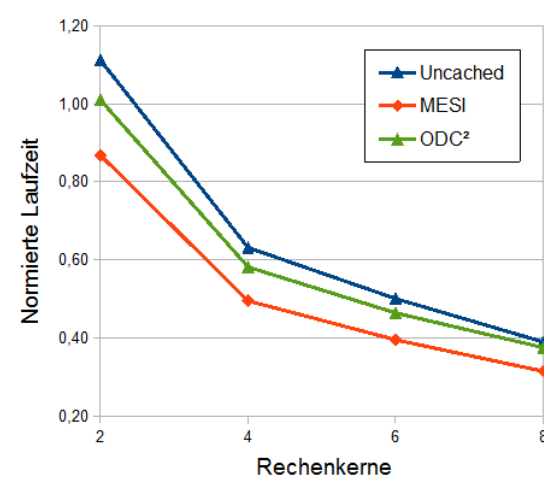


Abbildung 5.4: Auf sequentielle Ausführung normierte Skalierung der Laufzeit bei Ausführung von 3DPP im Bussystem.

lich. Während sich der ODC² bei Matrix mit steigender Anzahl beteiligter Rechenkerne zunehmend der Ausführungszeit von MESI annähert (siehe Abbildung 5.3, links), entspricht die Ausführungszeit bei FFT in etwa der Ausführungszeit von *Uncached* (siehe Abbildung 5.3, rechts). Der Einfluss der Zugriffsmuster in den einzelnen Benchmarks auf die Performanz des ODC² wird im weiteren Verlauf der Evaluierung noch detaillierter betrachtet. Für den industriellen Benchmark 3DPP erreicht der ODC² ebenfalls eine signifikante Verbesserung der Performanz im Vergleich mit *Uncached*, die jedoch stets von der MESI Plattform übertroffen wird (siehe Abbildung 5.4).

Die Resultate lassen darauf schließen, dass beim Einsatz des ODC² Kohärenzverfahrens in einem Bussystem eine signifikante Performanzsteigerung, im Vergleich zu einer Ausführung ohne Nutzung des Caches für gemeinsame Daten, möglich ist. Der erzielte Performanzgewinn liegt jedoch unter dem eines MESI Protokolls. Ein solches, auf Average-Case Performanz optimiertes Kohärenzprotokoll, bleibt die bevorzugte Wahl für Systeme, in denen zeitliche Vorhersagbarkeit keine Rolle spielt.

Für harte Echtzeitsysteme sind Kohärenzprotokolle wie das MESI-Protokoll nicht anwendbar. Die Sicherstellung kohärenter Zugriffe auf gemeinsame Daten darf eine WCET-Abschätzung mittels statischer Laufzeitanalyse nicht zu sehr beeinträchtigen. Für die weitere Evaluation wird der ODC² als Teil einer echtzeitfähigen Architektur untersucht und mit echtzeitfähigen Kohärenzlösungen verglichen.

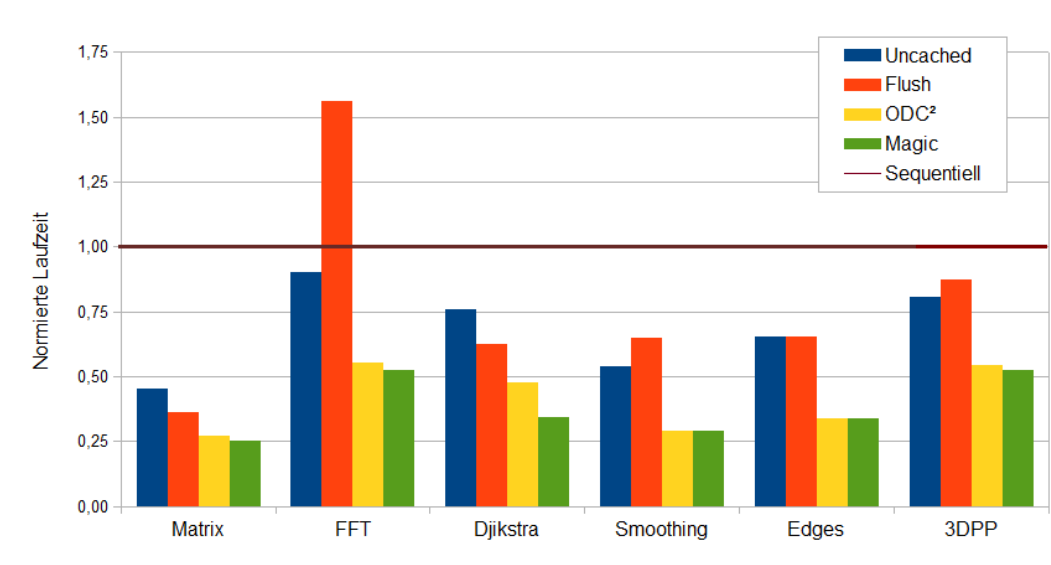


Abbildung 5.5: Auf sequentielle Ausführung normierte Laufzeit bei paralleler Ausführung mit 4 Rechenkernen.

5.5.1 Performanz im Echtzeitsystem

Die gemessene Laufzeit der Simulation sowie die WCET-Abschätzung fallen für die verwendeten Benchmarks je nach untersuchter Plattform sehr unterschiedlich aus. In Abbildung 5.5 ist die simulierte Laufzeit bei einer parallelen Ausführung mit 4 Rechenkernen dargestellt. Die Werte sind jeweils auf die Laufzeit normiert, die bei einer sequentiellen Ausführung mit einem Rechenkern auftritt. Die höchste Ausführungszeit wird je nach Benchmark entweder mit der *Uncached* Plattform oder der *Cache Flush* Plattform erreicht. Der ODC² hingegen erreicht für jeden Benchmark eine Verringerung der Laufzeit verglichen mit der *Uncached* und der *Cache Flush* Plattform. Die Anwendung des ODC² Verfahrens ermöglicht in dieser Simulation eine Reduktion der Laufzeit je nach Benchmark von 32,7% - 48,0% in Bezug zu *Uncached* und 23,7% - 64,5% bezogen auf *Cache Flush*. Bei Einsatz des ODC² ist also eine signifikant höhere Performanz erkennbar.

Generell zeigen alle Plattformen eine Beschleunigung der Ausführung durch die Parallelisierung mit 4 Rechenkernen. Einzige Ausnahme bildet die *Cache Flush* Plattform bei der Anwendung des *FFT* Benchmarks. Hier ist die Laufzeit trotz Parallelisierung um 56,0% höher als bei der sequentiellen Ausführung. Die Effizienz der Parallelisierung ist für alle Plattformen beim Benchmark *Matrix* am höchsten. Der ODC² erzielt (bei Ausführung mit 4 Rechenkernen) eine Beschleunigung um den Faktor 3,64, während bei *Uncached* Faktor 2,2 und bei *Cache Flush* Faktor 2,76 erreicht wird. Doch auch beim industriellen

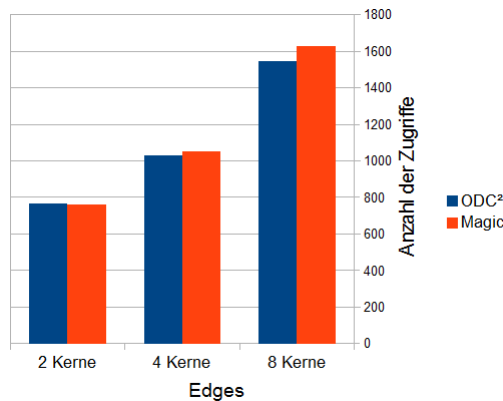


Abbildung 5.6: Anzahl von Cache-Misses bei der Ausführung von Edges mit ODC² und *Magic*.

Benchmark 3DPP wird immerhin noch ein Faktor von 1,84 (ODC²) bzw. 1,24 (*Uncached*) und 1,14 (*Cache Flush*) erreicht.

Die *Magic* Plattform, als Referenz für die theoretisch bestmögliche Laufzeit, schneidet wie zu erwarten am besten ab. Der ODC² erzielt eine Laufzeit, die im Höchstfall 38,6% höher ist als die der *Magic* Plattform. Auf den ersten Blick ungewöhnlich erscheint das Resultat des Benchmarks Edges. Hier schneidet der ODC² minimal besser ab als die *Magic* Plattform (Verringerung um 0,2%). Dies liegt zum einem am Benchmark selbst, der ein für den ODC² sehr vorteilhaftes Zugriffsmuster auf gemeinsame Daten aufweist. Die entscheidende Ursache ist jedoch im positiven Effekt der Invalidierung zu suchen, der bei diesem Benchmark Früchte trägt. Bereinigt von gemeinsamen Daten kann der Cache außerhalb des Shared-Mode effizienter auf private Dateien zugreifen (siehe Kapitel 4.1.4). So treten bei der Ausführung dieses Benchmarks mit steigender Anzahl an Rechenkernen auch weniger Cache-Misses auf als bei der *Magic* Plattform, wie in Abbildung 5.6 dargestellt.

Die Resultate der Laufzeitsimulation bei einer parallelen Ausführung mit 2 und 8 Rechenkernen sind vergleichbar zu den Resultaten der Ausführung mit 4 Rechenkernen. Lediglich bei *Cache Flush* ist für einige Benchmarks eine deutlich schlechtere Skalierung der Laufzeit mit steigenden Anzahl an Rechenkernen zu erkennen. Die entsprechenden Abbildungen sind in Anhang 1 zu finden. Die Skalierbarkeit der verschiedenen Kohärenzverfahren wird in Kapitel 5.5.3 detaillierter betrachtet.

Die Ermittlung einer oberen Schranke für die WCET mit Hilfe der statischen Laufzeitanalyse kommt zu einem vergleichbaren Ergebnis wie die Laufzeitsimulation. Abbildung 5.7 zeigt die WCET-Abschätzung bei einer parallelen Ausführung mit 4 Rechenkernen. Auch hier sind die Werte wieder auf die WCET-Abschätzung einer sequentiellen Ausführung normiert. Auf den ersten

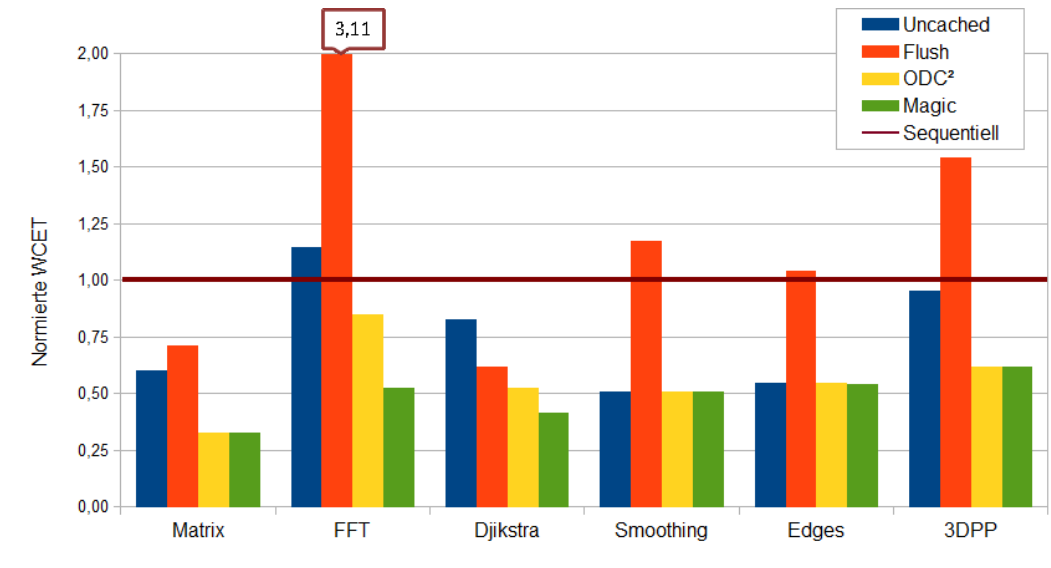


Abbildung 5.7: Auf sequentielle Ausführung normierte $WCET_{est}$ bei paralleler Ausführung mit 4 Rechenkernen.

Blick fällt auf, dass die *Cache Flush* Plattform deutlich schlechter abschneidet, als die übrigen Plattformen. Auch in Relation zur sequentiellen Ausführung sind deutlich höhere Werte erkennbar, in den meisten Fällen werden die Werte der sequentiellen Ausführung sogar übertroffen. Die ungewöhnlich hohe $WCET_{est}$ der *Cache Flush* Plattform ist mit der kontinuierlichen Write-Through Schreibstrategie zu erklären, die zu einer stark erhöhten Anzahl an Zugriffen auf den gemeinsamen Speicher führt. Da solche Zugriffe eine hohe maximale Latenz haben, fallen sie bei der Ermittlung der $WCET_{est}$ besonders ins Gewicht. Dieser Umstand wird in Kapitel 5.5.3 noch genauer beleuchtet.

Die ODC² Plattform erreicht auch bei der statischen Analyse in der Regel eine verbesserte Laufzeit gegenüber der *Uncached* Plattform, wenngleich die Differenz hier geringer ausfällt. Je nach Benchmark wird eine Verringerung der $WCET_{est}$ von 0,2% - 45,2% erreicht (Ausführung mit 4 Rechenkernen). Diese ausgeprägten Unterschiede in den Werten der einzelnen Mikro-Benchmarks sind direkt auf die dort angewendeten Algorithmen zurückzuführen. Hier wird deutlich wie sich verschiedenen Zugriffsmuster auf gemeinsame Daten auf die Worst-Case Performanz der einzelnen Plattformen auswirken. Darauf wird im Folgenden noch detaillierter eingegangen. Für den industriellen Benchmark 3DPP erzielt der ODC² eine Verringerung der $WCET_{est}$ von 34,9% im Vergleich zu *Uncached* und 59,8% im Vergleich zu *Cache Flush*. Die *Magic* Plattform schneidet in diesem Fall minimal schlechter ab als der ODC² mit einer um 0,3% höheren $WCET_{est}$. Auch hier produziert die *Magic* Plattform eine höhere Anzahl an Cache-Misses als der ODC², bei der Ausführung mit 4

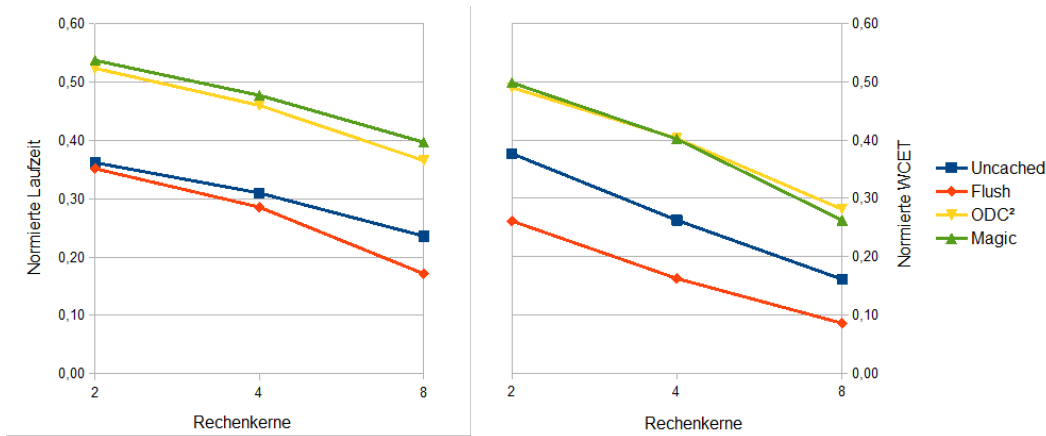


Abbildung 5.8: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von 3DPP.

Rechenkernen sind es sogar 57,4% mehr.

Die WCET-Analyse bei paralleler Ausführung mit 2 und 8 Rechenkernen kommt zu Ergebnissen, die vergleichbar mit der Auswertung der Ausführung mit 4 Rechenkernen sind. Abweichungen resultieren aus der unterschiedlichen Skalierung der Resultate bei einigen Benchmarks. Die entsprechenden Abbildungen sind in Anhang 2 zu finden.

Die angewendete Kohärenztechnik wirkt sich auch auf die Skalierbarkeit der Parallelisierung aus. Abbildung 5.8 illustriert die Effizienz der Parallelisierung bei der Ausführung des Benchmarks 3DPP. Die Werte sind normiert auf die theoretisch mögliche Laufzeitverbesserung ausgehend vom Resultat der sequentiellen Ausführung. D.h. es wird davon ausgegangen, dass die Laufzeit sich antiproportional zur Anzahl der Rechenkerne verändert. Ein Wert von 1 bei einer Ausführung mit 2 Rechenkernen entspricht der Hälfte der Laufzeit der sequentiellen Ausführung. So erzielt z.B. die *Cache Flush* Plattform einen Wert von 0,35 bei der Simulation mit 2 Rechenkernen und erreicht somit 35% der theoretisch optimalen Beschleunigung durch die Parallelisierung. Es ist zu erkennen, dass mit dem ODC² für alle Rechenkerne eine deutlich höhere Effizienz bei der Parallelisierung möglich ist als mit *Uncached* und *Cache Flush*. Alle Plattformen skalieren dabei ähnlich gut mit steigender Anzahl an Rechenkernen.

Die deutlich schlechteren Resultat der *Cache Flush* Plattform im Vergleich zum ODC² sind zum einen durch die Write-Through Schreibstrategie zu erklären, die bei der *Cache Flush* Plattform über die gesamte Laufzeit hin angewendet wird. Doch dies ist nicht die einzige Ursache, da der ODC² während des Shared-Mode ebenfalls mit Write-Through arbeitet. Abbildung 5.9 zeigt den relativen Anteil von Private-Mode und Shared Mode an Gesamtlaufzeit

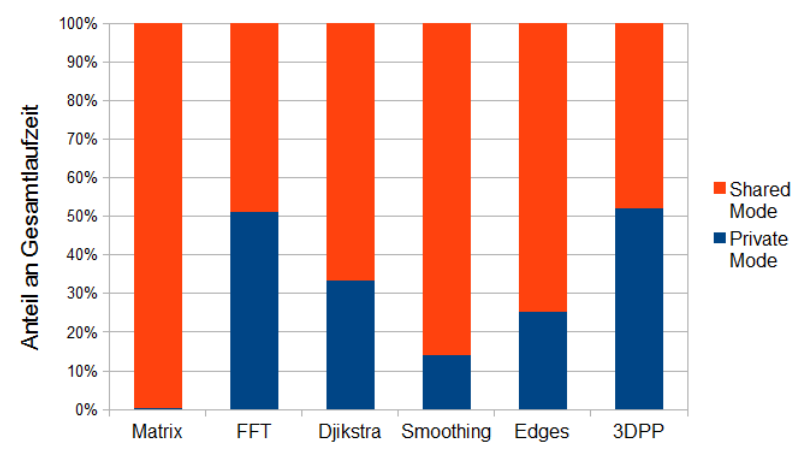


Abbildung 5.9: Relativer Anteil von Private-Mode und Shared Mode an der Gesamtlauzeit bei Ausführung mit 4 Rechenkernen.

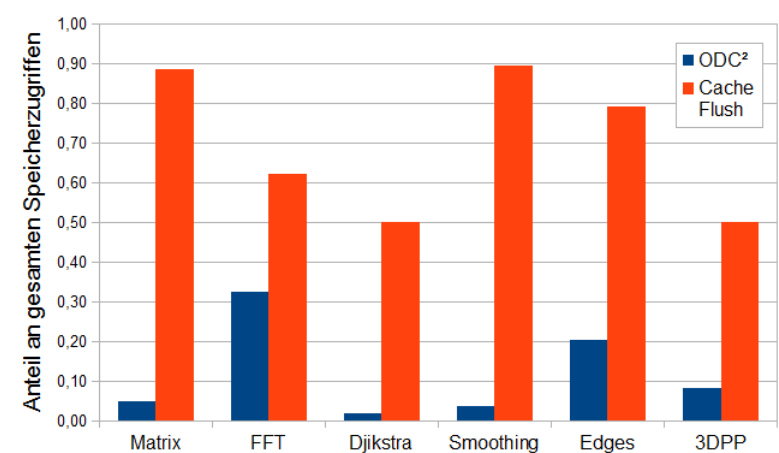


Abbildung 5.10: Relativer Anteil von Write-Through Schreibzugriffen an den gesamten Speicherzugriffen bei Ausführung mit 4 Rechenkernen.

bei paralleler Ausführung mit 4 Rechenkernen. Der Anteil der Ausführungszeit, der im Shared Mode und damit unter Anwendung der Write-Through Strategie verbraucht wird, beträgt dabei mindestens 48%, bei Matrix sind es sogar fast 100%. Ausschlaggebend ist jedoch, dass der ODC² im Shared-Mode nur Schreibzugriffe auf gemeinsame Daten mit Write-Through durchführt. Private Daten werden weiterhin mittels Write-Back beschrieben. Der Anteil von Write-Through Schreibzugriffen an den gesamten Speicherzugriffen ist daher bei ODC² signifikant geringer als bei *Cache Flush*. In Abbildung 5.10 wird dies für die Ausführung mit 4 Rechenkernen dargestellt. Es ist zu erkennen, dass, im Gegensatz zum ODC², Write-Through Schreibzugriffe bei der *Cache Flush* Plattform meist den überwiegenden Anteil an allen Speicherzugriffen haben. Das bedeutet nicht, dass generell mehr Schreibzugriffe ausgeführt werden als Lesezugriffe. Bei beiden Plattformen werden Leseanfragen zumeist direkt im Cache beantwortet und lösen nur im Falle eines Cache-Misses einen Speicherzugriff aus.

5.5.2 Einfluss der Kohärenzoperationen

Im Folgenden wird das Ausmaß der Invalidierung von Cache-Zeilen bei den Kohärenzoperationen des ODC² sowie der *Cache Flush* Plattform genauer untersucht. Die Verfahren beider Plattformen haben die Gemeinsamkeit, dass zu Synchronisationspunkten Daten aus dem Cache invalidiert werden. Während die *Cache Flush* Plattform den gesamten Cache invalidiert, beschränkt sich der ODC² darauf, nur einen Teil der Daten aus dem Cache zu entfernen. Dieser Unterschied ist eine der Ursachen für die Differenzen in Laufzeit und $WCET_{est}$ beider Plattformen. In Abbildung 5.11 ist ein Histogramm des Ausmaßes der Invalidierung während einer Restore-Procedure bei der Ausführung der Benchmarks mit 4 Rechenkernen dargestellt. Die Werte stellen die relative Häufigkeit von Invalidierungen dar, bezogen auf die Menge der Invalidierten Cache-Zeilen dar. So wird z.B. bei der Ausführung von Matrix, bei 53% aller Invalidierungen nur bis zu 5% des Cache-Speichers invalidiert. Bei den übrigen 47% sind zwischen 36-40% des Caches betroffen. Dieser Benchmark bildet jedoch die Ausnahme, in den anderen Benchmarks beschränken sich mindestens 61% der Invalidierungen auf ein Ausmaß von bis zu 5% des Caches. Bei der Simulation des industriellen Benchmarks 3DPP wird im Durchschnitt ungefähr 5,2% des Caches invalidiert.

Diese Resultate stehen in einem deutlichen Kontrast zu dem Ausmaß der Invalidierungen bei der *Cache Flush* Plattform. In Abbildung 5.12 sind die gleichen Daten für die *Cache Flush* Plattform zu sehen. Hierbei wird bei der Ausführung von Matrix in 90% aller Invalidierungen ein Umfang von mindestens 36% des Caches für ungültig erklärt. Auch für die anderen Benchmarks ist das Ausmaß der Invalidierung deutlich höher. Bei der Ausführung von 3DPP

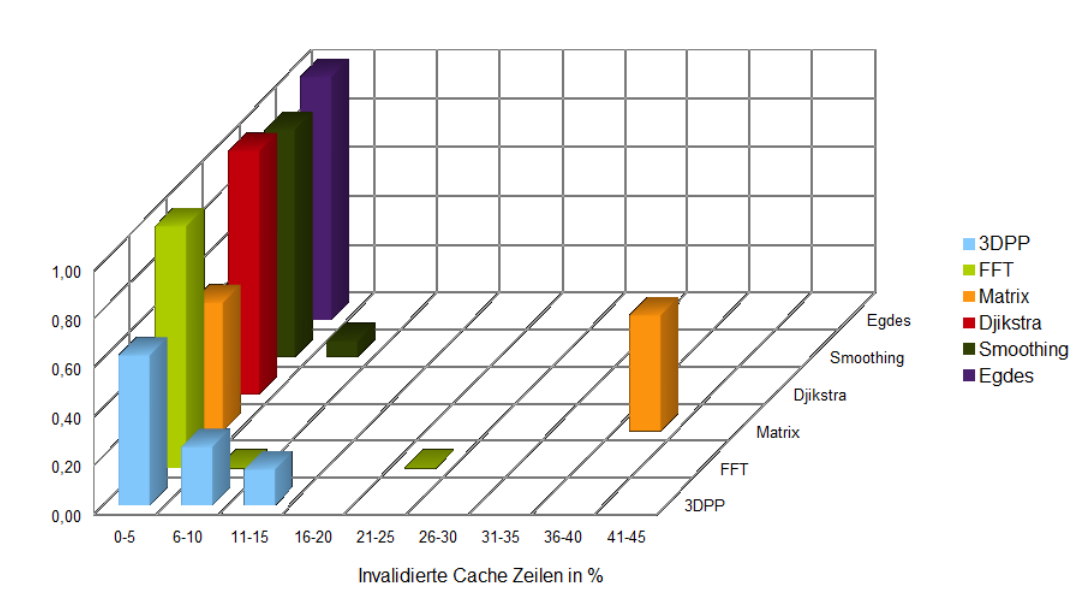


Abbildung 5.11: Histogramm des Ausmaßes der Invalidation während der Restore-Procedure im ODC² bei Ausführung mit 4 Rechenkernen.

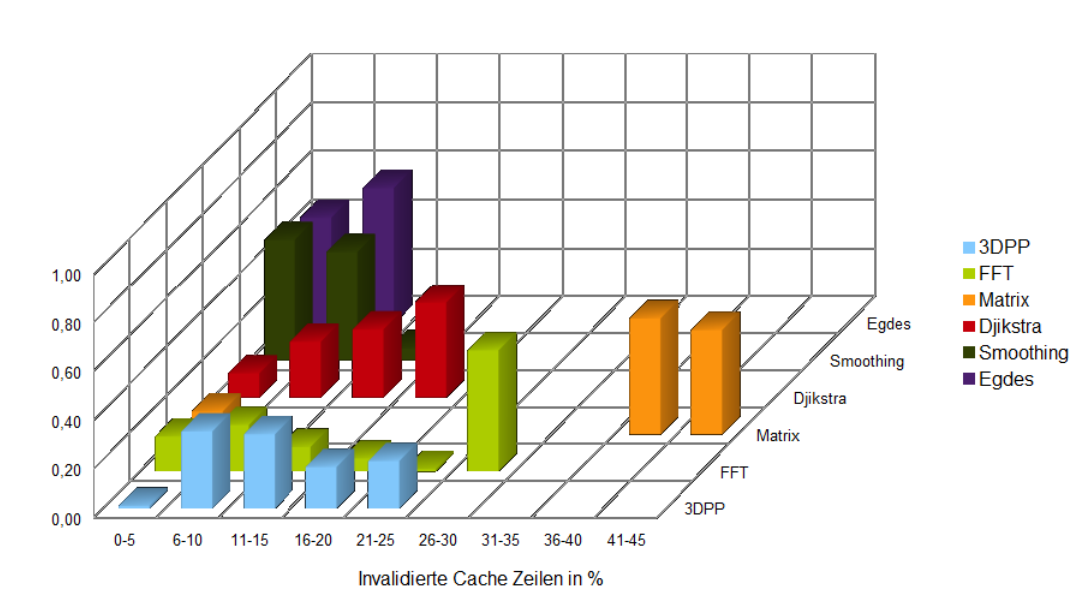


Abbildung 5.12: Histogramm des Ausmaßes der Invalidation an Synchronisationspunkten mit *Cache Flush* bei Ausführung mit 4 Rechenkernen.

wird im Durchschnitt bereits etwa 13,6% des Cache-Speichers invalidiert.

Das Ausmaß der Invalidierung hängt auch von der Anzahl der Rechenkerne ab, die an der parallelen Ausführung beteiligt sind. Mit steigender Anzahl an Teilnehmern sinkt die Menge an Daten, die ein einzelner Rechenkerne bearbeiten muss. Dementsprechend sinkt auch die Zahl der Invalidierungen. Dies trifft auf den ODC² wie auch auf *Cache Flush* zu. Doch unabhängig von der Anzahl der Rechenkerne ist das Ausmaß der Invalidierung beim ODC² stets erheblich geringer als bei *Cache Flush*. Die entsprechenden Abbildungen für die Ausführung mit 2 - 8 Rechenkernen sind in Anhang 4 zu finden.

Diese Unterschiede machen deutlich, wie groß der Effekt einer Beschränkung der Invalidierung ist, die ausschließlich auf Cache-Zeilen mit gemeinsamen Daten hinzielt. Die Anzahl an privaten Daten, die beim *Cache Flush* Verfahren unnötig invalidiert werden, ist erheblich und sorgt für eine stark erhöhte Zahl an Zugriffen auf den gemeinsamen Speicher.

5.5.3 Zugriffsverhalten im Cache

Eine detailliertere Betrachtung der verschiedenen Typen von Speicherzugriffen bei der Ausführung eines Benchmarks ermöglicht Aufschluss darüber, wie die verschiedenen Resultate der WCET-Abschätzung und Laufzeitsimulation zustande kommen. Für die Auswertung wird zwischen folgenden Zugriffstypen unterschieden: *Memory* beschreibt direkte Zugriffe auf den gemeinsamen Speicher, die unter Umgehung des Caches durchgeführt werden. Darunter fallen Zugriffe auf gemeinsame Daten bei der *Uncached* Plattform sowie Zugriffe auf Synchronisationsvariablen. Desweiteren werden Schreibzugriffe betrachtet, die als *Write-Through* durchgeführt werden, sowie Zugriffe die einen *Cache-Miss* und damit einen Zugriff auf den gemeinsamen Speicher erzeugen.

Das Zugriffsverhalten im Cache wurde bei paralleler Ausführung mit 2 - 8 Rechenkernen untersucht. Es hat sich gezeigt, dass die Anzahl der beteiligten Rechenkerne keinen relevanten Einfluss auf die Resultate und die daraus zu ziehenden Schlussfolgerungen hat. Die Auswertung in diesem Kapitel beschränkt sich daher auf die Ergebnisse bei Ausführung mit 4 Rechenkernen. Zudem ist es für die Veranschaulichung des Zugriffsverhaltens der untersuchten Kohärenzverfahren nicht erforderlich, die Ergebnisse sämtlicher Benchmarks zu kommentieren. Die Auswertung beschränkt sich daher auf die Resultate der Benchmarks Matrix, FFT, Smoothing und 3DPP. Eine vollständige Sammlung der Abbildungen aller Benchmarks bei Ausführung mit 2 - 8 Rechenkernen ist in Anhang 5 zu finden.

Matrix Multiplikation

Der Benchmark Matrix zeichnet sich durch einen besonders effektiv parallelisierbaren Algorithmus aus. In der Simulation skaliert die Laufzeit nahezu

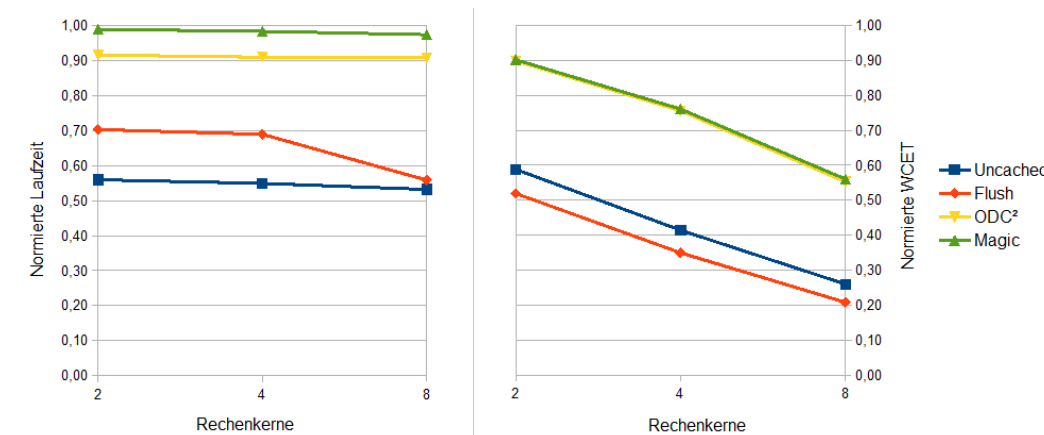


Abbildung 5.13: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von Matrix.

konstant mit steigender Anzahl an Rechenkernen, wie an Abbildung 5.13 links, zu sehen ist. Die Werte in der Abbildung sind, wie zuvor bei Abbildung 5.8, normiert auf die theoretisch mögliche Laufzeitverbesserung ausgehend vom Resultat der sequentiellen Ausführung. Auch bei diesem Benchmark rentiert sich der Einsatz eines Cache-Speichers für den Zugriff auf gemeinsame Daten, in Hinsicht auf die gemessene Laufzeit wie auch auf die $WCET_{est}$. Doch wie an der WCET-Abschätzung für die Plattformen ODC² und *Cache Flush* zu erkennen ist, schneiden diese Plattformen sehr unterschiedlich ab. Während die Resultate des ODC² annähernd an die der *Magic* Plattform heranreichen, übertrifft die $WCET_{est}$ der *Cache Flush* Plattform sogar die Werte von *Uncached*. Der Matrix Benchmark ist ein gutes Beispiel für ein Zugriffsmuster, dass für den ODC² von Vorteil ist. Einer Vielzahl von Lesezugriffen auf gemeinsame (Read-Only) Daten stehen vergleichsweise wenig Schreiboperationen auf gemeinsame Daten gegenüber.

In Abbildung 5.14 ist die Anzahl der unterschiedlichen Speicherzugriffe bei Ausführung des Benchmarks mit 4 Rechenkernen dargestellt. Es wurden sowohl Zugriffe während der Simulation gemessen (oben), als auch die erwarteten Zugriffe bei der statischen Analyse (unten) festgehalten. Da sowohl Zugriffe mit sehr geringer als auch sehr hoher Anzahl auftreten, wurde eine logarithmische Skalierung für die Y-Achse gewählt.

Die deutlich erhöhte Anzahl an Speicherzugriffen (Gesamt) bei der *Cache Flush* Plattform im Vergleich zu ODC² macht noch einmal die Ursache für die Unterschiede in Laufzeit und $WCET_{est}$ deutlich. Aufgrund der kontinuierlichen Write-Through Strategie treten bei *Cache Flush* signifikant mehr solcher Schreibzugriffe auf als bei ODC². In den übrigen Zugriffstypen unterscheiden sich beide Plattformen kaum. Bei Matrix nehmen Schreibzugriffe auf gemein-

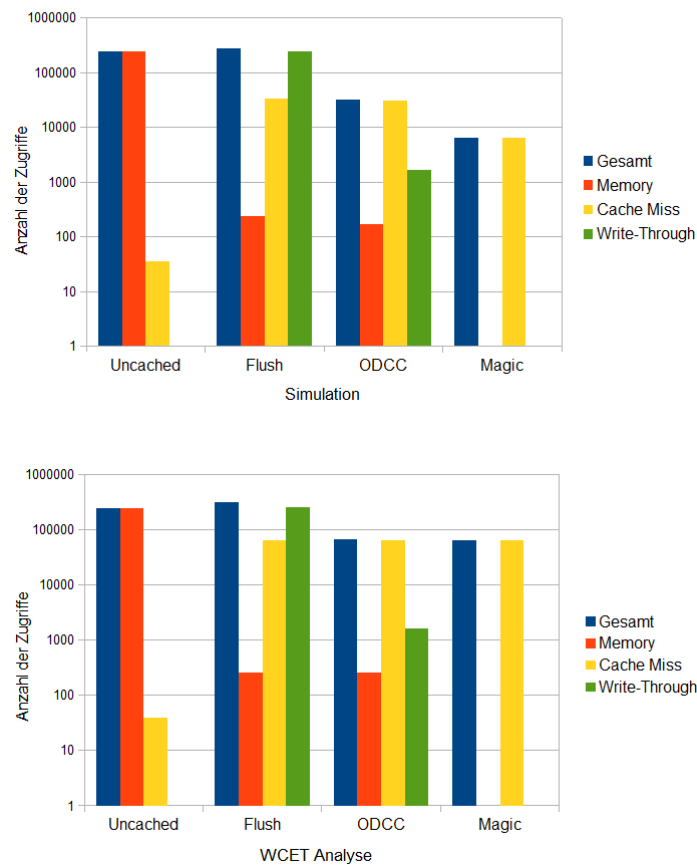


Abbildung 5.14: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Matrix mit 4 Rechenkernen.

same Daten eine äußerst geringe Rolle ein, vergleichen mit Schreibzugriffen auf private Daten. Und die Technik des ODC² ermöglicht es, diese Schreibzugriffe auf private Daten lokal im Cache durchzuführen. Das Mehr an Zugriffen auf den gemeinsamen Speicher bei *Cache Flush* wirkt sich bei der WCET-Abschätzung noch gravierender aus als in der Simulation (siehe Abbildung 5.13, rechts), da immer die maximale Latenz für solche Speicherzugriffe angenommen werden muss. Es sind also gerade Schreibzugriffe, die eine herausragende Rolle bei der Performanz der *Cache Flush* Plattform spielen.

Fast Fourier Transformation

Der FFT Benchmark macht deutlich, dass sich das Zugriffsmuster auf gemeinsame Daten auch nachteilig auf den ODC² auswirken kann. Der Algorithmus weist ausgesprochen viele Schreibzugriffe auf und der Anteil an Schreibzugrif-

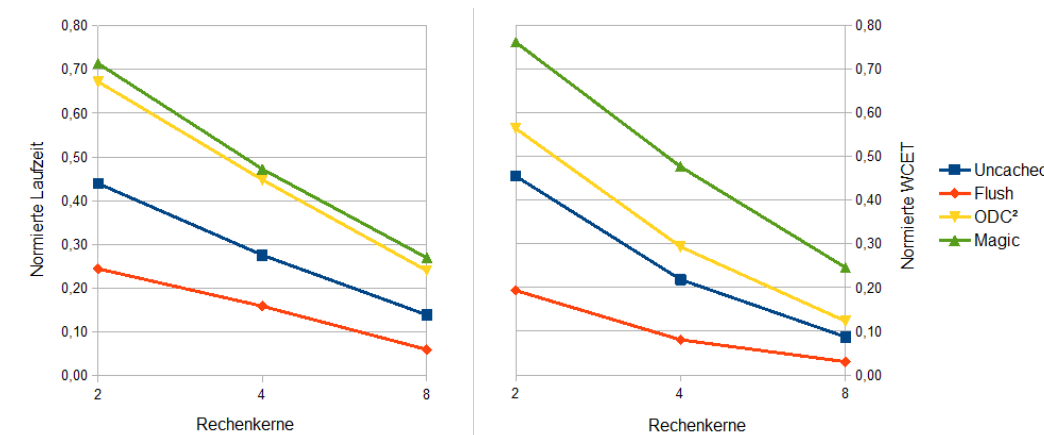


Abbildung 5.15: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von FFT.

fen auf gemeinsamen Daten ist größer als bei anderen Benchmarks. Auffallend sind die Unterschiede zwischen Laufzeit und WCET-Abschätzung der ODC² Plattform (siehe Abbildung 5.15). Betrachtet man die Resultate der *Uncached* und *Magic* Plattformen als obere und untere Schranke, so schneidet der ODC² bei der statischen Analyse deutlich schlechter ab als bei der Simulation. Der Grund dafür liegt in der hohen Zahl an Schreibzugriffen auf gemeinsame Daten. Diese werden im ODC² mittels Write-Through durchgeführt und resultieren in Zugriffen auf den gemeinsamen Speicher, und dementsprechend in hohen Zugriffslatenzen.

In Abbildung 5.16 ist zu sehen, dass die bei der statischen Laufzeitanalyse ermittelte Anzahl an Write-Through Zugriffen besonders hoch ist, im Vergleich zu den übrigen Zugriffen. Der ODC² bekommt bei diesem Benchmark den negativen Effekt von Write-Through Zugriffen deutlich zu spüren, unter dem die Performanz der *Cache Flush* Plattform generell leidet.

SUSAN Smoothing

Der SUSAN Smoothing Benchmark zeichnet sich dadurch aus, dass viele kostspielige Berechnungen auf einen jeweils kleinen Bereich der gemeinsamen wie auch privaten Daten durchgeführt wird. Die benötigte Laufzeit für die Ausführung dieses Benchmarks ist aufgrund der aufwändigen Berechnungen mit Abstand am größten und in etwa so groß wie die aller andern Benchmarks zusammen. In Relation dazu ist die dabei verwendete Menge an Daten sehr gering. Diese besondere Charakteristik sorgt dafür, dass mit dem ODC² eine simulierte Laufzeit erreicht wird, die nahezu identisch mit der *Magic* Plattform ist. In Abbildung 5.17 sind die Graphen für ODC² und *Magic* nahezu deckungsgleich. Da die Menge an Invalidierungen nach einer Restore-Procedure

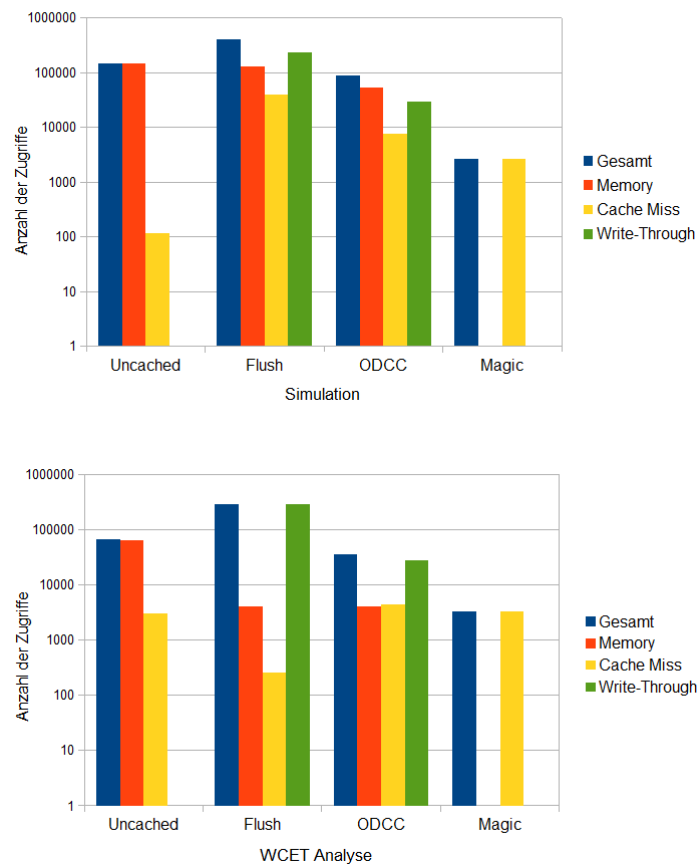


Abbildung 5.16: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von FFT mit 4 Rechenkernen.

des ODC^2 bei Smoothing sehr gering ist (siehe Abbildung 5.11), ist auch der zusätzliche Overhead an Cache-Misses gering. Wie in Abbildung 5.18 (oben) zu erkennen ist, erzeugen die Plattformen ODC^2 und *Magic* in etwa die gleiche Anzahl an Cache-Misses. Der ODC^2 profitiert auch hier davon, dass die Invalidierung gemeinsamer Daten die Ersetzungen von Cache-Zeilen mit privaten Daten verringert. Der in diesem Benchmark hohe Anteil an Zugriffen auf private Daten erhöht die Anzahl der Invalidierungen und Write-Through Zugriffe bei der *Cache Flush* Plattform im Vergleich zu ODC^2 und *Magic*. Besonders die Schreibzugriffe auf private Daten, die bei der *Cache Flush* Plattform im Gegensatz zur *Uncached* Plattform zu Zugriffen auf den gemeinsamen Speicher führen, sorgen dafür, dass die *Cache Flush* Plattform bei der Simulation am schlechtesten abschneidet.

Der Smoothing Benchmark enthält Zugriffe auf Arrays mit gemeinsamen Daten, die über Zeiger auf Zeiger und Arrays geregelt werden. Diese für eine

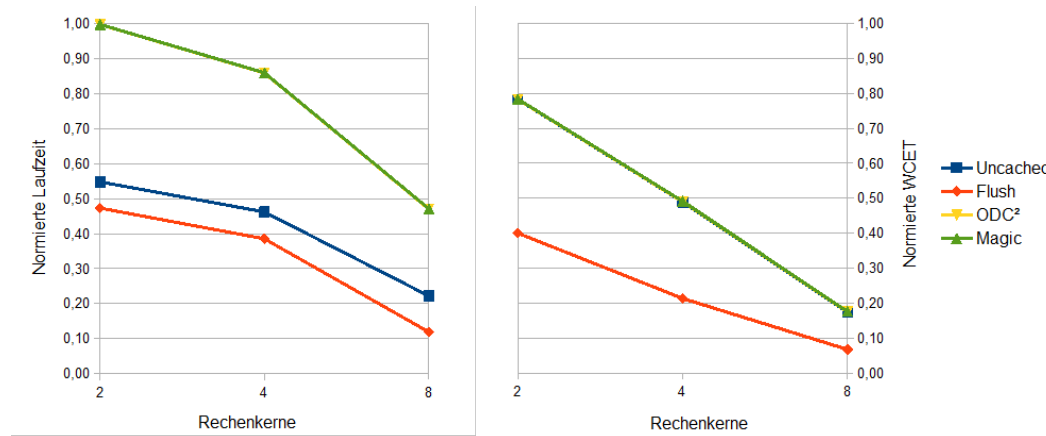


Abbildung 5.17: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von Smoothing.

Cacheanalyse unvorteilhafte Implementation kann es unter Umständen unmöglich machen, die Zieladresse eines Zugriffs zu ermitteln. Dies resultiert in einem hohen Pessimismus bei der WCET-Abschätzung [43]. Dies gilt aufgrund der hohen Zugriffsrates auf private Daten auch für die *Uncached* Plattform. Wie in Abbildung 5.17 rechts zu sehen ist, ist die ermittelte WCET-Abschätzung für alle Plattformen (mit Ausnahme von *Cache Flush*) nahezu identisch. Lediglich *Cache Flush* erzielt aufgrund der Schreibstrategie wiederum höhere Werte. Das schlechte Abschneiden ist in der geringen Präzision der WCET-Abschätzung begründet. Abbildung 5.18 unten zeigt, dass die Anzahl an Cache-Misses für alle Plattformen ausgesprochen hoch ist. Die damit einhergehenden hohen Zugriffslatenzen sorgen dafür, dass sich die WCET-Abschätzung für diese Plattformen kaum mehr unterscheidet.

3D Path Planing

Besonders interessant aufgrund seiner Implementierung einer industriellen Applikation ist der 3DPP Benchmark. Zudem wurde im Rahmen des parMERSA Projekts eine parallelisierte Implementierung entworfen, die auf die Anforderungen einer statische WCET-Analyse ausgelegt ist [68].

Die Übersicht über die normierte Laufzeit und WCET-Abschätzung in Abbildung 5.19 (identisch zu Abbildung 5.8) zeigt, dass mit dem ODC² Resultate erzielt werden, die nah am theoretischen denkbaren Optimum der *Magic* Plattform liegen. Mit steigender Anzahl an Rechenkernen erzielt der ODC² sogar eine bessere WCET-Abschätzung. Es wird je nach Anzahl der Rechenkerne eine um 2,6% - 8,6% höhere Laufzeit und eine Verringerung der $WCET_{est}$ um bis zu 7,2% erreicht. Der hohe Anteil an Schreibzugriffen bei 3DPP wirkt sich wiederum nachteilig auf die *Cache Flush* Plattform aus. Wie in Abbildung 5.20

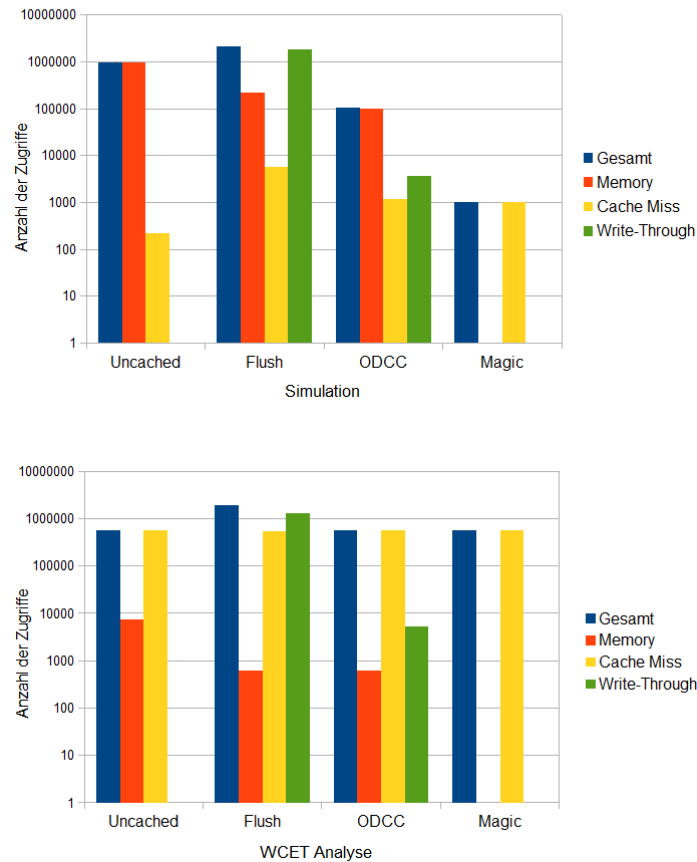


Abbildung 5.18: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Smoothing mit 4 Rechenkernen.

oben zu sehen, fällt in der Simulation die Anzahl an direkten Zugriffen auf den gemeinsamen Speicher (Memory) für jede Plattform relativ hoch aus. Dies ist in der Task-Parallelität des Benchmarks begründet. Während bei Daten-Parallelität die Arbeit einzelner Rechenkerns zwischen Synchronisationspunkten zumeist relativ ausgeglichen ist, ist dies bei Task-Parallelität oft nicht der Fall. Rechenkerns, die lange an Synchronisationsgrenzen warten müssen, greifen häufig auf die Synchronisationsvariablen zu, die in allen Plattformen mit Ausnahme von *Magic* ungedcached sind. In den Werten der Worst-Case Zugriffe ist dies nicht mehr zu erkennen, da bei der statischen Laufzeitanalyse für Wartezeiten an Synchronisationsgrenzen eine obere Schranke angenommen wird, ohne dabei auftretende Zugriffe zu berücksichtigen. An den Werten für die Worst-Case Anzahl von Cache-Misses ist wiederum der Pessimismus der Cacheanalyse zu erkennen. Der Einfluss auf die WCET hält sich jedoch aufgrund des verhältnismäßig niedrigen Anteils an Lesezugriffen in Grenzen.

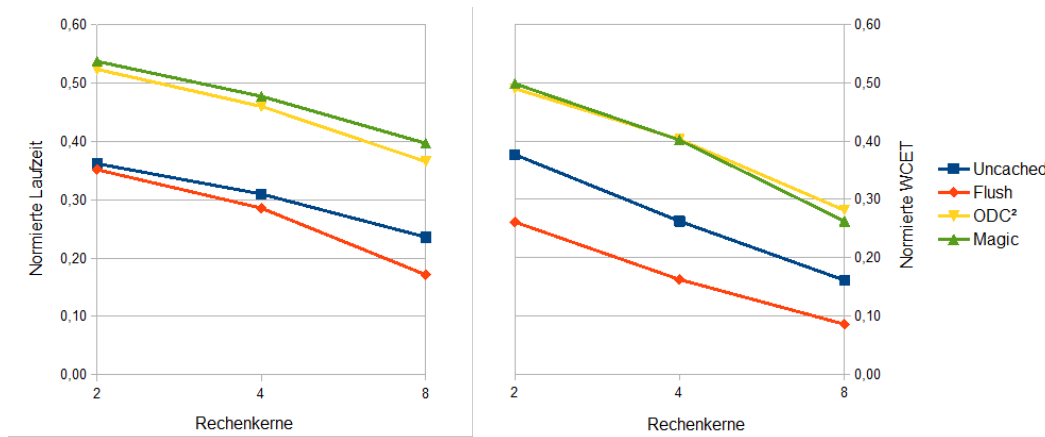


Abbildung 5.19: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von 3DPP.

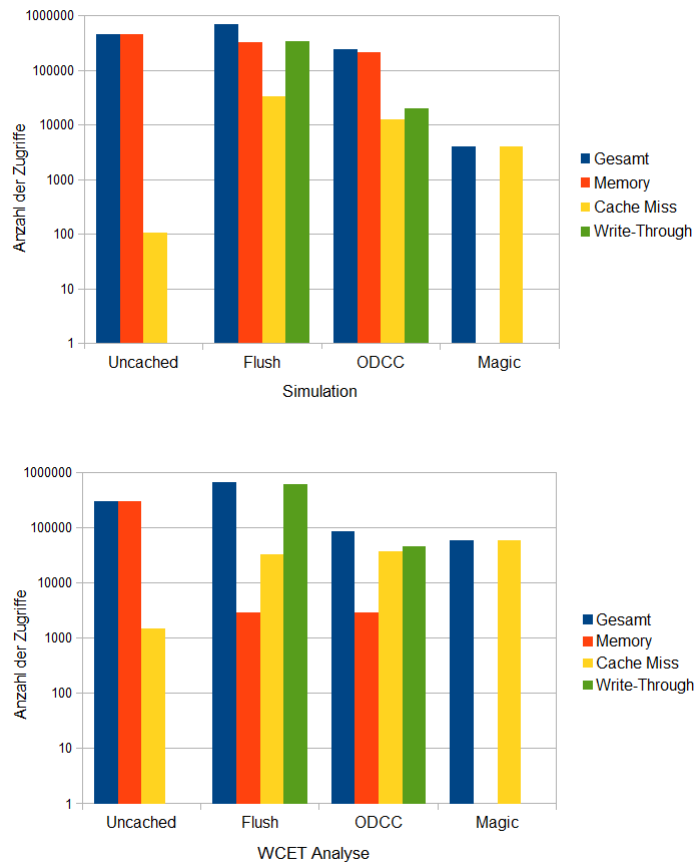


Abbildung 5.20: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von 3DPP mit 4 Rechenkernen.

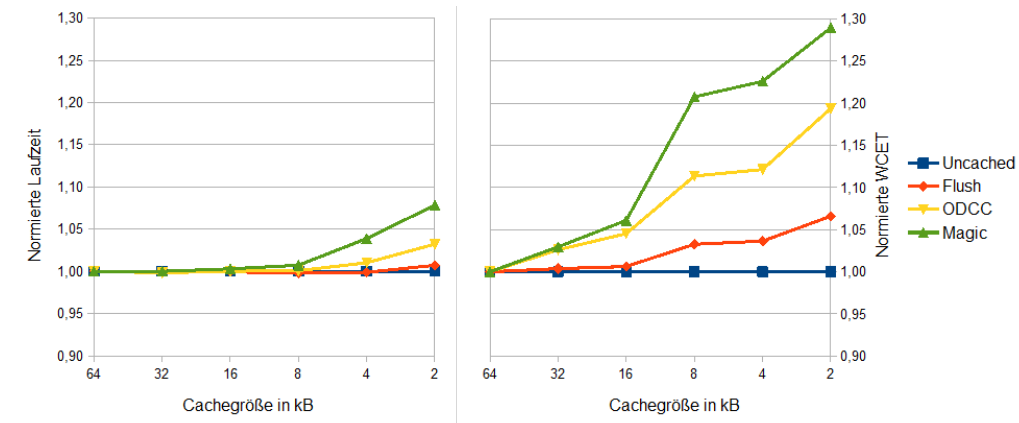


Abbildung 5.21: Auf die Ausführung mit 64 kB normierte Laufzeit und $WCET_{est}$ bei Variation der Cachegröße bei Ausführung von 3DPP mit 4 Rechenkernen.

5.5.4 Variation der Cachekonfiguration

Die Größe und Organisation des Cache-Speichers waren bislang konstante Parameter der Evaluation. Im Folgenden wird untersucht, wie sich eine Variation der Cachekonfiguration bei den untersuchten Plattformen auswirkt. Hierfür wird der industrielle 3DPP Benchmark erneut evaluiert, wobei Cachegröße, Assoziativität und Cache-Zeilengröße mehrfach modifiziert werden. Ziel ist es zu ergründen, in wieweit sich eine Veränderung dieser Parameter zum Vor- oder Nachteil der Kohärenzverfahren auswirkt. Als Basis wird die bisherige Konfiguration mit einer Cachegröße von 16 kB, 2-facher Assoziativität und einer Zeilengröße von 16 Byte verwendet.

Cachegröße

Beginnend mit einer Cachegröße von 64 kB wird die Größe des Cache-Speichers (bei gleichbleibender Assoziativität und Cache-Zeilengröße) schrittweise halbiert, bis zu einer Größe von 2 kB. Da eine Verkleinerung des Cache-Speichers einer Verringerung vorhandener Cache-Sätze entspricht, ist bei allen Plattformen mit einem ausschließlich negativen Effekt auf die Laufzeit zu rechnen [126]. In Abbildung 5.21 ist die simulierte Laufzeit und die WCET-Abschätzung bei Ausführung mit 4 Rechenkernen dargestellt. Die Werte für jede Plattform sind auf eine Ausführung mit 64 kB Cachegröße normiert und zeigen wie sich die Laufzeit mit abnehmender Cachegröße verändert. Wie zu erwarten war, steigt die Laufzeit je nach Plattform um bis zu 8% (Simulation) bzw. 29% (WCET-Analyse). Die *Cache Flush* Plattform erzielt einen deutlich geringeren Anstieg der Laufzeit im Vergleich zur ODC² und *Magic* Plattform. Eine geringe Steigerung der Laufzeit deutet auf eine geringe Auslastung des Cache-Speichers hin.

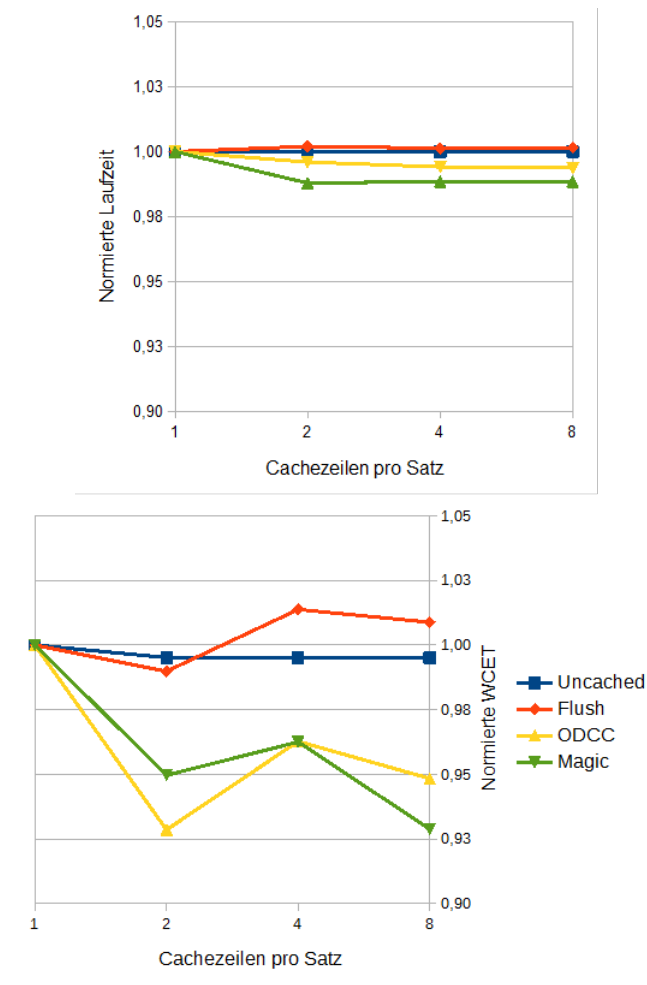


Abbildung 5.22: Auf die Ausführung mit einem direkt abgebildeten Cache normierte Laufzeit und $WCET_{est}$ bei Variation der Assoziativität bei Ausführung von 3DPP mit 4 Rechenkernen.

Die konstant bleibende Laufzeit der *Uncached* Plattform, die lediglich eine geringe Menge privater Daten im Cache speichert, stützt diese Annahme. Bei der *Magic* Plattform, bei der Cache-Zeilen erst durch Ersetzung auf dem Cache entfernt werden, ist die Cache-Auslastung und folgerichtig die Steigerung der Laufzeit mit abnehmender Cachegröße am höchsten. Der ODC² erreicht in diesem Sinne eine signifikant höhere Auslastung des Cache-Speichers als die *Cache Flush* Plattform. Nichtsdestotrotz sinkt der Vorsprung in der Performanz des ODC² gegenüber *Cache Flush* mit einer Verkleinerung des Cache-Speichers.

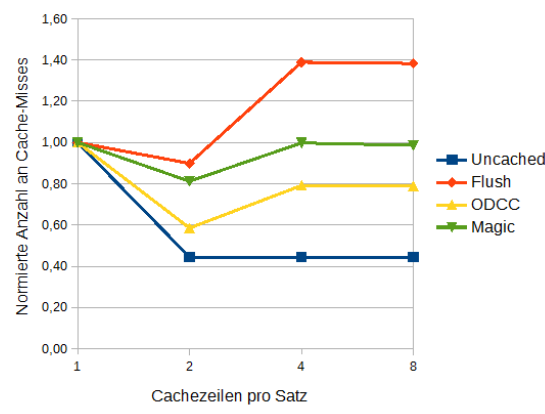


Abbildung 5.23: Auf die Ausführung mit einem direkt abgebildeten Cache normierte Anzahl der Cache-Misses bei Variation der Assoziativität bei Ausführung von 3DPP mit 4 Rechenkernen.

Assoziativität

Für den Einfluss der Assoziativität eines Cache-Speichers auf die Performanz wird ein direkt abgebildeter Cache (1-fach Assoziativ) sowie 2-, 4- und 8-fache Assoziativität betrachtet. Die in Abbildung 5.22 dargestellten Werte sind für jede Plattform auf die Ausführung mit einem direkt abgebildeten Cache normiert. Betrachtet man die Veränderungen bei der Laufzeitsimulation, so ist zu erkennen, dass der ODCC² wie auch die *Magic* Plattform von einer erhöhten Assoziativität profitieren. Während die Werte bei *Uncached* konstant bleiben, ist bei *Cache Flush* eine geringe Erhöhung der Laufzeit festzustellen. Bei der WCET-Analyse sind diese Auswirkungen noch deutlicher zu sehen. Eine Erhöhung der Assoziativität kann sich in erster Linie positiv auf Anzahl möglicher Ersetzungen von Cache-Zeilen auswirken. Der Wechsel von einem direkt abgebildeten zu einem 2-fach assoziativen zeigt dies sehr deutlich. Wie in Abbildung 5.23 zu sehen ist, treten bei einem 2-fach assoziativen Cache deutlich weniger Cache-Misses auf. Mit zunehmender Assoziativität schlägt jedoch auch die damit einhergehende Reduzierung der Cache-Sätze zu Buche. Gerade bei der *Magic* Plattform, die den Cache-Speicher wohl am stärksten auslastet, macht sich dies bemerkbar.

Cache-Zeilengröße

Die Größe einer Cache-Zeile wird von 4 Byte, der Wortgröße in der Architektur, bis zu einer Größe von 32 Byte variiert. Um die Cachegröße beizubehalten wird dabei entsprechend die Anzahl der Cache-Sätze verringert. Es ist zu erwarten, dass sich durch die Vergrößerung der Cache-Zeile die Anzahl an Cache-Misses reduziert. Dies sollte eine Verringerung der Ausführungszeit bewirken. Zum

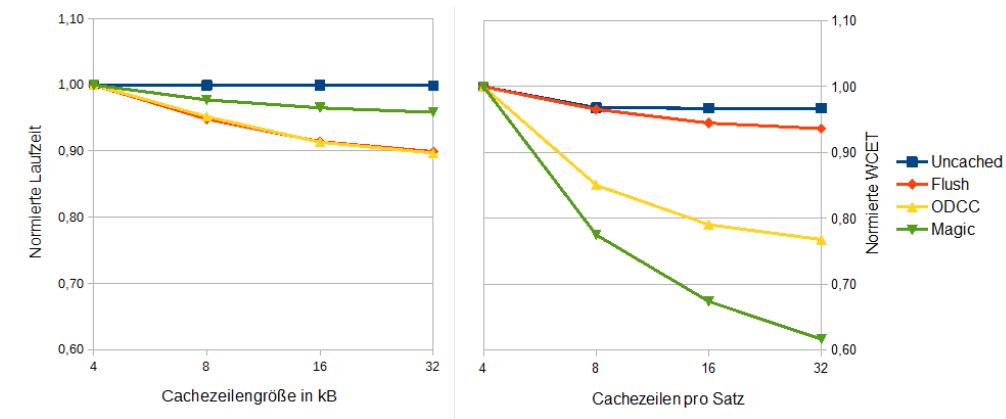


Abbildung 5.24: Auf die Ausführung mit 4 Byte Cache-Zeilengröße normierte Laufzeit und $WCET_{est}$ bei Variation der Cache-Zeilengröße bei Ausführung von 3DPP mit 4 Rechenkernen.

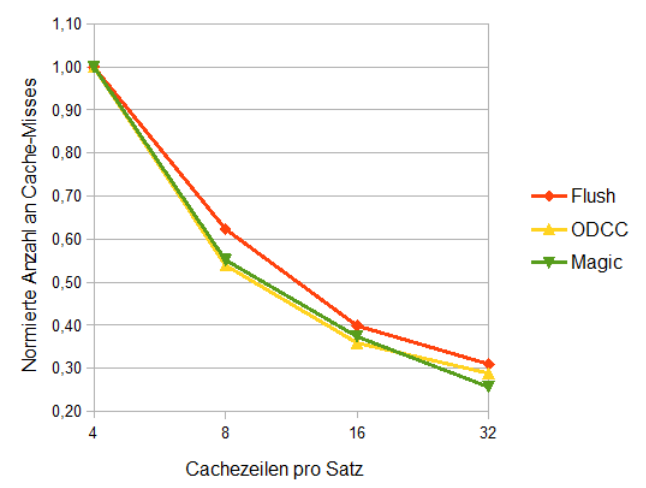


Abbildung 5.25: Auf die Ausführung 4 Byte Cache-Zeilengröße normierte Anzahl der Cache-Misses bei Variation der Cache-Zeilengröße bei Ausführung von 3DPP mit 4 Rechenkernen.

anderen ist beim Laden und Zurückschreiben einer Cache-Zeile eine größere Menge an Daten betroffen, wodurch wiederum die Zugriffszeit eines Cache-Misses erhöht wird. Wie in Abbildung 5.24 zu sehen ist, überwiegt bei allen Plattformen der positive Effekt auf die Laufzeit. Die Resultate sind für jede Plattform auf die Ausführung mit 4 Byte Zeilengröße normiert.

Der positive Effekt macht sich besonders bei der WCET-Abschätzung bemerkbar. Eine Erhöhung der Cache-Miss Latenz aufgrund einer höheren Cache-Zeilengröße fällt bei der ohnehin sehr hohen maximalen Latenz kaum ins Gewicht. In Abbildung 5.25 ist zu sehen wie sehr die Anzahl der Cache-Misses bei den cachebasierten Plattformen, bezogen auf die Anzahl an Cache-Misses bei 4 Byte Zeilengröße, reduziert wird. Die relative Verringerung der Anzahl an Cache-Misses ist bei den drei Plattformen in etwa gleich groß. Die WCET-Abschätzung verringert sich daher besonders bei den Plattformen, die eine hohe Zahl an Cache-Misses generieren.

5.6 Fazit

Die Ergebnisse der Evaluation zeigen, dass bei Einsatz des ODC² Verfahrens eine signifikante Verbesserung der (maximalen) Laufzeit und $WCET_{est}$, im Vergleich zu den Verfahren der *Uncached* und *Cache Flush* Plattformen erreicht wird. Die bei der simulierten Ausführung ermittelten Laufzeiten der Benchmarks bestätigen, dass trotz wiederholter Invalidierung gemeinsamer Daten und der temporären Write-Through Strategie, ein effizienter Zugriff auf diese Daten mit Hilfe des ODC² möglich ist. Betrachtet man die *Magic* Plattform als Referenz, so ist die Performanzeinbuße mitunter sehr gering. Aufgrund der zeitlichen Vorhersehbarkeit der Zugriffe beim ODC² gilt diese Schlussfolgerung auch für die Ermittlung der maximalen Laufzeit. Die Berechnung einer oberen Laufzeitschranke profitiert von der Handhabung der gemeinsamen Daten im ODC².

Eine detaillierte Auswertung der Simulation macht deutlich, dass der Overhead an zusätzlicher Laufzeit, der durch die Kohärenzoperationen entsteht, beim ODC² deutlich geringer ausfällt als mit einer reinen Software-Kohärenz bei der *Cache Flush* Plattform. Trotz wiederholter Invalidierung und damit erhöhter Anzahl an Cache-Misses bleibt die Zahl der Zugriffe auf den gemeinsamen Speicher deutlich hinter der zurück, die bei Zugriff unter Umgehung des Caches resultiert. Diese Zahl ist bei der *Cache Flush* Plattform zumeist am höchsten, da deutlich mehr Invalidierungen stattfinden als beim ODC². Unumgänglich für die *Cache Flush* Plattform ist außerdem die kontinuierliche Anwendung der Write-Through Schreibstrategie. Dies hat sich als Flaschenhals für die Performanz erwiesen. Eine Zunahme der Zugriffe auf den gemeinsamen Speicher hat insbesondere bei der WCET-Analyse einen großen Einfluss auf die

Laufzeit. In dieser Hinsicht bietet das ODC² Verfahren gute Voraussetzungen für die Ermittlung einer präzisen Laufzeitschranke.

Aufgrund der deutlichen Performanzsteigerung beim Einsatz des ODC² im Vergleich zu den Alternativen in aktuellen Systemen erscheint die Folgerung plausibel, dass sich der zusätzliche Aufwand für die Erweiterung der Cache-Hardware und der Ergänzung der Software mit ODC²-Kontrolloperationen schlussendlich rentiert.

Kapitel 6

Zusammenfassung und Ausblick

In den Architekturen heutiger und zukünftiger Echtzeitsysteme spielen Mehrkern-Prozessoren eine zentrale Rolle. Die parallele Ausführung von mehreren Applikationen auf einem Prozessor sowie die Ausführung parallelisierter Applikationen auf mehreren Rechenkernen zugleich, soll die steigenden Anforderungen an Performanz erfüllen. Der konkurrierende Zugriff auf gemeinsame Ressourcen hat sich als Flaschenhals für die Performanz herausgestellt. Dies gilt insbesondere für die Ermittlung einer maximalen Laufzeitschranke. Der mögliche Wettstreit um Ressourcen, wie gemeinsamer Speicher und Kommunikationsmedium, treibt die Überabschätzung der WCET in die Höhe. Hinzu kommt, dass gängige, auf hohe Ausführungseffizienz ausgerichtete Architekturoptimierungen, die zeitliche Vorhersagbarkeit des Systems negativ beeinträchtigen. In der aktuellen Forschung wird daher intensiv an geeigneten Architekturen und verbesserten Methoden für die Laufzeitanalyse gearbeitet.

Cache-Speicher stellen eine besonders kritische Komponente in Hinsicht auf zeitliche Vorhersagbarkeit dar. Als optimale Lösung für die Vermeidung hoher Zugriffslatenzen auf einen Off-Chip Speicher sind sie in heutigen Architekturen kaum noch wegzudenken. Doch für einen fehlerfreien Einsatz in Mehrkern-Prozessoren wird die Verwendung eines Verfahrens zur Erhaltung der Cache-Kohärenz vorausgesetzt. Jedoch hat sich gezeigt, dass gängige Cache-Kohärenzverfahren einen äußerst negativen Einfluss auf die WCET-Abschätzung einer statischen Laufzeitanalyse haben. Infolgedessen werden Cache-Speicher in harten Echtzeitsystemen kaum oder gar nicht eingesetzt. Scratchpad-Speicher haben vielerorts die Rolle von Cache-Speichern übernommen. Sie entsprechen den Anforderungen an zeitlicher Vorhersagbarkeit. Im Gegensatz zu Cache-Speichern, muss die Verwaltung der Daten innerhalb ei-

nes Scratchpads vom Benutzer selbst getätigt werden. Außerdem kann ein gemeinsames Datum zu einer Zeit in nur einem privaten Scratchpad gespeichert werden. Ein schneller Zugriff auf ein gemeinsames Datum ist also nicht für alle Rechenkerne möglich.

Auf die Vorteile, die Cache-Speicher zu einen unverzichtbaren Bestandteil einer Speicherhierarchie machen, sollte auch in einem echtzeitfähigen Mehrkern-Prozessor nicht verzichtet werden müssen. Echtzeitfähige Cache-Kohärenz kann dazu beitragen, dass Cache-Speicher zusammen mit Scratchpad-Speichern eine Architektur bilden, die den Anforderungen an Performanz gerecht wird.

In dieser Arbeit werden Cache-Kohärenzprotokolle, wie sie in aktuellen Mehrkern-Architekturen zu finden sind, auf ihre Auswirkungen auf die zeitliche Vorhersagbarkeit hin untersucht. Es werden Kohärenzoperationen betrachtet, die repräsentativ für verschiedenen Klassen von Kohärenzprotokollen stehen. Für diese Operationen wird dargelegt, welchen Einfluss sie auf die Cacheanalyse und die Latenz von Cache-Zugriffen haben. Anhand dieser Bewertungen ist es möglich, die Cache-Kohärenzverfahren in aktuellen, wie auch zukünftigen Architekturen auf ihre Echtzeitfähigkeit hin zu bewerten. Diese Arbeit ermöglicht Rückschlüsse darauf, wie ein Hardware-Kohärenzprotokoll konzipiert werden muss, um eine möglichst gute zeitliche Vorhersagbarkeit zu erzielen.

Das in dieser Arbeit vorgestellte ODC² Kohärenzverfahren stellt eine praktikable Lösung für echtzeitfähige Cache-Kohärenz dar. Es greift auf Methoden der Hardware- und Software-Kohärenz zurück, wobei nur solche Techniken berücksichtigt werden, die von einer statischen Laufzeitanalyse analysiert werden können. Diese Techniken sind Teil eines Gesamtkonzepts, das eine möglichst effiziente Nutzung des Cache-Speichers ermöglicht und dabei die zeitliche Vorhersagbarkeit aufrecht erhält.

Der ODC² setzt eine Erweiterung der Cache-Hardware voraus, die auf bereits in Cache-Speicher vorhandenen Funktionen aufbaut und keinen übermäßigen Overhead an Chipfläche beansprucht. Hingegen wird für die Anwendung des ODC² eine Anpassung der auszuführenden Applikation vorausgesetzt. Eine korrekte Anpassung einer Applikation setzt detailliertes Wissen über den Programm- und Datenfluss voraus und kann manuell, prinzipiell aber auch automatisiert, durchgeführt werden. Ein Hauptmerkmal des ODC²-Verfahrens ist, dass Cache-Kohärenz nur bei Bedarf sichergestellt wird. Die Kohärenzoperationen des ODC² werden explizit über die Kontrollanweisungen aktiviert. Es ist in einem System mit On-Demand Coherent Cache also problemlos möglich, Legacy-Software unverändert auszuführen, sofern diese auf Software-Kohärenztechniken zurückgreift, die vom Cache unterstützt werden.

In einer Evaluation wird der ODC² mit zwei gängigen, zeitlich vorhersagbaren Verfahren zur Erhaltung kohärenter Zugriffe verglichen: dem Zugriff auf gemeinsame Daten unter Umgehung des Caches sowie die Ausführung von An-

weisungen zur Invalidierung des Cache-Speichers. Beide Vergleichsverfahren ermöglichen eine zeitliche Vorhersagbarkeit der Zugriffe, verursachen jedoch eine signifikant höhere Abschätzung der WCET im Vergleich zu ODC². Das Kohärenzverfahren des ODC² erzeugt eine deutlich geringere Anzahl an Zugriffen auf den gemeinsamen Speicher, die aufgrund ihrer hohen maximalen Latenzen in Mehrkern-Prozessoren maßgeblich für die Überabschätzung der WCET verantwortlich sind.

6.1 Ausblick

Das ODC²-Verfahren bietet mögliche Ansätze zur Erweiterung und Modifikation, die lohnenswert für zukünftige Untersuchungen sind:

Weitere Ebenen des Shared-Mode

Es ist denkbar, mehrere Ebenen des Shared-Mode verfügbar zu machen. Analog zu verschachtelten Schleifen, kann der Cache mehrfach einen Shared-Mode betreten und dabei kennzeichnen in welcher Ebene des Shared-Mode er sich aktuell befindet. Cache-Zeilen werden dann nicht nur als geteilt markiert, sondern auch in welcher Ebene des Shared-Mode sie markiert wurden. Verlässt der Cache den Shared-Mode der aktuellen Ebene, so werden nur die Cache-Zeilen invalidiert, die in der entsprechenden Ebene markiert wurden. Mehrere Ebenen des Shared-Mode wären eine Möglichkeit, den theoretisch denkbaren Fall zu unterstützen, dass ein Rechenkern ineinander verschachtelte kritische Abschnitte betritt. Andererseits wäre es denkbar, dass der Cache durch Anwendung mehrerer Ebenen des Shared-Mode effizienter genutzt werden kann. Bei sehr umfangreichen Codeabschnitten, könnte durch zwischenzeitliche Invalidierung eines Teils der gemeinsamen Daten, der positive Effekt der Invalidierung ausgenutzt werden, wie er in Kapitel 4.1.4 beschrieben ist.

Für diese Erweiterung wäre es notwendig die Anzahl an Shared-Bits innerhalb der Kontrollinformationen jedes Cache-Blocks auf die Zahl der gewünschten Ebenen zu erhöhen. Dieser erhöhte Speicherbedarf des Verfahrens muss durch eine verbesserte Laufzeit gerechtfertigt werden. Es müsste untersucht werden, welche Programmkonstrukte sich für diese Erweiterung eignen und anschließend evaluieren, ob tatsächlich eine positive Auswirkung auf die Laufzeit festzustellen ist.

Behandlung von False Sharing

In Kapitel 4.1.3 wird erläutert, wie durch Anwendung der Write-Through Schreibstrategie verhindert wird, das *False Sharing* zu inkohärentem Verhalten führt. Eine denkbare Alternative wäre es, die Datenwörter einer Cache-Zeile separat mit einem Dirty-Bit zu markieren und in der Restore-Procedure nur

die markierten Datenwörter der Cache-Zeile zurückzuschreiben. Dies ist jedoch mit einem Overhead an zusätzlichem Speicher und zusätzlicher Logik für die Cache-Hardware verbunden. Das ODC²-Verfahren wäre dabei nicht mehr so flexibel in beliebige Cache-Speicher integrierbar. Nichtsdestotrotz könnte es sinnvoll sein, diese Technik zu evaluieren.

Automatisierte Anpassung der Software

In Kapitel 4.2 wurde bereits angemerkt, dass eine Platzierung der Anweisungen zum Betreten und Verlassen des Shared-Mode auch automatisiert stattfinden kann. Ein möglicher Ansatz für weitere Forschung wäre es, zu ergründen, wie solch eine Platzierung, basierend auf einer compilergestützten Analyse des Programmcodes, durchzuführen ist. Auch für die korrekte Abbildung privater und gemeinsamer Daten, gemäß den Anforderungen des ODC², kann ein Compiler eingesetzt werden. Eine automatisierte Anpassung von Applikationen würde einen großen Gewinn für das ODC²-Verfahren darstellen. Integriert in eine Entwicklungsumgebung wäre der Einsatz des ODC² ebenso transparent für den Benutzer, wie es Hardware-Kohärenzverfahren bereits sind.

Abkürzungsverzeichnis

BCET	Best-Case Execution Time
DRAM	Dynamic Random Access Memory
FIFO	First In First Out
HRT	Hard Real-Time
IMA	Integrated Modular Avionics
LRU	Least Recently Used
LSB	Least Significant Bit
MMU	Memory Management Unit
MESI	Modified-Exclusive-Shared-Invalid
MOESI	Modified-Owned-Exclusive-Shared-Invalid
MSB	Most Significant Bit
NUCA	Non-Uniform Cache Architecture
ODC²	On-Demand Coherent Cache
SRAM	Static Random Access Memory
WCET	Worst-Case Execution Time
WCET_{est}	Estimated Worst-Case Execution Time

Abbildungsverzeichnis

2.1	Aufbau einer Speicherhierarchie	8
2.2	Aufbau eines Cache-Blocks	9
2.3	Gliederung einer Cache-Adresse	10
2.4	LEON4 SPARC V8 Blockdiagramm	12
2.5	OCTEON III CN7XXX Blockdiagramm	13
2.6	Verbindungstypen	13
2.7	TilePro64 Blockdiagramm	14
2.8	Dateninkonsistenz	16
2.9	Inkohärenter Lesezugriff	19
2.10	Datenverlust bei Inkonsistenz	20
2.11	MESI Zustandsdiagramm	22
2.12	Laufzeitschranken	27
3.1	Basisarchitektur der Analyse	38
3.2	Cachezustand mit Alterswerten	40
3.3	Cachezustandswechsel nach Zugriff	41
3.4	Cachezustandswechsel nach Invalidierung	43
3.5	C^u Zustandswechsel nach unbekannter Invalidierung	43
3.6	C^l Zustandswechsel nach unbekannter Invalidierung	44
3.7	Blockierung eines Rechenkerns im TDMA-Bus	49
3.8	Aushungern eines Rechenkerns im TDMA-Bus	50
4.1	ODC ² Blockrahmen	66
4.2	False Sharing	68
4.3	Effekt der Invalidierung beim ODC ²	70
4.4	ODC ² Kontrollanweisungen	71
4.5	Einbettung von ODC ² Kontrollanweisungen	72
5.1	parMERASA Architektur	84

5.2	Laufzeit im Bussystem	92
5.3	Laufzeit von Matrix und FFT im Bussystem	92
5.4	Laufzeit von 3DPP im Bussystem	93
5.5	Laufzeit mit 4 Rechenkernen	94
5.6	Cache-Misses beim Benchmark Edges	95
5.7	$WCET_{est}$ mit 4 Rechenkernen	96
5.8	Laufzeit und $WCET_{est}$ bei Benchmark 3DPP	97
5.9	Anteil an Private-Mode und Shared-Mode	98
5.10	Anteil an Write-Through Zugriffen	98
5.11	Histogramm der Invalidierung beim ODC^2	100
5.12	Histogramm der Invalidierung bei <i>Cache Flush</i>	100
5.13	Laufzeit und $WCET_{est}$ bei Benchmark Matrix	102
5.14	Zugriffstypen bei Benchmark Matrix	103
5.15	Laufzeit und $WCET_{est}$ bei Benchmark FFT	104
5.16	Zugriffstypen bei Benchmark FFT	105
5.17	Laufzeit und $WCET_{est}$ bei Benchmark Smoothing	106
5.18	Zugriffstypen bei Benchmark Smoothing	107
5.19	Laufzeit und $WCET_{est}$ bei Benchmark 3DPP	108
5.20	Zugriffstypen bei Benchmark 3DPP	108
5.21	Variation der Cachegröße	109
5.22	Variation der Assoziativität	110
5.23	Cache-Misses bei Variation der Assoziativität	111
5.24	Variation der Cache-Zeilengröße	112
5.25	Cache-Misses bei Variation der Cache-Zeilengröße	112
7.1	Laufzeit mit 2 Rechenkernen	144
7.2	Laufzeit mit 4 Rechenkernen	144
7.3	Laufzeit mit 8 Rechenkernen	144
7.4	$WCET_{est}$ mit 2 Rechenkernen	145
7.5	$WCET_{est}$ mit 4 Rechenkernen	145
7.6	$WCET_{est}$ mit 8 Rechenkernen	145
7.7	Laufzeit und $WCET_{est}$ bei Benchmark Matrix	146
7.8	Laufzeit und $WCET_{est}$ bei Benchmark FFT	146
7.9	Laufzeit und $WCET_{est}$ bei Benchmark Dijkstra	146
7.10	Laufzeit und $WCET_{est}$ bei Benchmark Smoothing	147
7.11	Laufzeit und $WCET_{est}$ bei Benchmark Edges	147
7.12	Laufzeit und $WCET_{est}$ bei Benchmark 3DPP	147
7.13	Histogramm der Invalidierung beim ODC^2 mit 2 Rechenkernen . .	148
7.14	Histogramm der Invalidierung bei <i>Cache Flush</i> mit 2 Rechenkernen	148
7.15	Histogramm der Invalidierung beim ODC^2 mit 4 Rechenkernen . .	149
7.16	Histogramm der Invalidierung bei <i>Cache Flush</i> mit 4 Rechenkernen	149
7.17	Histogramm der Invalidierung beim ODC^2 mit 8 Rechenkernen . .	150

7.18	Histogramm der Invalidierung bei <i>Cache Flush</i> mit 8 Rechenkernen	150
7.19	Zugriffstypen bei Benchmark Matrix mit 2 Rechenkernen	151
7.20	Zugriffstypen bei Benchmark Matrix mit 4 Rechenkernen	151
7.21	Zugriffstypen bei Benchmark Matrix mit 8 Rechenkernen	151
7.22	Zugriffstypen bei Benchmark FFT mit 2 Rechenkernen	152
7.23	Zugriffstypen bei Benchmark FFT mit 4 Rechenkernen	152
7.24	Zugriffstypen bei Benchmark FFT mit 8 Rechenkernen	152
7.25	Zugriffstypen bei Benchmark Dijkstra mit 2 Rechenkernen	153
7.26	Zugriffstypen bei Benchmark Dijkstra mit 4 Rechenkernen	153
7.27	Zugriffstypen bei Benchmark Dijkstra mit 8 Rechenkernen	153
7.28	Zugriffstypen bei Benchmark Smoothing mit 2 Rechenkernen	154
7.29	Zugriffstypen bei Benchmark Smoothing mit 4 Rechenkernen	154
7.30	Zugriffstypen bei Benchmark Smoothing mit 8 Rechenkernen	154
7.31	Zugriffstypen bei Benchmark Edges mit 2 Rechenkernen	155
7.32	Zugriffstypen bei Benchmark Edges mit 4 Rechenkernen	155
7.33	Zugriffstypen bei Benchmark Edges mit 8 Rechenkernen	155
7.34	Zugriffstypen bei Benchmark 3DPP mit 2 Rechenkernen	156
7.35	Zugriffstypen bei Benchmark 3DPP mit 4 Rechenkernen	156
7.36	Zugriffstypen bei Benchmark 3DPP mit 8 Rechenkernen	156

Tabellenverzeichnis

5.1	Zugriffslatenzen in der Evaluation	87
5.2	Benchmarks der Evaluation	90

Literaturverzeichnis

- [1] ABSINT ANGEWANDTE INFORMATIK GMBH: *aiT - WCET-Analyse*. http://www.absint.com/ait/index_de.htm. Version: 2014
- [2] ADVE, S. V. ; GHARACHORLOO, K. : Shared Memory Consistency Models: A Tutorial. In: *Computer* 29 (1996), Dez., Nr. 12, S. 66–76. <http://dx.doi.org/10.1109/2.546611>. – DOI 10.1109/2.546611. – ISSN 0018–9162
- [3] AEROFLEX GAISLER AB: SPARC V8 32-bit Processor LEON3 / LEON3-FT CompanionCore Data Sheet / Aeroflex Gaisler AB. Version: March 2010. http://www.actel.com/ipdocs/LEON3_DS.pdf. – Forschungsbericht
- [4] AEROFLEX GAISLER AB: Dual Core LEON4 SPARC V8 Processor LEON4-ASIC-DEMO Data Sheet and User’s Manual / Aeroflex Gaisler AB. Version: May 2011. <http://www.gaisler.net/doc/leon4-asic-demo-ag-rev11.pdf>. – Forschungsbericht
- [5] AGARWAL, A. ; SIMONI, R. ; HENNESSY, J. ; HOROWITZ, M. : An Evaluation of Directory Schemes for Cache Coherence. In: *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. New York, NY, USA : ACM, 1998 (ISCA '98). – ISBN 1–58113–058–9, S. 353–362
- [6] AGROU, H. ; SAINRAT, P. ; GATTI, M. ; FAURA, D. ; TOILLON, P. : A design approach for predictable and efficient multi-core processor for avionics. In: *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, 2011. – ISSN 2155–7195, S. 7D3–1–7D3–11
- [7] AKESSON, B. ; GOOSSENS, K. ; RINGHOFER, M. : Predator: A predictable SDRAM memory controller. In: *Hardware/Software Codesign and*

- System Synthesis (CODES+ISSS), 2007 5th IEEE/ACM/IFIP International Conference on*, 2007, S. 251–256
- [8] ANDREI, A. ; ELES, P. ; PENG, Z. ; ROSEN, J. : Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In: *VLSI Design, 2008. VLSID 2008. 21st International Conference on*, 2008. – ISSN 1063–9667, S. 103–110
- [9] ARCHIBALD, J. ; BAER, J. L.: An Economical Solution to the Cache Coherence Problem. In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1984 (ISCA '84). – ISBN 0–8186–0538–3, S. 355–362
- [10] ARCHIBALD, J. ; BAER, J.-L. : Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. In: *ACM Trans. Comput. Syst.* 4 (1986), Sept., Nr. 4, S. 273–298. <http://dx.doi.org/10.1145/6513.6514>. – DOI 10.1145/6513.6514. – ISSN 0734–2071
- [11] ARTEMIS PROJECT EMC2: *EMC2 Project Homepage*. <http://www.artemis-emc2.eu/>. Version: 2014
- [12] ASHBY, T. ; DÍAZ, P. ; CINTRA, M. : Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters. In: *Computers, IEEE Transactions on* 60 (2011), April, Nr. 4, S. 472–483. <http://dx.doi.org/10.1109/TC.2010.155>. – DOI 10.1109/TC.2010.155. – ISSN 0018–9340
- [13] AUTOSAR: *Automotive Open System Architecture Release 4.0*. <http://www.autosar.org/specifications/release-40/>. Version: 2014
- [14] BALLABRIGA, C. ; CASSÉ, H. ; ROCHANGE, C. ; SAINRAT, P. : OTAWA: an Open Toolbox for Adaptive WCET Analysis. In: *The 8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2010)*
- [15] BANAKAR, R. ; STEINKE, S. ; LEE, B.-S. ; BALAKRISHNAN, M. ; MARWEDEL, P. : Scratchpad Memory : A Design Alternative for Cache On-chip memory in Embedded Systems. In: *CODES*. Estes Park (Colorado), May 2002
- [16] BENNETT, J. K. ; CARTER, J. B. ; ZWAENEPOEL, W. : Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. In: *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*. New York, NY, USA : ACM, 1990 (PPOPP '90). – ISBN 0–89791–350–7, S. 168–176

-
- [17] BENSALÉM, S. ; GOOSSENS, K. ; KIRSCH, C. ; OBERMAISSER, R. ; LEE, E. ; SIFAKIS, J. : Time-predictable and composable architectures for dependable embedded systems. In: *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, 2011, S. 351–352
- [18] BERG, C. ; ENGBLOM, J. ; WILHELM, R. : Requirements for and Design of a Processor with Predictable Timing. In: THIELE, L. (Hrsg.) ; WILHELM, R. (Hrsg.): *Perspectives Workshop: Design of Systems with Predictable Behaviour*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (Dagstuhl Seminar Proceedings 03471). – ISSN 1862–4405
- [19] BJERREGAARD, T. ; MAHADEVAN, S. : A survey of research and practices of Network-on-chip. In: *ACM Comput. Surv.* 38 (2006), Jun., Nr. 1. <http://dx.doi.org/http://doi.acm.org/10.1145/1132952.1132953>. – DOI <http://doi.acm.org/10.1145/1132952.1132953>. – ISSN 0360–0300
- [20] BMBF PROJECT ARAMIS: *ARAMiS Project Homepage*. <http://www.projekt-aramis.de/index.php>. Version: 2013
- [21] BOST, E. : Hardware Support for Robust Partitioning in Freescale QorIQ Multicore SoCs (P4080 and derivatives) / Freescale Semiconductor, Inc. Version: May 2013. http://cache.freescale.com/files/32bit/doc/white_paper/QORIQHSRPWP.pdf. – Forschungsbericht
- [22] BROMAN, D. ; ZIMMER, M. ; KIM, Y. ; KIM, H. ; CAI, J. ; SHRIVASTAVA, A. ; EDWARDS, S. ; LEE, E. : Precision timed infrastructure: Design challenges. In: *Electronic System Level Synthesis Conference (ESLsyn), 2013*, 2013, S. 1–6
- [23] BROWN, J. A. ; KUMAR, R. ; TULLSEN, D. : Proximity-aware directory-based coherence for multi-core processor architectures. In: *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA : ACM, 2007 (SPAA '07). – ISBN 978–1–59593–667–7, S. 126–134
- [24] CASSÉ, H. ; SAINRAT, P. : OTAWA, a framework for experimenting WCET computations. In: *3rd European Congress on Embedded Real-Time Software*. Toulouse, France, January 2006
- [25] CAVIUM: *OCTEON III CN7XXX Multi-Core MIPS64 Processors*. http://www.cavium.com/OCTEON-III_CN7XXX.html. Version: 2014

- [26] CENSIER, L. ; FEAUTRIER, P. : A New Solution to Coherence Problems in Multicache Systems. In: *Computers, IEEE Transactions on* C-27 (1978), dec., Nr. 12, S. 1112–1118. <http://dx.doi.org/10.1109/TC.1978.1675013>. – DOI 10.1109/TC.1978.1675013. – ISSN 0018–9340
- [27] CEZE, L. ; TUCK, J. ; MONTESINOS, P. ; TORRELLAS, J. : BulkSC: bulk enforcement of sequential consistency. In: TULLSEN, D. M. ; CALDER, B. (Hrsg.): *ISCA*, ACM, 2007. – ISBN 978–1–59593–706–3, S. 278–289
- [28] CHAIKEN, D. ; FIELDS, C. ; KURIHARA, K. ; AGARWAL, A. : Directory-based cache coherence in large-scale multiprocessors. In: *Computer* 23 (1990), june, Nr. 6, S. 49–58. <http://dx.doi.org/10.1109/2.55500>. – DOI 10.1109/2.55500. – ISSN 0018–9162
- [29] CHAIKEN, D. ; KUBIATOWICZ, J. ; AGARWAL, A. : LimitLESS Directories: A Scalable Cache Coherence Scheme. In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA : ACM, 1991 (ASPLOS IV). – ISBN 0–89791–380–9, S. 224–234
- [30] CHATTOPADHYAY, S. ; CHONG, L. K. ; ROYCHOUDHURY, A. ; KELTER, T. ; MARWEDEL, P. ; FALK, H. : A Unified WCET Analysis Framework for Multicore Platforms. In: *ACM Trans. Embed. Comput. Syst.* 13 (2014), Apr., Nr. 4s, S. 124:1–124:29. <http://dx.doi.org/10.1145/2584654>. – DOI 10.1145/2584654. – ISSN 1539–9087
- [31] CHEONG, H. ; VEIDENBAUM, A. : A cache coherence scheme with fast selective invalidation. In: *Computer Architecture, 1988. Conference Proceedings. 15th Annual International Symposium on*, 1988, S. 299–307
- [32] CHEONG, H. : Life span strategy: a compiler-based approach to cache coherence. In: *Proceedings of the 6th international conference on Supercomputing*. New York, NY, USA : ACM, 1992 (ICS '92). – ISBN 0–89791–485–6, S. 139–148
- [33] CHEONG, H. ; VEIDENBAUM, A. : A Version Control Approach to Cache Coherence. In: *Proceedings of the 3rd International Conference on Supercomputing*. New York, NY, USA : ACM, 1989 (ICS '89). – ISBN 0–89791–309–4, S. 322–330
- [34] COUSOT, P. ; COUSOT, R. : Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1977 (POPL '77), S. 238–252

- [35] COUSOT, P. ; HALBWACHS, N. : Automatic Discovery of Linear Restraints Among Variables of a Program. In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1978 (POPL '78), S. 84–96
- [36] CULLMANN, C. ; FERDINAND, C. ; GEBHARD, G. ; GRUND, D. ; MAIZA, C. ; REINEKE, J. ; TRIQUET, B. ; WEGENER, S. ; WILHELM, R. : Predictability Considerations in the Design of Multi-Core Embedded Systems. In: *Ingénieurs de l'Automobile* 807 (2010), September, S. 36–42. – ISSN 0020–1200
- [37] CYTRON, R. ; KARLOVSKY, S. ; MCAULIFFE, K. P.: Automatic Management of Programmable Caches. In: *ICPP (2)'88*, 1988, S. 229–238
- [38] DARNELL, E. ; KENNEDY, K. : Cache coherence using local knowledge. In: *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. New York, NY, USA : ACM, 1993 (Supercomputing '93). – ISBN 0–8186–4340–4, S. 720–729
- [39] DINECHIN, B. D. ; MASSAS, P. G. ; LAGER, G. ; LÄNGER, C. ; ORGOGOZO, B. ; REYBERT, J. ; STRUDEL, T. : A Distributed Run-Time Environment for the Kalray MPPA®-256 Integrated Manycore Processor. In: *Procedia Computer Science* 18 (2013), Nr. 0, S. 1654 – 1663. <http://dx.doi.org/http://dx.doi.org/10.1016/j.procs.2013.05.333>. – DOI <http://dx.doi.org/10.1016/j.procs.2013.05.333>. – ISSN 1877–0509. – 2013 International Conference on Computational Science
- [40] EDLER, J. ; GOTTLIEB, A. ; KRUSKAL, C. P. ; MCAULIFFE, K. P. ; RUDOLPH, L. ; SNIR, M. ; TELLER, P. J. ; WILSON, J. : Issues Related to MIMD Shared-memory Computers: The NYU Ultracomputer Approach. In: *Proceedings of the 12th Annual International Symposium on Computer Architecture*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1985 (ISCA '85). – ISBN 0–8186–0634–7, S. 126–135
- [41] EDWARDS, S. ; LEE, E. : The Case for the Precision Timed (PRET) Machine. In: *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, 2007. – ISSN 0738–100X, S. 264–265
- [42] EGGERS, S. J. ; KATZ, R. H.: Evaluating the Performance of Four Snooping Cache Coherency Protocols. In: *Proceedings of the 16th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1989 (ISCA '89). – ISBN 0–89791–319–1, S. 2–15

- [43] ENGBLOM, J. : Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In: *Real-Time Technology and Applications Symposium, 1999. Proceedings of the Fifth IEEE*, 1999. – ISSN 1080–1812, S. 46–55
- [44] EU FP7 PROJECT CERTAINTY: *CERTAINTY Project Homepage*. <http://www.certainty-project.eu/>. Version: 2012
- [45] EU FP7 PROJECT DREAMS: *DREAMS Project Homepage*. <http://www.dreams-project.eu/>. Version: 2013
- [46] EU FP7 PROJECT MULTIPARTES: *MultiPARTES Project Homepage*. <http://www.multipartes.eu/>. Version: 2014
- [47] EU FP7 PROJECT T-CREST: *T-CREST Project Homepage*. <http://www.t-crest.org/>. Version: 2014
- [48] FENSCH, C. ; CINTRA, M. : An OS-based alternative to full hardware coherence on tiled CMPs. In: *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, 2008. – ISSN 1530–0897, S. 355–366
- [49] FERDINAND, C. ; HECKMANN, R. ; LANGENBACH, M. ; MARTIN, F. ; SCHMIDT, M. ; THEILING, H. ; THESING, S. ; WILHELM, R. : Reliable and Precise WCET Determination for a Real-Life Processor. In: *Proceedings of the First International Workshop on Embedded Software*. London, UK, UK : Springer-Verlag, 2001 (EMSOFT '01). – ISBN 3–540–42673–6, S. 469–485
- [50] FERDINAND, C. ; WILHELM, R. : Efficient and Precise Cache Behavior Prediction for Real-Time Systems. In: *Real-Time Systems* 17 (1999), Nr. 2-3, S. 131–181. <http://dx.doi.org/10.1023/A:1008186323068>. – DOI 10.1023/A:1008186323068. – ISSN 0922–6443
- [51] GERDES, M. ; KLUGE, F. ; UNGERER, T. ; ROCHANGE, C. ; SAINRAT, P. : Time analysable synchronisation techniques for parallelised hard real-time applications. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012. – ISSN 1530–1591, S. 671–676
- [52] GHARACHORLOO, K. ; LENOSKI, D. ; LAUDON, J. ; GIBBONS, P. ; GUPTA, A. ; HENNESSY, J. : Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1990 (ISCA '90). – ISBN 0–89791–366–3, S. 15–26

- [53] GOODMAN, J. R.: Using Cache Memory to Reduce Processor-memory Traffic. In: *Proceedings of the 10th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1983 (ISCA '83). – ISBN 0–89791–101–6, S. 124–131
- [54] GOODMAN, J. R.: *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department, 1991
- [55] HANSSON, A. ; GOOSSENS, K. ; BEKOOIJ, M. ; HUISKEN, J. : CoMPSoC: A Template for Composable and Predictable Multi-processor System on Chips. In: *ACM Trans. Des. Autom. Electron. Syst.* 14 (2009), Jan., Nr. 1, S. 2:1–2:24. <http://dx.doi.org/10.1145/1455229.1455231>. – DOI 10.1145/1455229.1455231. – ISSN 1084–4309
- [56] HARDAVELLAS, N. ; FERDMAN, M. ; FALSAFI, B. ; AILAMAKI, A. : Reactive NUCA: near-optimal block placement and replication in distributed caches. In: *SIGARCH Comput. Archit. News* 37 (2009), Jun., Nr. 3, S. 184–195. <http://dx.doi.org/10.1145/1555815.1555779>. – DOI 10.1145/1555815.1555779. – ISSN 0163–5964
- [57] HARDY, D. ; PIQUET, T. ; PUAUT, I. : Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In: *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, 2009 (RTSS '09). – ISBN 978–0–7695–3875–4, S. 68–77
- [58] HARRIS, T. ; CRISTAL, A. ; UNSAL, O. ; AYGAUDE, E. ; GAGLIARDI, F. ; SMITH, B. ; VALERO, M. : Transactional Memory: An Overview. In: *Micro, IEEE* 27 (2007), May, Nr. 3, S. 8–29. <http://dx.doi.org/10.1109/MM.2007.63>. – DOI 10.1109/MM.2007.63. – ISSN 0272–1732
- [59] HECKMANN, R. ; LANGENBACH, M. ; THESING, S. ; WILHELM, R. : The influence of processor architecture on the design and the results of WCET tools. In: *Proceedings of the IEEE* 91 (2003), July, Nr. 7, S. 1038–1054. <http://dx.doi.org/10.1109/JPROC.2003.814618>. – DOI 10.1109/JPROC.2003.814618. – ISSN 0018–9219
- [60] HERLIHY, M. ; MOSS, J. E. B.: Transactional Memory: Architectural Support for Lock-free Data Structures. In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1993 (ISCA '93). – ISBN 0–8186–3810–9, S. 289–300
- [61] HILL, M. D.: Multiprocessors Should Support Simple Memory-Consistency Models. In: *Computer* 31 (1998), Aug., Nr. 8, S. 28–34. <http://dx.doi.org/10.1109/2.707614>. – DOI 10.1109/2.707614. – ISSN 0018–9162

- [62] HILL, M. D. ; LARUS, J. R. ; REINHARDT, S. K. ; WOOD, D. A.: Co-operative Shared Memory: Software and Hardware for Scalable Multiprocessors. In: *ACM Trans. Comput. Syst.* 11 (1993), Nov., Nr. 4, S. 300–318. <http://dx.doi.org/10.1145/161541.161544>. – DOI 10.1145/161541.161544. – ISSN 0734–2071
- [63] HILL, M. ; MARTY, M. : Amdahl’s Law in the Multicore Era. In: *Computer* 41 (2008), July, Nr. 7, S. 33–38. <http://dx.doi.org/10.1109/MC.2008.209>. – DOI 10.1109/MC.2008.209. – ISSN 0018–9162
- [64] HUYCK, P. : ARINC 653 and multi-core microprocessors - Considerations and potential impacts. In: *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, 2012. – ISSN 2155–7195, S. 6B4–1–6B4–7
- [65] INFINEON TECHNOLOGIES AG: Highly Integrated and Performance Optimized 32-bit Microcontrollers for Automotive and Industrial Applications / Infineon Technologies AG. Version: Februar 2014. http://www.infineon.com/dgdl/TriCore_Family_BR-2014.pdf?folderId=db3a304412b407950112b409ae660342&fileId=db3a30431f848401011fc664882a7648. – Forschungsbericht
- [66] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO 26262-5:2011 - Road vehicles - Functional safety - Part 5: Product development at the hardware level*. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=51360. Version: 2011
- [67] IQBAL, S. M. Z. ; LIANG, Y. ; GRAHN, H. : ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems. In: *IEEE Comput. Archit. Lett.* 9 (2010), Jul., Nr. 2, S. 45–48. <http://dx.doi.org/10.1109/L-CA.2010.14>. – DOI 10.1109/L-CA.2010.14. – ISSN 1556–6056
- [68] JAHR, R. ; GERDES, M. ; UNGERER, T. ; OZAKTAS, H. ; ROCHANGE, C. ; ZAYKOV, P. : Effects of structured parallelism by parallel design patterns on embedded hard real-time systems. In: *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, 2014, S. 1–10
- [69] KATZ, R. H. ; EGGERS, S. J. ; WOOD, D. A. ; PERKINS, C. L. ; SHELDON, R. G.: Implementing a cache consistency protocol. In: *SIGARCH Comput. Archit. News* 13 (1985), Jun., Nr. 3, S. 276–283. <http://dx.doi.org/10.1145/327070.327237>. – DOI 10.1145/327070.327237. – ISSN 0163–5964

- [70] KELM, J. H. ; JOHNSON, D. R. ; TUOHY, W. ; LUMETTA, S. S. ; PATEL, S. J.: Cohesion: a hybrid memory model for accelerators. In: *SIGARCH Comput. Archit. News* 38 (2010), Jun., Nr. 3, S. 429–440. <http://dx.doi.org/10.1145/1816038.1816019>. – DOI 10.1145/1816038.1816019. – ISSN 0163–5964
- [71] KELTER, T. ; FALK, H. ; MARWEDEL, P. ; CHATTOPADHYAY, S. ; ROY-CHOUDHURY, A. : Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds. In: *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, 2011. – ISSN 1068–3070, S. 3–12
- [72] KELTER, T. ; HARDE, T. ; MARWEDEL, P. ; FALK, H. : Evaluation of resource arbitration methods for multi-core real-time systems. In: MAIZA, C. (Hrsg.): *13th International Workshop on Worst-Case Execution Time Analysis* Bd. 30. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (OpenAccess Series in Informatics (OASICS)). – ISBN 978–3–939897–54–5, 1–10
- [73] KESSLER, R. E.: Multicore and the 32 Core Cavium OCTEON II 68xx / Cavium Inc. Version: Februar 2013. https://www.ece.cmu.edu/~calcm/lib/exe/fetch.php?media=seminars:2013_spring_cavium.pdf. – Forschungsbericht
- [74] KHAN, O. ; HOFFMANN, H. ; LIS, M. ; HIJAZ, F. ; AGARWAL, A. ; DEVADAS, S. : ARCC: A Case for an Architecturally Redundant Cache-coherence Architecture for Large Multicores. In: *Proceedings of the 2011 IEEE 29th International Conference on Computer Design*. Washington, DC, USA : IEEE Computer Society, 2011 (ICCD '11). – ISBN 978–1–4577–1953–0, S. 411–418
- [75] KLIMA, S. : *Zusammenstellung und Evaluierung einer Benchmark - Suite für eingebettete Systeme*, Universität Augsburg, Bachelorarbeit, 2009
- [76] KUMAR, R. ; ZYUBAN, V. ; TULLSEN, D. M.: Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In: *SIGARCH Comput. Archit. News* 33 (2005), Mai, Nr. 2, S. 408–419. <http://dx.doi.org/10.1145/1080695.1070004>. – DOI 10.1145/1080695.1070004. – ISSN 0163–5964
- [77] LACKORZYNSKI, A. ; ENGEL, B. ; VÖLP, M. : Predictable Coherent Caching with Incoherent Caches. In: *15 th Real-Time Linux Workshop, Lugano-Manno, Switzerland, October 2013*
- [78] LAI, A.-C. ; FALSAFI, B. : Selective, Accurate, and Timely Self-invalidation Using Last-touch Prediction. In: *Proceedings of the 27th*

- Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 2000 (ISCA '00). – ISBN 1–58113–232–8, S. 139–148
- [79] LAKIS, E. ; SCHOEBERL, M. : An SDRAM controller for real-time systems. In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, 2013, S. 1–8
- [80] LAMPORT, L. : How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. In: *IEEE Trans. Comput.* 28 (1979), Sept., Nr. 9, S. 690–691. <http://dx.doi.org/10.1109/TC.1979.1675439>. – DOI 10.1109/TC.1979.1675439. – ISSN 0018–9340
- [81] LAUER, H. C.: Bulk Core in a 360/67 Time-sharing System. In: *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*. New York, NY, USA : ACM, 1967 (AFIPS '67 (Fall)), S. 601–609
- [82] LEBECK, A. R. ; WOOD, D. A.: Dynamic Self-invalidation: Reducing Coherence Overhead in Shared-memory Multiprocessors. In: *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1995 (ISCA '95). – ISBN 0–89791–698–0, S. 48–59
- [83] LEBLANC, T. J. ; SCOTT, M. L. ; BROWN, C. M.: Large-scale Parallel Programming: Experience with BBN Butterfly Parallel Processor. In: *Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems*. New York, NY, USA : ACM, 1988 (PPEALS '88). – ISBN 0–89791–276–4, S. 161–172
- [84] LEE, H. ; CHO, S. ; CHILDERS, B. R.: CloudCache: Expanding and Shrinking Private Caches. In: *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. Washington, DC, USA : IEEE Computer Society, 2011 (HPCA '11). – ISBN 978–1–4244–9432–3, S. 219–230
- [85] LEE, R. L. ; YEW, P.-C. ; LAWRIE, D. H.: Multiprocessor cache design considerations. In: *Proceedings of the 14th annual international symposium on Computer architecture* ACM, 1987, S. 253–262
- [86] LENOSKI, D. ; LAUDON, J. ; GHARACHORLOO, K. ; GUPTA, A. ; HENNESSY, J. : The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In: *SIGARCH Comput. Archit. News* 18 (1990), Mai, Nr. 2SI, S. 148–159. <http://dx.doi.org/10.1145/325096.325132>. – DOI 10.1145/325096.325132. – ISSN 0163–5964

- [87] LESAGE, B. ; PUAUT, I. ; SEZNEC, A. : PRETI: Partitioned Real-time Shared Cache for Mixed-criticality Real-time Systems. In: *Proceedings of the 20th International Conference on Real-Time and Network Systems*. New York, NY, USA : ACM, 2012 (RTNS '12). – ISBN 978–1–4503–1409–1, S. 171–180
- [88] LICKLY, B. ; LIU, I. ; KIM, S. ; PATEL, H. D. ; EDWARDS, S. A. ; LEE, E. A.: Predictable Programming on a Precision Timed Architecture. In: *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY, USA : ACM, 2008 (CASES '08). – ISBN 978–1–60558–469–0, S. 137–146
- [89] LIU, I. ; REINEKE, J. ; LEE, E. : A PRET architecture supporting concurrent programs with composable timing properties. In: *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, 2010. – ISSN 1058–6393, S. 2111–2115
- [90] LUNDQVIST, T. ; STENSTRÖM, P. : A method to improve the estimated worst-case performance of data caching. In: *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, 1999, S. 255–262
- [91] MARWEDEL, P. : *Eingebettete Systeme*. Springer London, Limited <http://books.google.de/books?id=94L961HZ51YC>. – ISBN 9783540340492
- [92] MIN, S. L. ; BAER, J.-L. : A Timestamp-based Cache Coherence Scheme. In: *ICPP (1)*, 1989, S. 23–32
- [93] MOORE, G. : Cramming More Components Onto Integrated Circuits. In: *Proceedings of the IEEE* 86 (1998), Jan, Nr. 1, S. 82–85. <http://dx.doi.org/10.1109/JPROC.1998.658762>. – DOI 10.1109/JPROC.1998.658762. – ISSN 0018–9219
- [94] MUKHERJEE, S. S. ; HILL, M. D.: Using Prediction to Accelerate Coherence Protocols. In: *Proceedings of the 25th Annual International Symposium on Computer Architecture*. Washington, DC, USA : IEEE Computer Society, 1998 (ISCA '98). – ISBN 0–8186–8491–7, S. 179–190
- [95] NOWOTSCH, J. ; PAULITSCH, M. : Leveraging Multi-core Computing Architectures in Avionics. In: *Dependable Computing Conference (EDCC), 2012 Ninth European*, 2012, S. 132–143
- [96] O'BOYLE, M. F. P. ; FORD, R. W. ; STOHR, E. A.: Towards General and Exact Distributed Invalidation. In: *J. Parallel Distrib. Comput.* 63

- (2003), Nov., Nr. 11, S. 1123–1137. <http://dx.doi.org/10.1016/j.jpdc.2003.07.007>. – DOI 10.1016/j.jpdc.2003.07.007. – ISSN 0743–7315
- [97] PAOLIERI, M. ; QUIÑONES, E. ; CAZORLA, F. J. ; BERNAT, G. ; VALERO, M. : Hardware Support for WCET Analysis of Hard Real-time Multicore Systems. In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 2009 (ISCA '09). – ISBN 978–1–60558–526–0, S. 57–68
- [98] PAPAMARCOS, M. S. ; PATEL, J. H.: A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories. In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1984 (ISCA '84). – ISBN 0–8186–0538–3, S. 348–354
- [99] PFISTER, G. F. ; BRANTLEY, W. C. ; GEORGE, D. A. ; HARVEY, S. L. ; KLEINFELDER, W. J. ; MCAULIFFE, K. P. ; MELTON, E. S. ; NORTON, V. A. ; WEISS, J. : The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In: *ICPP*, 1985, S. 764–771
- [100] PITTER, C. ; SCHOEBERL, M. : A Real-time Java Chip-multiprocessor. In: *ACM Trans. Embed. Comput. Syst.* 10 (2010), Aug., Nr. 1, S. 9:1–9:34. <http://dx.doi.org/10.1145/1814539.1814548>. – DOI 10.1145/1814539.1814548. – ISSN 1539–9087
- [101] PUFFITSCH, W. : Data Caching, Garbage Collection, and the Java Memory Model. In: *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*. New York, USA : ACM, 2009 (JTRES '09). – ISBN 978–1–60558–732–5, S. 90–99
- [102] PUGSLEY, S. H. ; SPJUT, J. B. ; NELLANS, D. W. ; BALASUBRAMONIAN, R. : SWEL: hardware cache coherence protocols to map shared data onto shared caches. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. New York, USA : ACM, 2010 (PACT '10). – ISBN 978–1–4503–0178–7, S. 465–476
- [103] PYKA, A. ; ROHDE, M. ; UHRIG, S. : Performance Evaluation of the Time Analysable On-Demand Coherent Cache. In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, 2013, S. 1887–1892
- [104] PYKA, A. ; ROHDE, M. ; UHRIG, S. : Extended performance analysis of the time predictable on-demand coherent data cache for multi- and many-core systems. In: *International Conference on Embedded Computer*

- Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014*, 2014, S. 107–114
- [105] PYKA, A. ; ROHDE, M. ; FERNANDES, J. ; UHRIG, S. : On-Demand Coherent Cache for Parallelised Hard Real-Time Applications. In: *Proceedings of the Work-in-Progress Session of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013), Paris, France*
- [106] PYKA, A. ; ROHDE, M. ; UHRIG, S. : A Real-time Capable First-level Cache for Multi-cores. In: *Workshop on High Performance and Real-time Embedded Systems (HiRES) in conjunction with HiPEAC'13, Berlin, Germany*
- [107] PYKA, A. ; ROHDE, M. ; UHRIG, S. : A real-time capable coherent data cache for multicores. In: *Concurrency and Computation: Practice and Experience* 26 (2014), Nr. 6, S. 1342–1354. <http://dx.doi.org/10.1002/cpe.3172>. – DOI 10.1002/cpe.3172. – ISSN 1532–0634
- [108] PYKA, A. ; ROHDE, M. ; ZAYKOV, P. G. ; UHRIG, S. : Case Study: On-Demand Coherent Cache for Avionic Applications. In: *2nd Workshop on High-performance and Real-time Embedded Systems (HiRES 2014), Vienna, Austria*
- [109] PYKA, A. ; TADROS, L. ; UHRIG, S. ; CASSÉ, H. ; OZAKTAS, H. ; ROCHANGE, C. : WCET Analysis of Parallel Benchmarks using On-Demand Coherent Cache. In: *3rd Workshop on High-performance and Real-time Embedded Systems (HiRES 2015), Amsterdam, the Netherlands*
- [110] QUIÑONES, E. : Deliverable 5.3 - Predictable parMERA-SA Multicore Processor / Barcelona Supercomputing Center. Version: September 2013. http://www.parmerasa.eu/files/deliverables/Deliverable_5_3_update.pdf. – Forschungsbericht
- [111] RAO, L. ; WANG, Q. ; LIU, X. ; WANG, Y. : Analysis of TDMA crossbar real-time switch design for AFDX networks. In: *INFOCOM, 2012 Proceedings IEEE*, 2012. – ISSN 0743–166X, S. 2462–2470
- [112] RAPITA SYSTEMS: *RapiTime*. <http://www.rapitasystems.com/products/rapitime>. Version: 2014
- [113] REINEKE, J. ; GRUND, D. ; BERG, C. ; WILHELM, R. : Timing Predictability of Cache Replacement Policies. In: *Real-Time Syst.* 37 (2007), Nov., Nr. 2, S. 99–122. <http://dx.doi.org/10.1007/s11241-007-9032-3>. – DOI 10.1007/s11241-007-9032-3. – ISSN 0922–6443

- [114] REINEKE, J. ; WACHTER, B. ; THESING, S. ; WILHELM, R. ; POLIAN, I. ; EISINGER, J. ; BECKER, B. : A Definition and Classification of Timing Anomalies. In: MUELLER, F. (Hrsg.): *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)* Bd. 4. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (OpenAccess Series in Informatics (OASICS)). – ISBN 978–3–939897–03–3
- [115] ROS, A. ; ACACIO, M. ; GARCIA, J. : DiCo-CMP: Efficient cache coherency in tiled CMP architectures. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008. – ISSN 1530–2075, S. 1–11
- [116] ROSEÁN, J. ; ANDREI, A. ; ELES, P. ; PENG, Z. : Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In: *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, 2007. – ISSN 1052–8725, S. 49–60
- [117] RUDOLPH, L. ; SEGALL, Z. : Dynamic Decentralized Cache Schemes for Mimd Parallel Processors. In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1984 (ISCA '84). – ISBN 0–8186–0538–3, S. 340–347
- [118] SANDHU, H. S. ; GAMSA, B. ; ZHOU, S. : The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA : ACM, 1993 (PPOPP '93). – ISBN 0–89791–589–5, S. 229–238
- [119] SCHOEBERL, M. : Time-predictable Cache Organization. In: *Future Dependable Distributed Systems, 2009 Software Technologies for*, 2009, S. 11–16
- [120] SCHOEBERL, M. : A Time-Predictable Object Cache. In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, 2011. – ISSN 1555–0885, S. 99–105
- [121] SCHOEBERL, M. ; BRANDNER, F. ; SPARSØ, J. ; KASAPAKI, E. : A Statically Scheduled Time-Division-Multiplexed Network-on-Chip for Real-Time Systems. In: *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, 2012, S. 152–160
- [122] SCHOEBERL, M. ; HILBER, P. : Design and Implementation of Real-Time Transactional Memory. In: *Field Programmable Logic and Applications*

- (FPL), 2010 International Conference on, 2010. – ISSN 1946–1488, S. 279–284
- [123] SCHOEBERL, M. ; BRANDNER, F. ; VITEK, J. : RTTM: Real-time Transactional Memory. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2010 (SAC '10). – ISBN 978–1–60558–639–7, S. 326–333
- [124] SMITH, A. J.: CPU Cache Consistency with Software Support and using One Time Identifiers. In: *Proceedings of the Pacific Computer Communications '85*, 1985, S. 153–161
- [125] STEFAN, R. ; MOLNOS, A. ; AMBROSE, A. ; GOOSSENS, K. : A TDM NoC supporting QoS, multicast, and fast connection set-up. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012. – ISSN 1530–1591, S. 1283–1288
- [126] STENSTRÖM, P. : A Survey of Cache Coherence Schemes for Multiprocessors. In: *Computer* 23 (1990), Jun., Nr. 6, S. 12–24. <http://dx.doi.org/10.1109/2.55497>. – DOI 10.1109/2.55497. – ISSN 0018–9162
- [127] STENSTRÖM, P. ; BRORSSON, M. ; SANDBERG, L. : An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1993 (ISCA '93). – ISBN 0–8186–3810–9, S. 109–118
- [128] SUHENDRA, V. ; MITRA, T. : Exploring locking - partitioning for predictable shared caches on multi-cores. In: *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, 2008. – ISSN 0738–100X, S. 300–303
- [129] SWEAZEY, P. ; SMITH, A. J.: A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In: *Proceedings of the 13th Annual International Symposium on Computer Architecture*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1986 (ISCA '86). – ISBN 0–8186–0719–X, S. 414–423
- [130] TANG, C. K.: Cache System Design in the Tightly Coupled Multiprocessor System. In: *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*. New York, NY, USA : ACM, 1976 (AFIPS '76), S. 749–753
- [131] THE SOCLIB PROJECT: *SoCLib Homepage*. <http://www.soclib.fr/>. Version: 2014

- [132] THIELE, L. ; WILHELM, R. : Design for Timing Predictability. In: *Real-Time Syst.* 28 (2004), Nov., Nr. 2-3, 157–177. <http://dx.doi.org/10.1023/B:TIME.0000045316.66276.6e>. – DOI 10.1023/B:TIME.0000045316.66276.6e. – ISSN 0922–6443
- [133] TILERA CORPORATION: TILE Pro64 Processor Product Brief / Tiler Corporation. Version: 2011. http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf. – Forschungsbericht
- [134] TILERA CORPORATION: Tile Processor Architecture Overview For The TILEPro Series / Tiler Corporation. Version: November 2013. <http://www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf> (Release 1.2). – Forschungsbericht
- [135] UHRIG, S. : Evaluation of Different Multithreaded and Multicore Processor Configurations for SoPC. In: *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*. Berlin, Heidelberg : Springer-Verlag, 2009 (SAMOS '09). – ISBN 978–3–642–03137–3, S. 68–77
- [136] UHRIG, S. : Deliverable 5.5 - parMERASA Multi-core Processor Refinement and Optimisation / Barcelona Supercomputing Center. Version: March 2014. http://www.parmerasa.eu/files/deliverables/Deliverable_5_5.pdf. – Forschungsbericht
- [137] UHRIG, S. ; TADROS, L. ; PYKA, A. : MESI-based Cache Coherence for Hard Real-time Multicore Systems. In: *28th GI/ITG International Conference on Architecture of Computing Systems, Porto, Portugal*
- [138] UNGERER, T. ; BRADATSCH, C. ; GERDES, M. ; KLUGE, F. ; JAHR, R. ; MISCHÉ, J. ; FERNANDES, J. ; ZAYKOV, P. ; PETROV, Z. ; BODDEKER, B. ; KEHR, S. ; REGLER, H. ; HUGL, A. ; ROCHANGE, C. ; OZAKTAS, H. ; CASSÉ, H. ; BONENFANT, A. ; SAINRAT, P. ; BROSTER, I. ; LAY, N. ; GEORGE, D. ; QUINONES, E. ; PANIC, M. ; ABELLA, J. ; CAZORLA, F. ; UHRIG, S. ; ROHDE, M. ; PYKA, A. : parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. In: *Digital System Design (DSD), 2013 Euromicro Conference on*, 2013, S. 363–370
- [139] UNGERER, T. ; BRADATSCH, C. ; FRIEB, M. ; KLUGE, F. ; MISCHÉ, J. ; STEGMEIER, A. ; JAHR, R. ; GERDES, M. ; ZAYKOV, P. ; MATUSOVA, L. ; LI, Z. J. J. ; PETROV, Z. ; BÖDDEKER, B. ; KEHR, S. ; REGLER, H. ; HUGL, A. ; ROCHANGE, C. ; OZAKTAS, H. ; CASSÉ, H. ; BONENFANT,

- A. ; SAINRAT, P. ; LAY, N. ; GEORGE, D. ; BROSTER, I. ; QUIÑONES, E. ; PANIC, M. ; ABELLA, J. ; HERNANDEZ, C. ; CAZORLA, F. ; UHRIG, S. ; ROHDE, M. ; PYKA, A. : Experiences and Results of Parallelisation of Industrial Hard Real-time Applications for the parMERASA Multi-core. In: *3rd Workshop on High-performance and Real-time Embedded Systems (HiRES 2015), Amsterdam, the Netherlands*
- [140] UNGERER, T. ; CAZORLA, F. ; SAINRAT, P. ; BERNAT, G. ; PETROV, Z. ; CASSÉ, H. ; ROCHANGE, C. ; QUINONES, E. ; UHRIG, S. ; GERDES, M. ; GULIASHVILI, I. ; HOUSTON, M. ; KLUGE, F. ; METZLAFF, S. ; MISCHÉ, J. ; PAOLIERI, M. ; WOLF, J. : MERASA: Multi-Core Execution of Hard Real-Time Applications Supporting Analysability. In: *IEEE Micro, European Multicore Processing Projects* 30 (2010), septembre, Nr. 5, S. 66–75
- [141] VEIDENBAUM, A. V.: A Compiler-Assisted Cache Coherence Solution for Multiprocessors. In: *ICPP'86*, 1986, S. 1029–1036
- [142] VERA, X. ; LISPER, B. ; XUE, J. : Data cache locking for higher program predictability. In: *SIGMETRICS Perform. Eval. Rev.* 31 (2003), Jun., Nr. 1, S. 272–282. <http://dx.doi.org/10.1145/885651.781062>. – DOI 10.1145/885651.781062. – ISSN 0163–5999
- [143] WARD, B. ; HERMAN, J. ; KENNA, C. ; ANDERSON, J. : Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms. In: *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, 2013, S. 157–167
- [144] WHITHAM, J. ; AUDSLEY, N. : Implementing Time-predictable Load and Store Operations. In: *Proceedings of the Seventh ACM International Conference on Embedded Software*. New York, NY, USA : ACM, 2009 (EMSOFT '09). – ISBN 978–1–60558–627–4, S. 265–274
- [145] WILHELM, R. ; REINEKE, J. : Embedded systems: Many cores - Many problems. In: *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*, 2012, S. 176–180
- [146] WILHELM, R. : Determining Bounds on Execution Times. In: ZURAWSKI, R. (Hrsg.): *Embedded Systems Design and Verification*. CRC Press, 2009. – ISBN 978–1–4398–0763–7, S. 9
- [147] WILHELM, R. ; ENGBLOM, J. ; ERMEDAHL, A. ; HOLSTI, N. ; THESING, S. ; WHALLEY, D. ; BERNAT, G. ; FERDINAND, C. ; HECKMANN, R. ; MITRA, T. ; MUELLER, F. ; PUAUT, I. ; PUSCHNER, P. ; STASCHULAT,

- J. ; STENSTRÖM, P. : The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. In: *ACM Trans. Embed. Comput. Syst.* 7 (2008), Mai, Nr. 3, S. 36:1–36:53. <http://dx.doi.org/10.1145/1347375.1347389>. – DOI 10.1145/1347375.1347389. – ISSN 1539–9087
- [148] YAN, J. ; ZHANG, W. : WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In: *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA : IEEE Computer Society, 2008 (RTAS '08). – ISBN 978-0-7695-3146-5, S. 80–89
- [149] YEN, W. ; YEN, D. ; FU, K.-S. : Data Coherence Problem in a Multicache System. In: *Computers, IEEE Transactions on* C-34 (1985), Jan, Nr. 1, S. 56–65. <http://dx.doi.org/10.1109/TC.1985.1676515>. – DOI 10.1109/TC.1985.1676515. – ISSN 0018–9340
- [150] ZEBCHUK, J. ; SRINIVASAN, V. ; QURESHI, M. ; MOSHOVOS, A. : A Tagless Coherence Directory. In: *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009. – ISSN 1072–4451, S. 423–434
- [151] ZHANG, M. ; ASANOVIC, K. : Victim Replication: Maximizing Capacity While Hiding Wire Delay in Tiled Chip Multiprocessors. In: *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*. Washington, DC, USA : IEEE Computer Society, 2005 (ISCA '05). – ISBN 0-7695-2270-X, S. 336–345

Kapitel 7

Anhang

Dieser Anhang enthält sämtliche Abbildungen, die für die Evaluation in dieser Arbeit angefertigt wurden und ermöglicht einen kompakten Überblick über die Evaluationsergebnisse. Abbildungen, die für die Veranschaulichung der Resultate erforderlich sind, werden zusätzlich in Kapitel 5 dargestellt und kommentiert. Der Vollständigkeit halber und um die Auswertung der Evaluation lesbar und übersichtlich zu gestalten, wird auf eine Darstellung der übrigen Abbildungen in Kapitel 5 verzichtet und stattdessen auf diesen Anhang verwiesen.

Anhang 1: Laufzeitsimulation

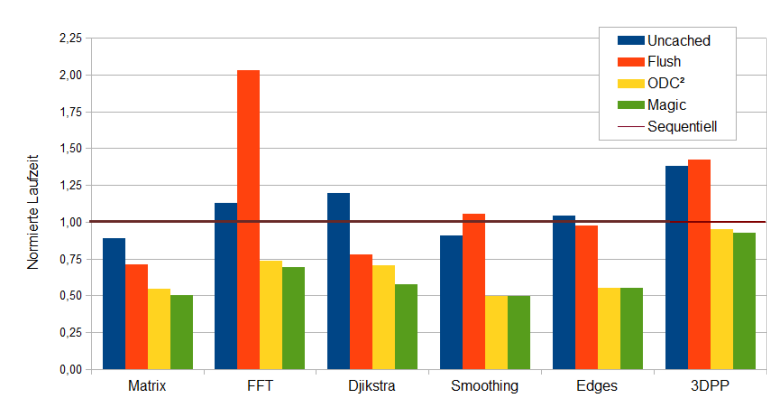


Abbildung 7.1: Auf sequentielle Ausführung normierte Laufzeit bei paralleler Ausführung mit 2 Rechenkernen.

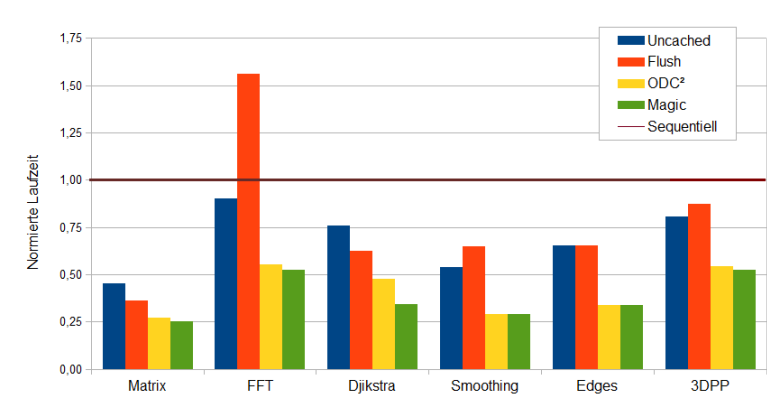


Abbildung 7.2: Auf sequentielle Ausführung normierte Laufzeit bei paralleler Ausführung mit 4 Rechenkernen.

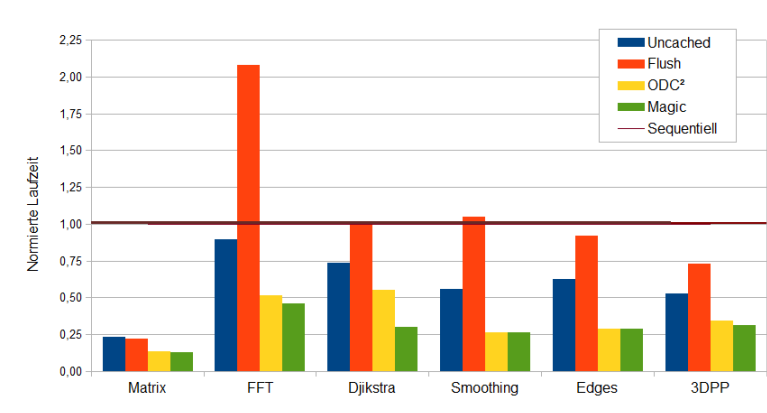


Abbildung 7.3: Auf sequentielle Ausführung normierte Laufzeit bei paralleler Ausführung mit 8 Rechenkernen.

Anhang 2: WCET-Abschätzung

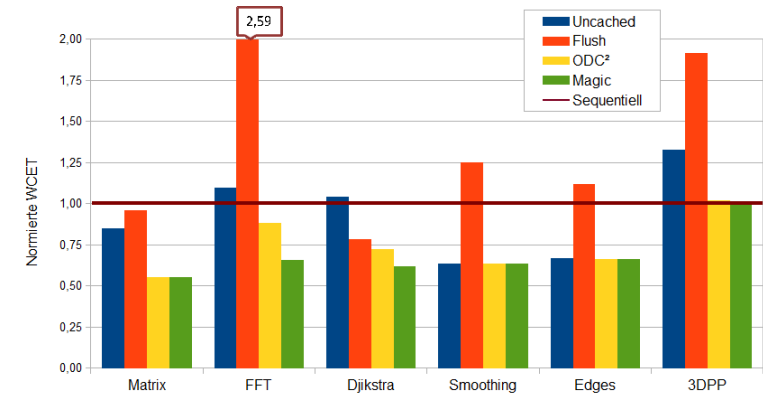


Abbildung 7.4: Auf sequentielle Ausführung normierte $WCET_{est}$ bei paralleler Ausführung mit 2 Rechenkernen.

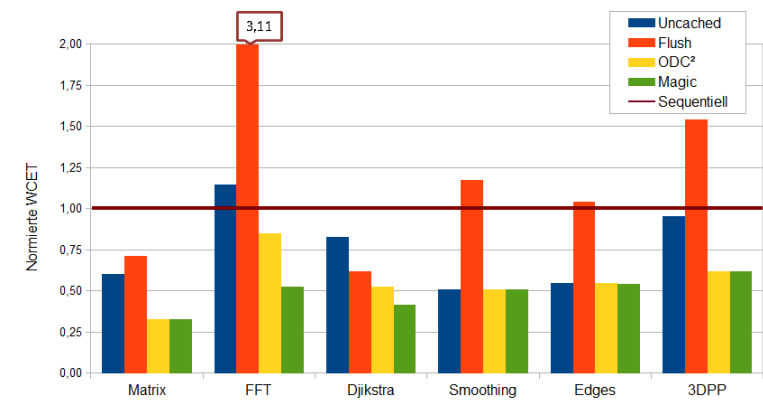


Abbildung 7.5: Auf sequentielle Ausführung normierte $WCET_{est}$ bei paralleler Ausführung mit 4 Rechenkernen.

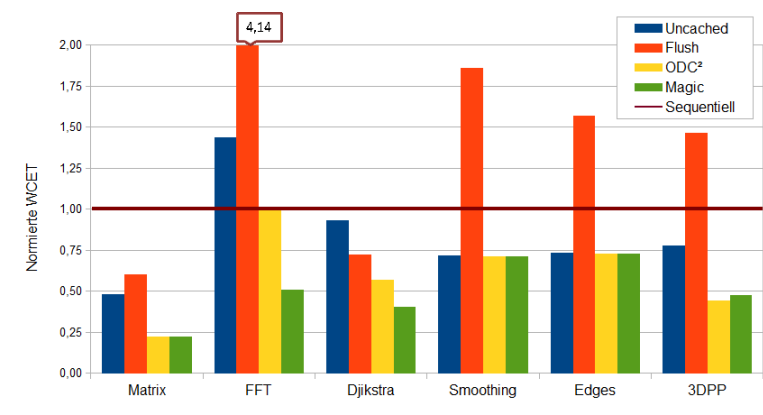


Abbildung 7.6: Auf sequentielle Ausführung normierte $WCET_{est}$ bei paralleler Ausführung mit 8 Rechenkernen.

Anhang 3: Skalierung der Laufzeit und WCET

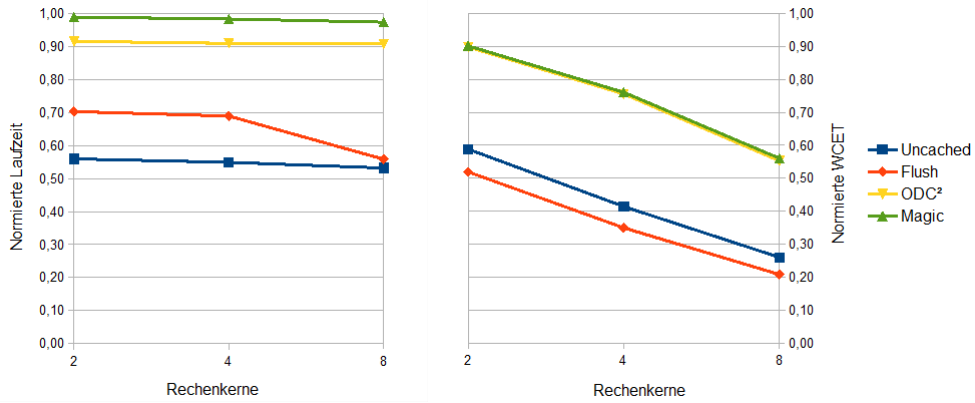


Abbildung 7.7: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von Matrix.

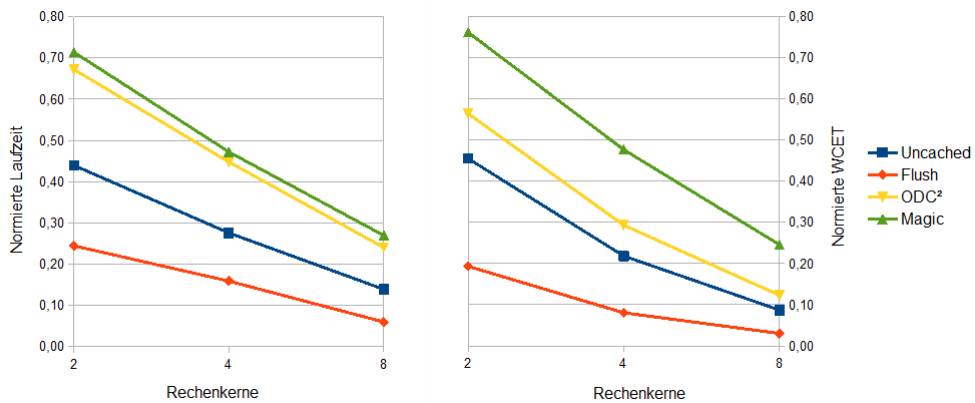


Abbildung 7.8: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von FFT.

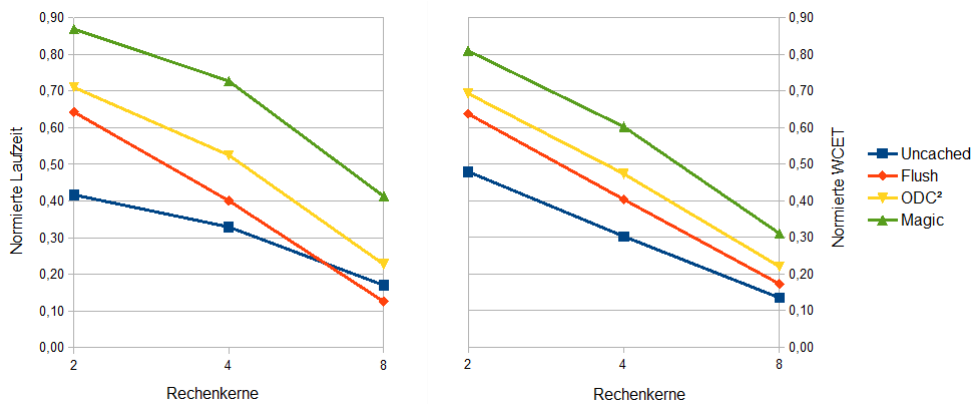


Abbildung 7.9: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von Dijkstra.

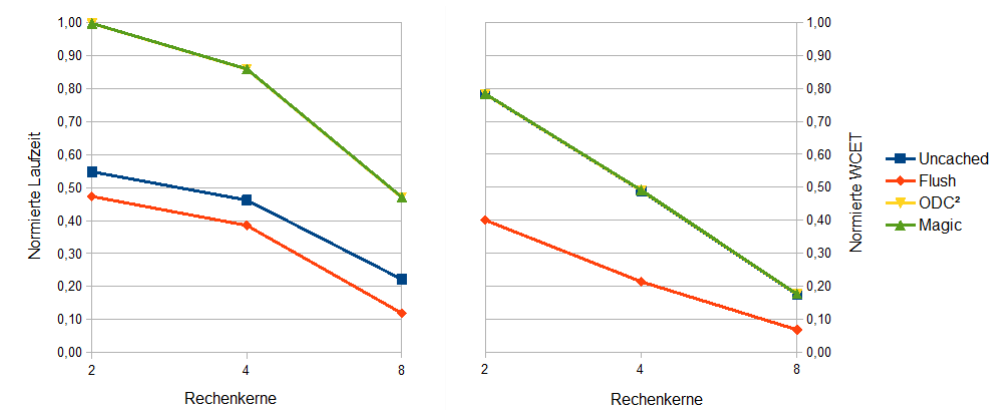


Abbildung 7.10: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von Smoothing.

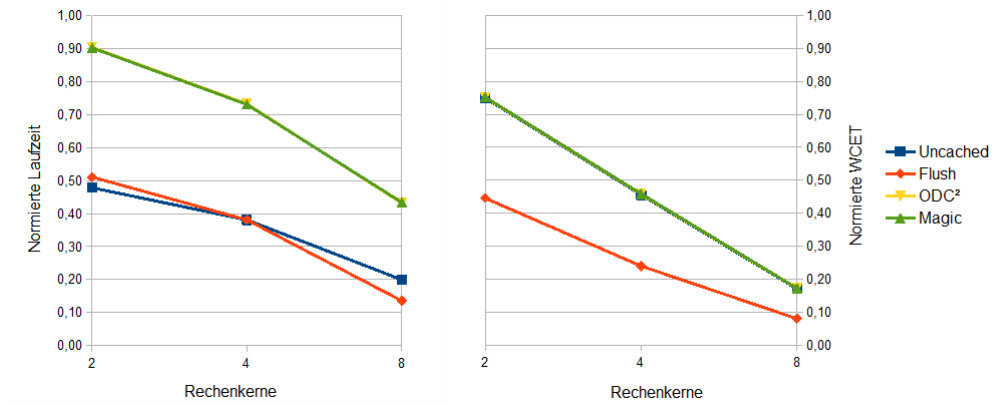


Abbildung 7.11: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von Edges.

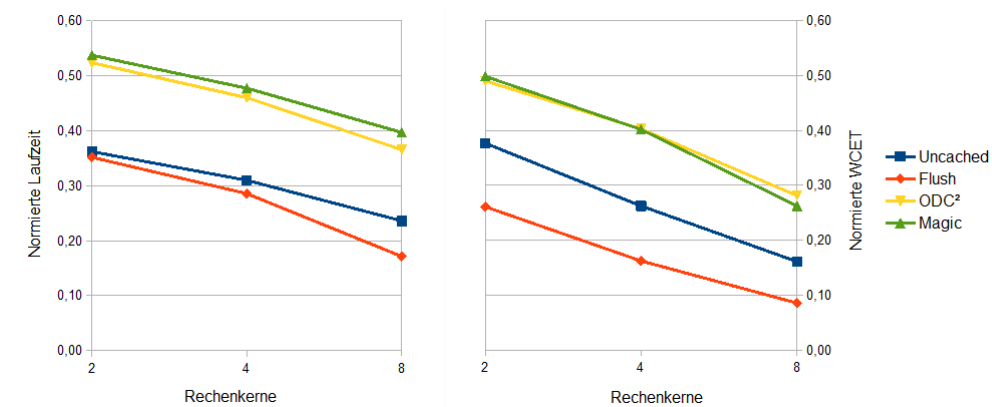


Abbildung 7.12: Auf bestmögliche Skalierung normierte Laufzeit und $WCET_{est}$ bei Ausführung von 3DPP.

Anhang 4: Histogramm der Invalidierung

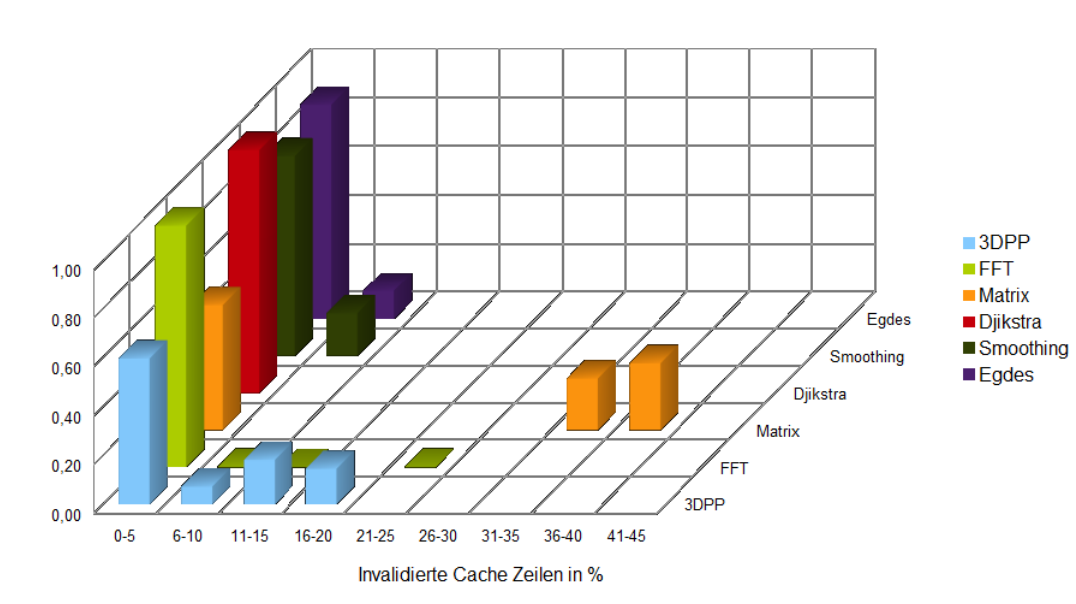


Abbildung 7.13: Histogramm des Ausmaßes der Invalidierung während der Restore-Procedure im ODC² bei Ausführung mit 2 Rechenkernen.

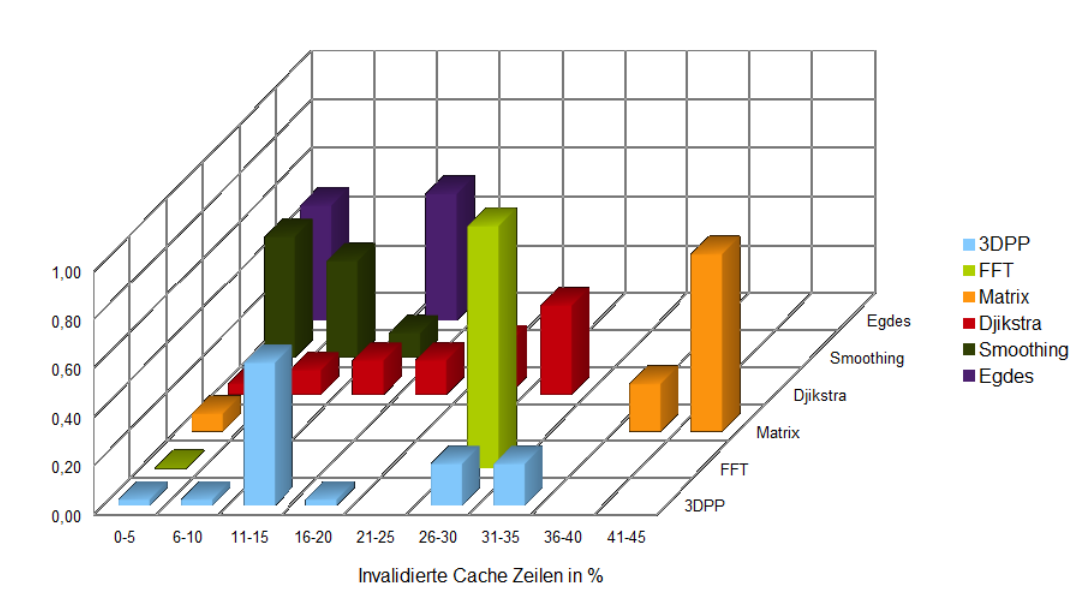


Abbildung 7.14: Histogramm des Ausmaßes der Invalidierung an Synchronisationspunkten mit *Cache Flush* bei Ausführung mit 2 Rechenkernen.

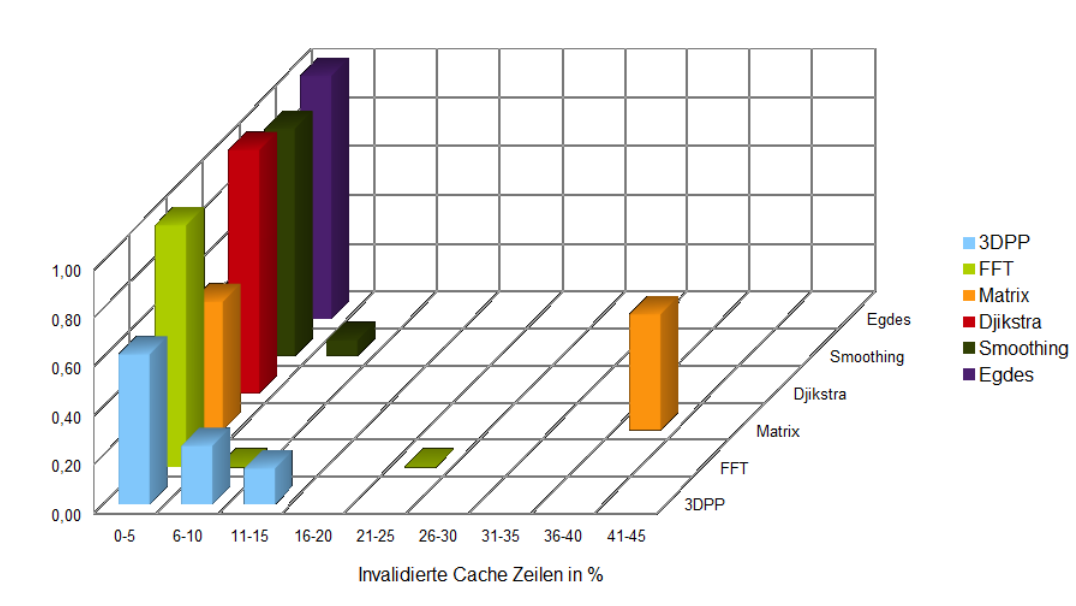


Abbildung 7.15: Histogramm des Ausmaßes der Invalidation während der Restore-Procedure im ODC² bei Ausführung mit 4 Rechenkernen.

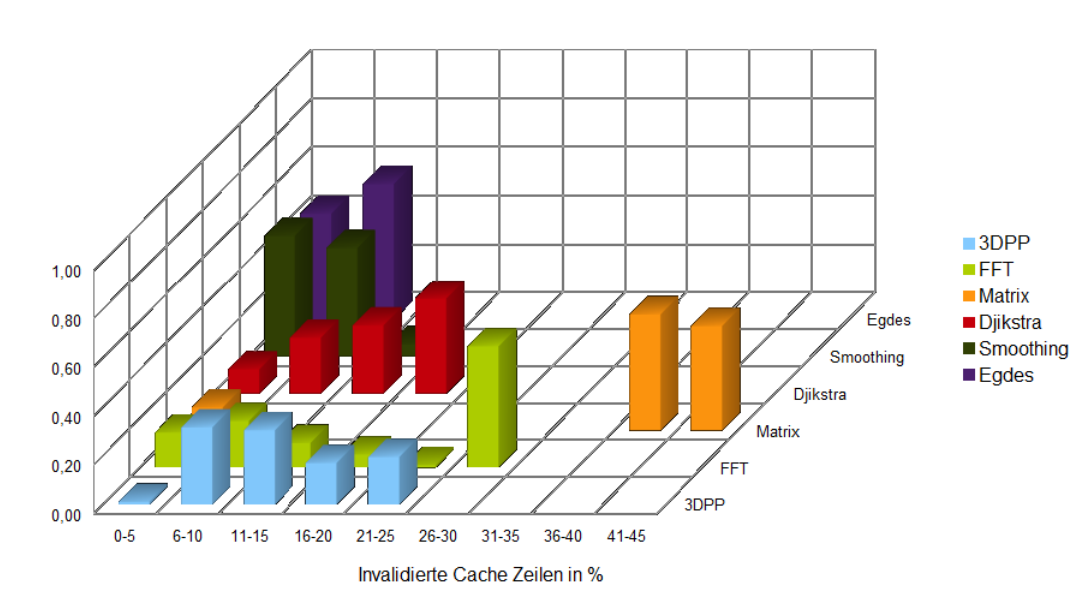


Abbildung 7.16: Histogramm des Ausmaßes der Invalidation an Synchronisationspunkten mit *Cache Flush* bei Ausführung mit 4 Rechenkernen.

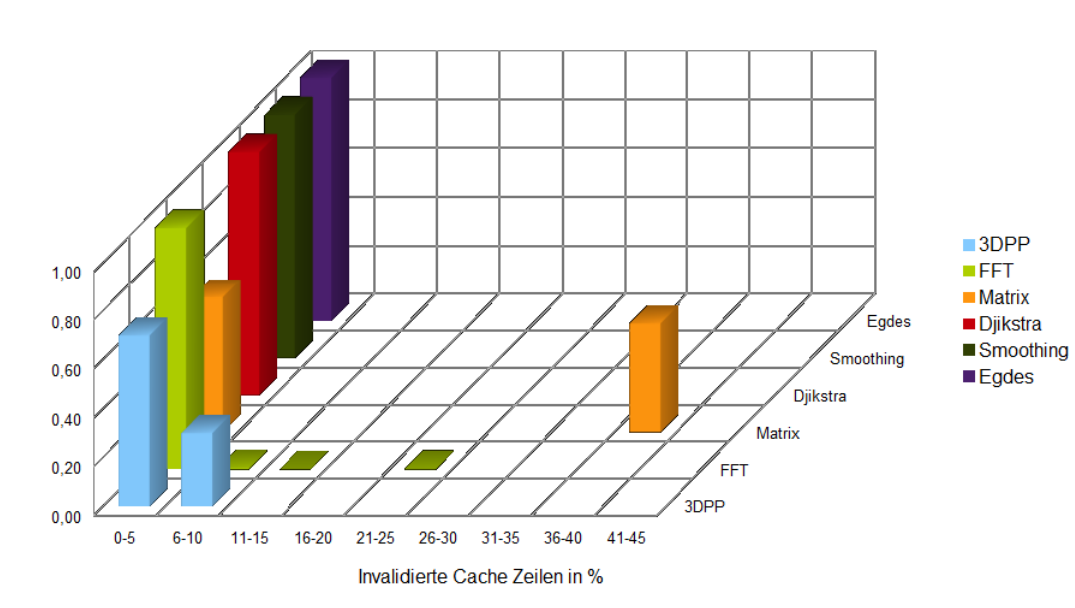


Abbildung 7.17: Histogramm des Ausmaßes der Invalidation während der Restore-Procedure im ODC² bei Ausführung mit 8 Rechenkernen.

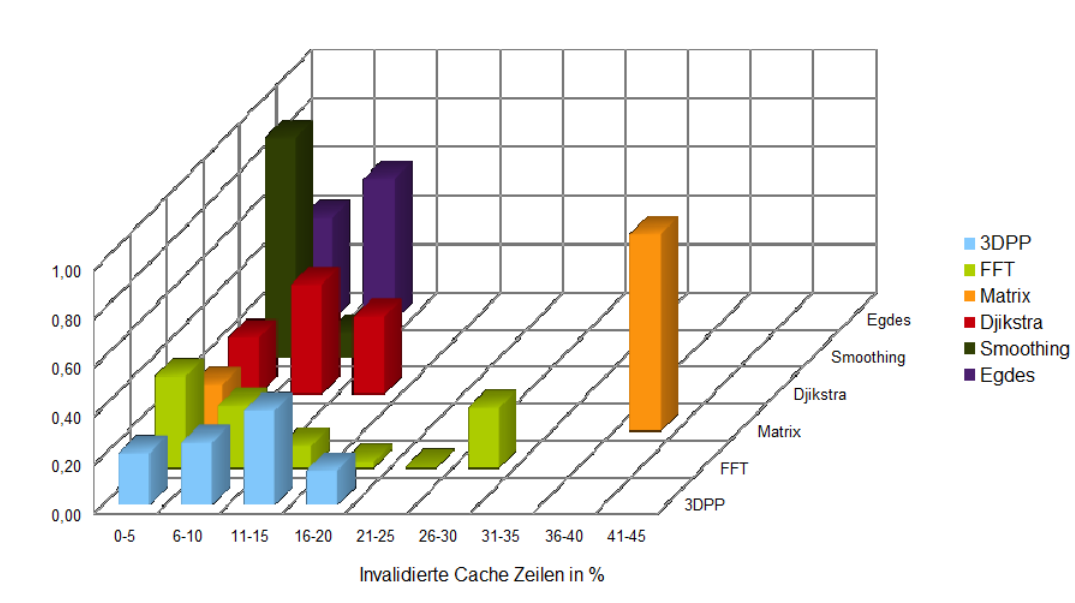


Abbildung 7.18: Histogramm des Ausmaßes der Invalidation an Synchronisationspunkten mit *Cache Flush* bei Ausführung mit 8 Rechenkernen.

Anhang 5: Zugriffstypen

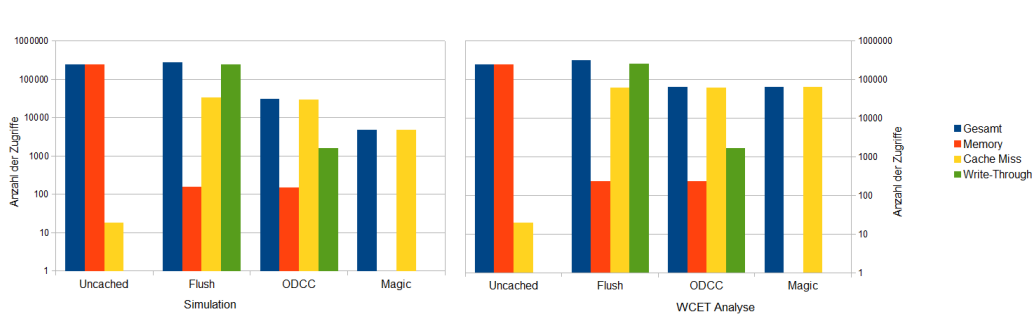


Abbildung 7.19: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Matrix mit 2 Rechenkernen.

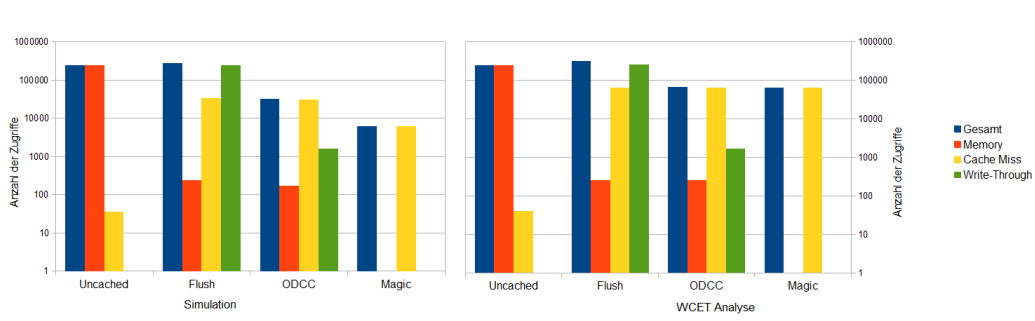


Abbildung 7.20: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Matrix mit 4 Rechenkernen.

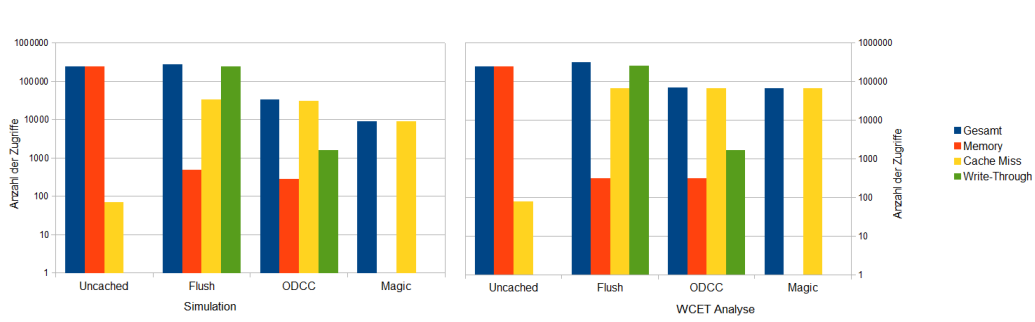


Abbildung 7.21: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Matrix mit 8 Rechenkernen.

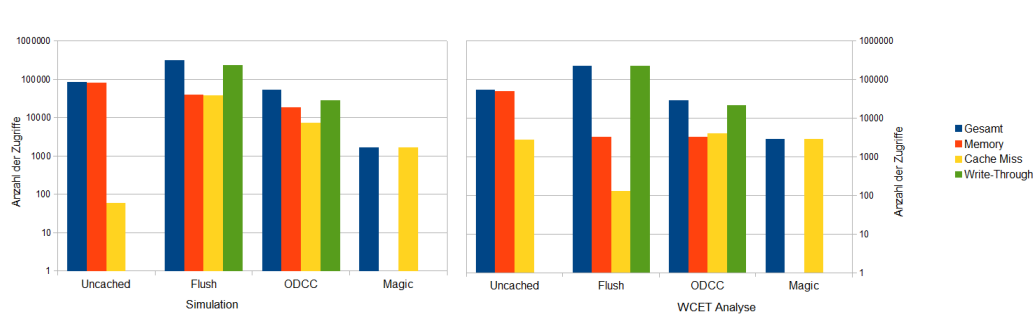


Abbildung 7.22: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von FFT mit 2 Rechenkernen.

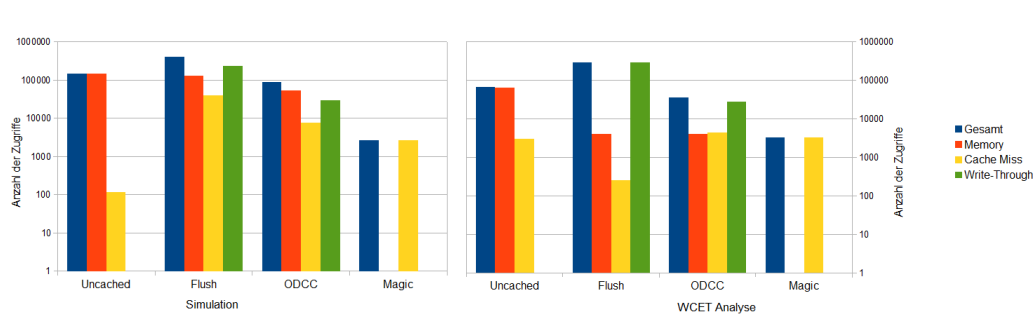


Abbildung 7.23: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von FFT mit 4 Rechenkernen.

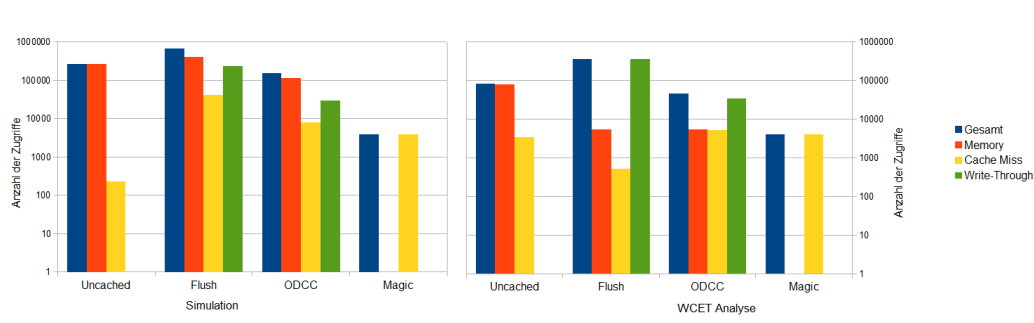


Abbildung 7.24: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von FFT mit 8 Rechenkernen.

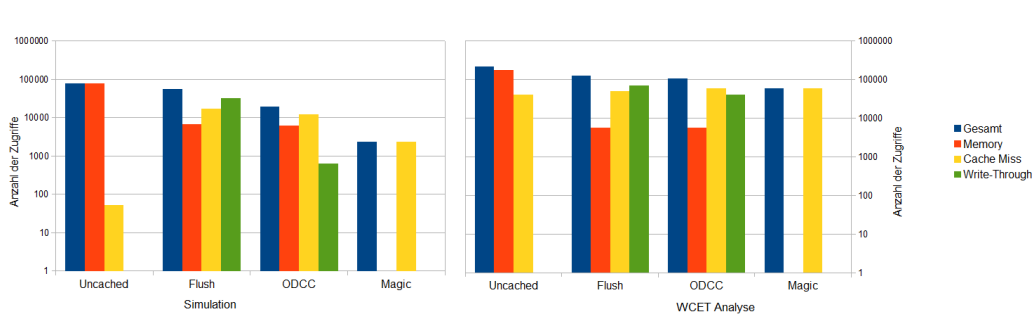


Abbildung 7.25: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Dijkstra mit 2 Rechenkernen.

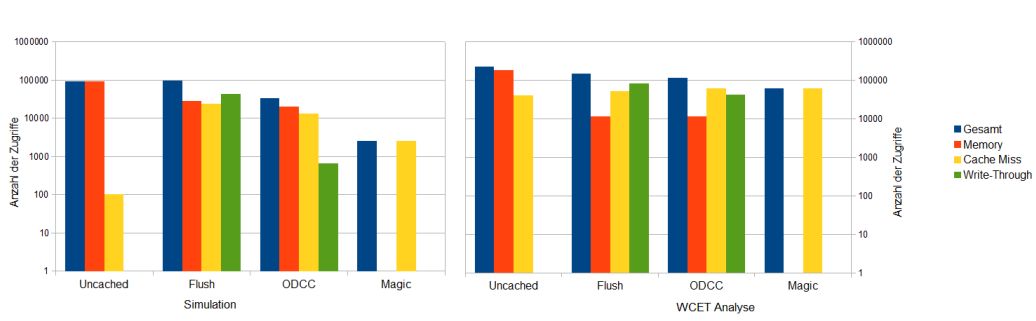


Abbildung 7.26: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Dijkstra mit 4 Rechenkernen.

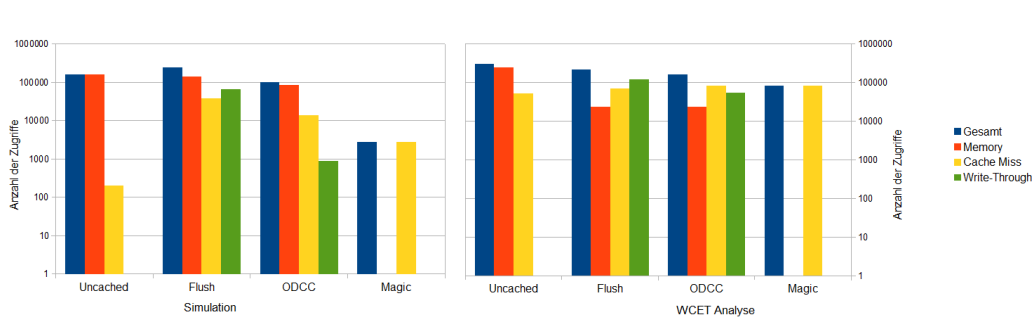


Abbildung 7.27: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Dijkstra mit 8 Rechenkernen.

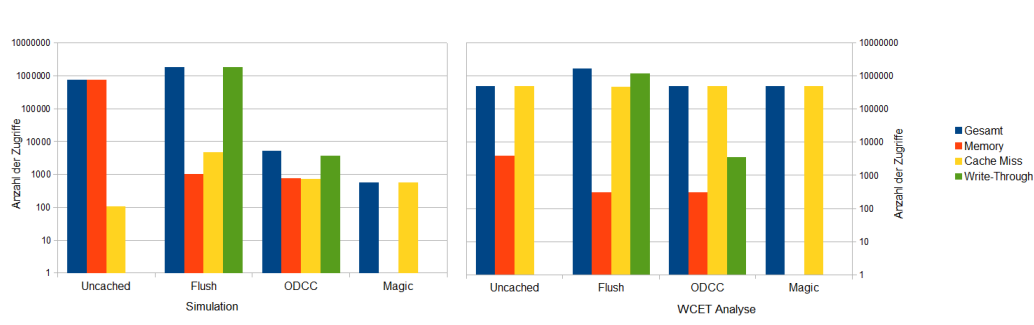


Abbildung 7.28: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Smoothing mit 2 Rechenkernen.

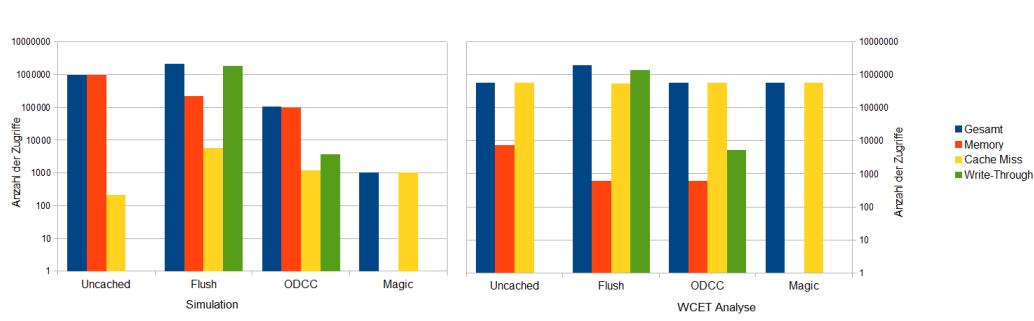


Abbildung 7.29: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Smoothing mit 4 Rechenkernen.

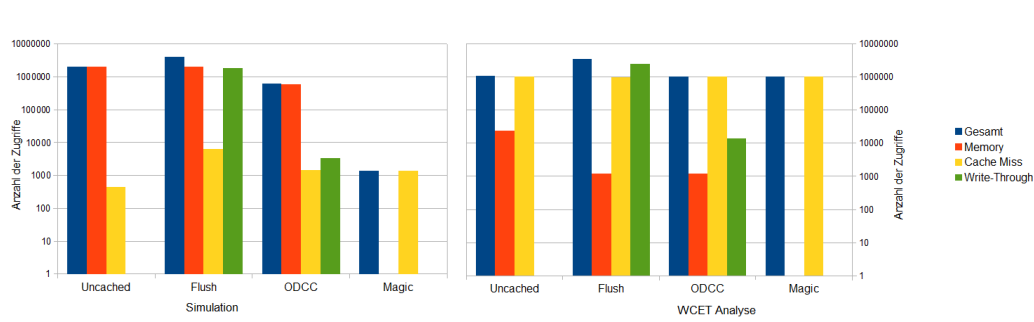


Abbildung 7.30: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Smoothing mit 8 Rechenkernen.

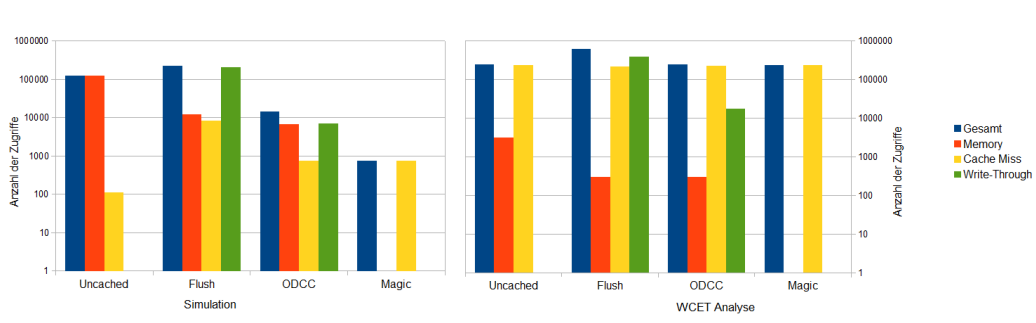


Abbildung 7.31: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Edges mit 2 Rechenkernen.

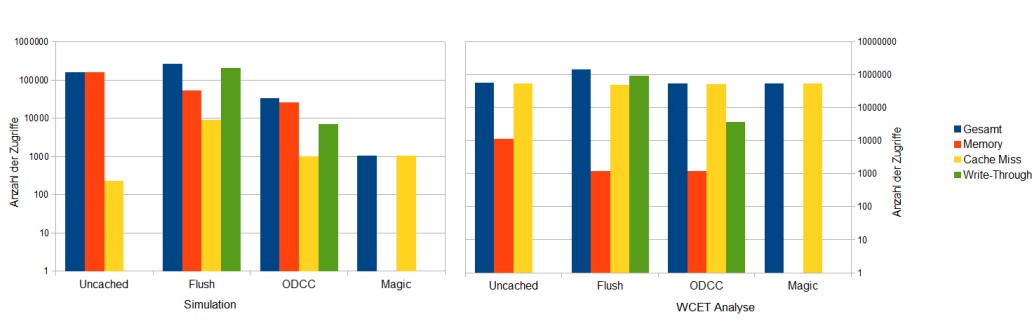


Abbildung 7.32: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Edges mit 4 Rechenkernen.

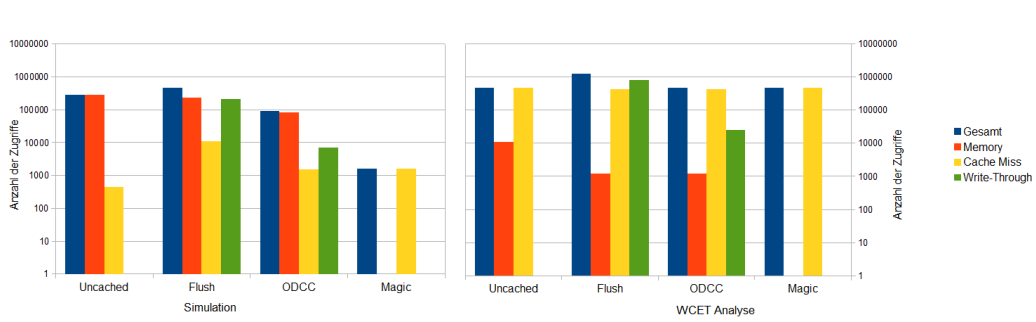


Abbildung 7.33: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von Edges mit 8 Rechenkernen.

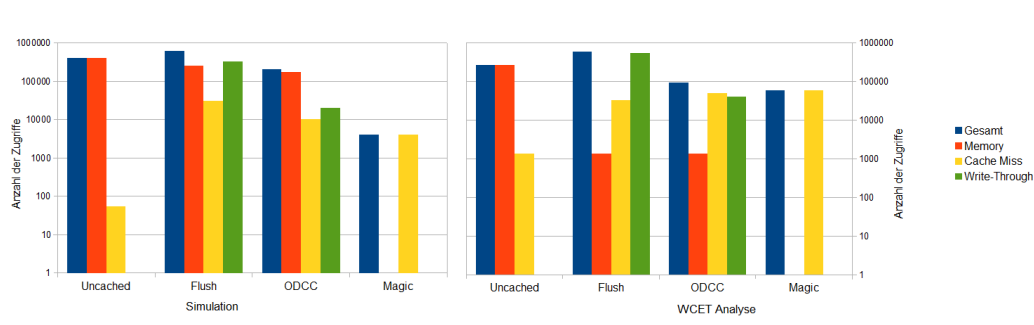


Abbildung 7.34: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von 3DPP mit 2 Rechenkernen.

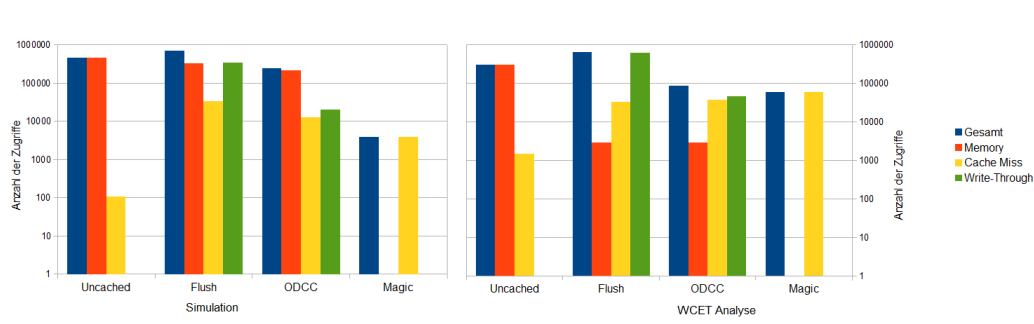


Abbildung 7.35: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von 3DPP mit 4 Rechenkernen.

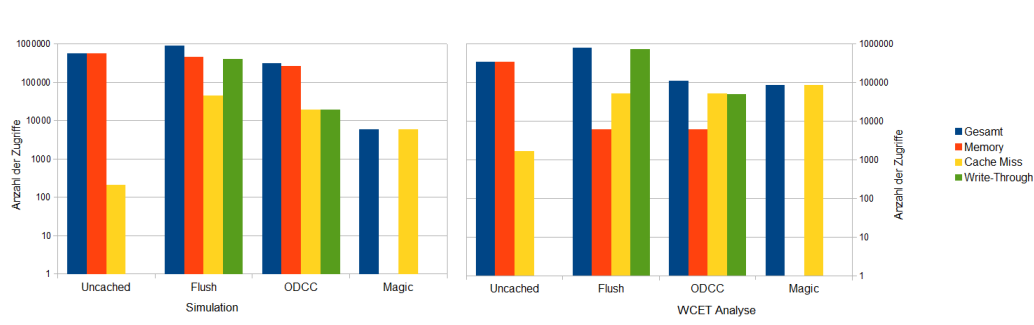


Abbildung 7.36: Differenzierung der Speicherzugriffe nach Zugriffstypen bei Ausführung von 3DPP mit 8 Rechenkernen.