Zusammenfassung

In dieser Arbeit wird ein modularer, trace-gesteuerter Cache-Simulator in Java^{TM} implementiert. Die entwickelten Komponenten entsprechen dabei der JavaBeans^{TM} Spezifikation, um einen einfachen Aufbau möglichst vieler Cache-Architekturen mit Hilfe von visuellen Entwicklungswerkzeugen zu ermöglichen. Aus Performance Gründen werden alle Komponenten sowohl visuell für den didaktischen Einsatz bei kleinen, als auch nichtvisuell zur „schnellen" Simulation größerer Caches entwickelt.

Als Entwicklungsumgebung und Testplattform kommt das Java^{TM} 2 SDK und das BDK 1.1 der Firma Sun Microsystems, Inc. zum Einsatz.

Eberhard-Karls-Universität Tübingen
Wilhelm-Schickhard-Institut für Informatik
Arbeitsbereich Technische Informatik

Studienarbeit

Ein konfigurierbarer, visueller Cache-Simulator unter spezieller Berücksichtigung komponentenbasierter Modellierung mit Java Beans

Holger Morgenstern
Breslauer Str. 24
D-72501 Gammertingen
Holger@Morgenstern.net

06. März 2001

Inhaltsverzeichnis

Abbildungsverzeichnis 4

1 Einleitung 6

2 Cache-Speicher 8

2.1 Grundidee . . . 8
2.2 Lokalitätseigenschaft von Referenzen . . . 9
2.3 Cache-Struktur . . . 9
2.4 Cache Designparameter . . . 10
2.4.1 Cache-Größe . . . 11
2.4.2 Mapping-Funktion . . . 11
2.4.3 Ersetzungs-Algorithmen . . . 13
2.4.4 Schreibstrategie - Cache-Konsistenz - Allocate-On-Write . . . 14
2.4.5 Zeilengröße . . . 16
2.4.6 Cache-Anzahl . . . 17
2.5 Trace-gesteuerte Cache-Simulation . . . 18

3 Simulator Beans 20

3.1 JavaBeans™ . . . 21
3.2 Abstrakter Aufbau eines Cache-Simulators . . . 22
3.3 Eventmodell . . . 23
3.4 Simulator Beans im Detail . . . 25
3.4.1 ReadTrace . . . 25
3.4.2 VisualReadTrace . . . 26
3.4.3 Cache . . . 27
3.4.4 VisualCache . . . 28
3.4.5 Memory . . . 30
3.4.6 VisualMemory . . . 31

4 Anwendung und Ergebnisse **32**
4.1 Anwendung der Klassenbibliothek 32
4.1.1 Ein nichtvisueller Splitcache 32
4.1.2 Ein visueller Splitcache 34
4.2 Anwendung mit grafischen Entwicklungsumgebungen 36
4.3 Überprüfung der Ergebnisse 40

5 Zusammenfassung **44**

Literaturverzeichnis **45**

Abbildungsverzeichnis

2.1 Cache und Hauptspeicher 8
2.2 Cache-Hauptspeicher-Struktur 10
2.3 Direct Mapping: Bitselektion der Hauptspeicheradresse 12
2.4 Associative Mapping: Bitselektion der Hauptspeicheradresse 12
2.5 Set Associative Mapping: Bitselektion, Cachezugriff 14
2.6 write-through mit No-Allocate-On-Write Strategie 16
2.7 write-back mit Allocate-On-Write Strategie 16
2.8 Two Level Cache 17
2.9 Split-Cache: Instruction-Cache und Data-Cache 18

3.1 Tracegesteuerter Cache-Simulator: Abstrakter Aufbau 22
3.2 Beispiel Bean-Kommunikation mit Events 24
3.3 Klassendiagramm: ReadTrace 25
3.4 Klassendiagramm: VisualReadTrace 26
3.5 Klassendiagramm: Cache 27
3.6 Klassendiagramm: VisualCache 29
3.7 Klassendiagramm: Memory 30
3.8 Klassendiagramm: VisualMemory 31

4.1 Beispiel 1: Ein nichtvisueller Splitcache, Sourcecode 33
4.2 Beispiel 1: Simulationsergebnis 34
4.3 Beispiel 2: Ein visueller Splitcache, Sourcecode 35
4.4 Beispiel 2: Ein visueller Splitcache in Aktion 36
4.5 BeanBox Toolbox mit Simulator Beans 37
4.6 BeanBox Property Editoren für VisualCache Bean 38
4.7 BeanBox Eventschnittstellen 39
4.8 BeanBox Eventlistener Auswahl 39
4.9 Simulation: Cachegröße, direct mapped 40
4.10 Simulation: Cachegröße, 2-way-set-associative 41

4.11 Simulation: Cachegröße, 4-way-set-associative 41
4.12 Simulation von Cache Größe und Assoziativität (compress, inst. miss rate) 42
4.13 Simulation von Cache Größe und Assoziativität (compress, data miss rate) 42
4.14 Simulation: Anteil Daten/Instruktionen 43
4.15 Simulation: Anteil Schreib-/Lesezugriffe 43

1 Einleitung

Caches[1] sind kleine, dafür aber sehr schnelle Speicher, die in nahezu allen modernen Computersystemen eingesetzt werden, um Zugriffe auf nachfolgende Speichereinheiten (Hauptspeicher, Festplatten, ...) zu beschleunigen. Eine integrierte, von den Cache-Designparametern bestimmte Logik regelt dabei, welche Speicherabschnitte der nachfolgenden Einheit aktuell in den Cache geladen bzw. dort gehalten werden, um eine möglichst niedrige Gesamtzugriffszeit zu erreichen.

Cache-Simulatoren werden einerseits dazu benutzt, um die prinzipielle Performance unterschiedlicher Cache-Konfigurationen und Designparameter zu testen. Mit Simulatoren, die die internen Vorgänge eines Caches visualisieren, kann andererseits das Verständnis für die Bedeutung und Funktion der einzelnen Designparameter vertieft werden. Cache-Simulatoren sollten dabei flexibel in ihrer Konfiguration sein, um ein möglichst breites Spektrum der unterschiedlichen Designs abzudecken.

Ziel dieser Arbeit ist daher die Entwicklung von Softwarekomponenten aus denen trace-gesteuerte, visuelle Simulatoren für eine breite Palette von Cache-Architekturen möglichst einfach und mit Hilfe visueller Entwicklungswerkzeuge aufgebaut werden können. Die einzelnen Komponenten sollen dabei wiederverwendbar und möglichst einfach zu konfigurieren sein. Um diese Ziele und gleichzeitig eine größtmögliche Plattformunabhängigkeit mit vertretbarem Aufwand zu erreichen, kommt für die Implementierung Java™ zum Einsatz.

Die Softwarekomponenten werden gemäß der JavaBeans™ Spezifikation entwickelt und können so mit allen visuellen Java™-Entwicklungsumgebungen, die JavaBeans™ unterstützen, einfach und persistent konfiguriert werden. Auch der Aufbau von komplexen Cache-Konfigurationen gestaltet sich mit Hilfe dieser Werkzeuge einfach und übersichtlich. Daneben ist es natürlich auch möglich, diese Komponenten (Beans) „klassisch“, also als Klassenbibliothek zu verwenden.

Alle Komponenten werden jeweils in einer visuellen und in einer nichtvisuellen Form getrennt entwickelt und bereitgestellt. Die visuellen Komponenten eignen sich dabei vor allem für den Aufbau von Simulatoren relativ kleiner Cache-Konfigurationen. Sie gestatten eine einzelschrittweise Simulation und stellen dabei den jeweils aktuellen Cache-Zustand detailliert grafisch dar, was vor allem für didaktische Zwecke nützlich ist. Interessiert bei der Simulation weniger der interne Ablauf und mehr das statistische Ergebnis der gewählten Konfiguration, können die Simulatoren mit den nichtvisuellen

[1] Das Wort Cache stammt vom Französischen *cacher* (verstecken), da sich Caches normalerweise vollkommen transparent verhalten.

Komponenten implementiert werden, die am Ende des Simulationslaufs eine entsprechende Statistik bereitstellen. Ein Einsatz der Komponenten in größeren Projekten, bei denen nicht der Cache an sich Untersuchungsgegenstand ist, wird durch eine komplette Modellierung des Datenpfads ebenfalls problemlos ermöglicht.

Als Entwicklungsumgebung und Testplattform kommt das Java™ 2 SDK sowie die BeanBox aus dem BDK 1.1 der Firma Sun Microsystems, Inc.[2] zum Einsatz, welche für eine Vielzahl von Hardwareplattformen kostenlos verfügbar sind[3].

Kapitel 2 *Cache-Speicher* dieser Ausarbeitung stellt die für die Simulatorentwicklung wichtigsten Grundlagen von Caches vor, gibt einen Überblick über Designparameter und begründet die Auswahl einer trace-gesteuerten Simulation. Die entwickelten Softwarekomponenten werden in Kapitel 3 *Simulator Beans* detailiert vorgestellt. Kapitel 4 *Anwendung und Ergebnisse* stellt die konkrete Anwendung der Simulatorkomponenten in grafischen Entwicklungsumgebungen und als Klassenbibliothek vor. Eine Zusammenfassung der Ergebnisse dieser Arbeit und ein Ausblick schließen sich in Kapitel 5 *Zusammenfassung* an.

[2]http://www.sun.com/
[3]http://java.sun.com/products/jdk/1.2/index.html

2 Cache-Speicher

2.1 Grundidee

In modernen Computersystemen möchte man einen möglichst großen, kostengünstigen Speicher zur Verfügung stellen, dessen Zugriffszeit dabei aber möglichst gering sein sollte. Da die schnellsten verfügbaren Speicherbausteine meist aber gleichzeitig die teuersten sind, kann man dieses Ziel nicht direkt erreichen. Die Idee ist nun, den großen Speicher mit preislich günstigen Bausteinen zu realisieren und einen relativ kleinen Puffer-Speicher, den *Cache*, aus schnellen IC's transparent davorzuschalten.

Alle Speicherzugriffe laufen dabei über den Cache, der eine Kopie von bestimmten Speicherbereichen enthält. Versucht der Prozessor jetzt, ein bestimmtes Wort im Speicher zu lesen, wird zunächst geprüft, ob es sich gerade im Cache befindet. Falls ja, wird es direkt aus dem Cache geliefert. Falls nein, wird zunächst ein ganzer Speicherblock, der das gewünschte Wort enthält, in den Cache geladen[1]. Dieses Schema zeigt Abbildung 2.1.

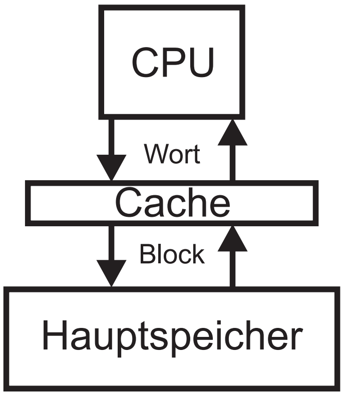


Abbildung 2.1: Cache und Hauptspeicher

[1]Die Cache-Speicher-Verbindung kann so entworfen werden, dass der Speicher durch das zusätzliche Lesen eines ganzen Blocks nicht langsamer wird (ein paar Schlagworte dazu: nibble mode, block mode der Speicher, Interleaving, Pipelining, breite Speicherbusse zum parallelen übertragen mehrerer Worte. . .). Oft wird auch das benötigte Wort zuerst geladen und zum Prozessor weitergeleitet und erst danach der restliche Block.

Wenn es gelingt, den weitaus größten Teil der Speicherzugriffe direkt aus dem Cache zu bedienen, erreicht man so für das Speichersystem eine Zugriffszeit, die nahe an der der schnellsten Bausteine liegt, gleichzeitig aber auch Pro-Bit-Kosten, die nicht wesentlich über denen von günstigen Speicher-IC's liegen. Die wesentliche Vorraussetzung dafür, das dies tatsächlich möglich ist, wird im nächsten Abschnitt beschrieben.

2.2 Lokalitätseigenschaft von Referenzen

Untersuchungen des Speicherzugriffsverhaltens von Programmen ergaben eine interessante Eigenschaft: Die Zugriffe eines Programmes auf den Speicher besitzen eine große *Lokalität* [Denn72, Smith82]. Diese hat zwei Aspekte:

Temporale Lokalität: Ein Programm verteilt seine Speicherreferenzen innerhalb von kurzen Zeitspannen uneinheitlich über seinen Adressraum. Die Bereiche, die dabei bevorzugt werden, bleiben weitgehend über längere Zeitspannen dieselben. Daraus folgt, dass ein Programm zu jedem Zeitpunkt in naher Zukunft mit großer Wahrscheinlichkeit nur solche Adressen referenzieren wird, die es in der unmittelbaren Vergangenheit referenziert hat. (z.B. Programmschleifen, Stacks)

Räumliche Lokalität: Die von einem Programm benutzten Teile des Adressraums bestehen generell aus einer Anzahl von individuellen, jeweils zusammenhängenden Segmenten. Das bedeutet, zu jedem Zeitpunkt werden in naher Zukunft mit großer Wahrscheinlichkeit nur solche Adressen referenziert, die nahe bei denen liegen, die in der unmittelbaren Vergangenheit referenziert wurden. (z.B. sequentieller Code, Arrays)

Da der Cache gerade Kopien solcher Speicherbereiche enthält, die in der „näheren" Vergangenheit referenziert wurden, ergibt sich eine große Wahrscheinlichkeit, dass sich zukünftig benötigte Daten bereits im Cache befinden.

2.3 Cache-Struktur

Abbildung 2.2 auf der nächsten Seite nach Stallings [Stall96] zeigt eine typische Struktur eines Cache-Hauptspeicher-Systems. Der Hauptspeicher besteht aus maximal 2^n adressierbaren Worten. Jedes Wort hat dabei eine eindeutige n-bit Adresse. Um das Mapping zwischen Cache und Hauptspeicher zu vereinfachen, unterteilt man den Hauptspeicher in $M = 2^n/K$ gleich große Blöcke zu je K Worten. Der Cache besteht seinerseits aus C Zeilen zu ebenfalls je K Worten. Die Anzahl der Cache-Zeilen ist dabei sehr viel kleiner als die der Hauptspeicherblöcke ($C \ll M$).

Da zu jedem Zeitpunkt immer nur ein Teil der Hauptspeicherblöcke in Cache-Zeilen gespeichert sein kann, und daher eine Cache-Zeile nicht eindeutig und permanent ei-

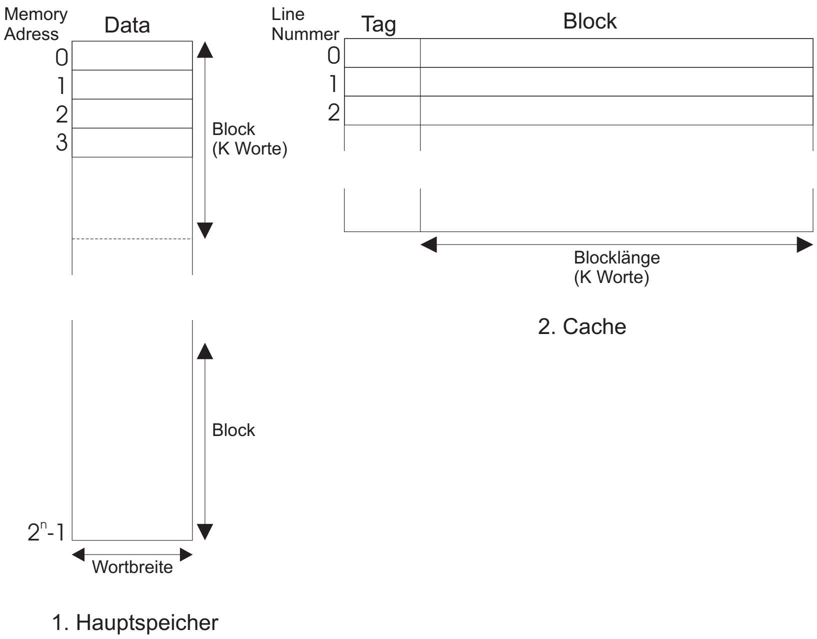


Abbildung 2.2: Cache-Hauptspeicher-Struktur

nem speziellen Block zugeordnet werden kann, enthält sie noch einen sogenannten *Tag*, der den jeweils gespeicherten Block eindeutig identifiziert[2].

2.4 Cache Designparameter

Das Ziel eines guten Cache-Designs ist es natürlich, eine möglichst optimale Performance des Speichersystems unter den gegebenen Kosten- und/oder Chipflächenbeschränkungen zu erreichen. Mit Hilfe von Cache-Simulationen versucht man, die Performance verschiedener Designalternativen abzuschätzen. Dabei spielen zwei Kriterien eine wichtige Rolle:

Zugriffszeit: Welche Zeit wird benötigt, um Daten aus dem Cache zu erhalten oder in den Cache zu schreiben? Da sich diese Frage aber schlecht allgemein, ohne Angabe einer konkreten Technologie und Implementierung beantworten läßt, konzentrieren sich Cache-Simulationen vor allem auf den nächsten Punkt.

miss-ratio: Welcher Prozentsatz aller Speicherreferenzen führt zu einem *miss* und kann nicht direkt aus dem Cache bedient werden?

Da ein Miss ja bedeutet, dass der Prozessor solange warten muß, bis das gewünschte Datum im langsamen Hauptspeicher erreicht werden kann, ist die miss-ratio für die

[2]Üblicherweise ist der Tag ein Teil der Hauptspeicheradresse des entsprechenden Blocks.

Performance des gesamten Speichersystems ein entscheidendes Merkmal. Durch Simulationen mit unterschiedlichen Design-Parametern versucht man, ein Cache-Design zu finden, das eine möglichst geringe miss-ratio aufweist. Im Folgenden werden einige grundlegende dieser Design-Parameter vorgestellt.

2.4.1 Cache-Größe

Die gewählte Größe des Caches bestimmt die miss-ratio natürlich wesentlich. Ein größerer Cache hat sicherlich eine geringere miss-ratio als ein kleinerer. Sie wird im Wesentlichen durch die Vorgabe bestimmt, dass sie einerseits groß genug sein muß, um eine akzeptable miss-ration zu erreichen und andererseits klein genug, um die Pro-Bit-Kosten nicht zu stark zu erhöhen[3].

Da bei größeren Caches auch die Zahl der zur Adressierung benötigten gates zunimmt, neigen diese dazu, mit wachsender Größe geringfügig langsamer zu werden. Ein weiterer Grund, die Cache-Größe zu minimieren[4].

2.4.2 Mapping-Funktion

Wie bereits im Abschnitt 2.3 *Cache-Struktur* erwähnt, ist die Anzahl der Zeilen im Cache sehr viel kleiner, als die der Blöcke im Hauptspeicher. Daher wird eine Mapping-Funktion, die einem Hauptspeicherblock einen Platz im Cache, also eine konkrete Zeile zuordnet, benötigt. Die eindeutige Identifikation des in einer Zeile gespeicherten Blocks erfolgt über den *Tag*. Die Art des verwendeten Mappings bestimmt weitgehend, wie der Cache weiter organisiert wird. Im Folgenden werden die drei möglichen Mapping-Techniken, *Direct*, *Associative* und *Set Associative*, jeweils näher beschrieben.

Direct Mapping

Die einfachste Technik, Hauptspeicherblöcke einer bestimmten Cache-Zeile zuzuordnen, ist das Direct Mapping. Dabei wird jeder Block auf nur eine mögliche Zeile abgebildet. Diese Funktion kann mit einer einfachen modulo-Operation beschrieben werden:

$$i = j \bmod m$$

mit i = Zeilennummer, j = Blocknummer, m = Zeilenanzahl

Direct Mapping kann mit Hilfe einer Bit-Selektion sehr einfach implementiert werden. Die Hauptspeicheradresse wird dazu in drei Felder aufgeteilt. Die niederwertigsten w Bits selektieren dabei ein Wort, die übrigen s Bits einen der 2^s Hauptspeicherblöcke. Im Cache werden die höherwertigen $s - r$ Bits als Tag verwendet und die mittleren r Bits selektieren eine der $m = 2^r$ Zeilen (siehe Abbildung 2.3 auf der nächsten Seite).

[3] Bei On-Chip-Caches ist die Größenbeschränkung durch die verfügbare Chipfläche vorgegeben.

[4] Stallings bemerkt in [Stall96], dass eine Reihe von diesbezüglichen Studien eine „optimale“ Cache-Größe zwischen 1K und 512K Worten ergeben haben, wobei es durch die starke Abhängigkeit der Performance von der Art des konkreten workloads nicht möglich ist, eine wirklich „optimale“ Größe zu erreichen.

Abbildung 2.3: Direct Mapping: Bitselektion der Hauptspeicheradresse

Dies ergibt eine eindeutige Zuordnung von Hauptspeicherblöcken zu Cache-Zeilen. Direct Mapping kann mit geringem Hardwareaufwand und einem Minimum an control overhead implementiert werden und bietet die schnellste Zugriffszeit für beliebige Cache-Größen. Der Nachteil dieser Methode ist, dass jeder Hauptspeicherblock einer *festen* Cache-Zeile zugeordnet wird. Referenziert ein Programm wiederholt Worte aus unterschiedlichen Blöcken, die auf die gleiche Zeile abgebildet werden, hat das ein ständiges Laden dieser Blöcke in den Cache mit anschließender Verdrängung (*discard*, *force out* derselben Daten bevor sie erneut benötigt werden zur Folge, was oft als „schlimmstes Scenario einer Cachearchitektur“, als *Ping Pong Effekt* oder *Cache Trashing* bezeichnet wird [Lin00] und eine sehr hohe miss-ratio zur Folge hat.

Associative Mapping

Beim Associative Mapping kann ein Hauptspeicherblock im Gegensatz zum Direct Mapping auf jede Cache-Zeile abgebildet werden. Die Hauptspeicheradresse wird dabei in nur zwei Felder unterteilt. Die niederwertigsten w Bits selektieren ein Wort und die restilchen s Bits werden als Tag benutzt (siehe Abbildung 2.4). Um herauszufinden, ob ein Block im Cache ist, müssen simultan die Tags aller Cache-Zeilen untersucht werden, was eine sehr komplexe Logik erfordert.

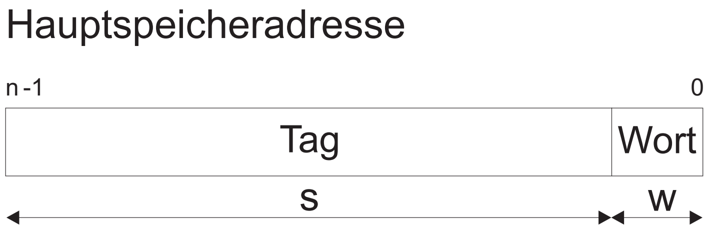


Abbildung 2.4: Associative Mapping: Bitselektion der Hauptspeicheradresse

Associative Mapping behebt den Hauptnachteil des Direct Mapping. Leider ist für die Implementierung dazu ein enormer Hardwareaufwand für das parallele Untersuchen aller Cache-Zeilen nötig, so dass dieses Schema (fast) nie verwendet wird.

Set Associative Mapping

Die Vorteile der beiden ersten Mapping-Funktionen ohne deren Nachteile, leistet das Set Associative Mapping. Es unterteilt den Cache in v gleichgroße Abschnitte (*Sets*), die jeweils k Zeilen umfassen. Ein Hauptspeicherblock wird dann auf ein festes Set abgebildet (Direct Mapping), kann darin aber eine beliebige Zeile belegen (Associative Mapping). Die Abbildung kann so formuliert werden:

$$m = v * k$$
$$i = j \bmod v$$

mit i = Zeilennummer, j = Blocknummer, m = Zeilenanzahl

Abbildung 2.5 auf der nächsten Seite zeigt die entsprechende Bitselektion der Hauptspeicheradresse. Die niederwertigsten w Bits selektieren wieder ein Wort, die mittleren d Bits eines der $v = 2^d$ Sets und die höherwertigen $s - d$ Bits werden als Tag verwendet.

Wählt man $v = m$ und $k = 1$, erhält man ein Direct, bei $v = 1$ und $k = m$ ein Associative Mapping. Die Fälle $v = m/2, k = 2$ und $v = m/4, k = 4$, auch *Two-way* bzw. *Four-way set associative* genannt, sind die am häufigsten verwendeten. Sie optimieren die miss-ratio im Vergleich zum Direct Mapping erheblich und verursachen dabei nur einen relativ geringen Hardwaremehraufwand.

Die Freiheit bei der Abbildung eines Hauptspeicherblocks auf eine Cache-Zeile erfordert beim Laden desselben eine Auswahl, welcher Block dafür aus dem Cache entfernt werden soll. Funktionen, die das leisten, beschreibt der nächste Abschnitt.

2.4.3 Ersetzungs-Algorithmen

Funktionen, die bestimmen, welcher alte Block beim Laden eines Neuen aus dem Cache entfernt werden soll, haben natürlich das Ziel, die miss-ratio zu optimieren. Gleichzeitig müssen sie aus aus Performancegründen in Hardware implementiert werden und sollten daher nicht zu aufwendig sein. Die dafür in Frage kommenden Algorithmen können grob in *nutzungsbasierte* und *nicht-nutzungsbasierte* unterteilt werden. Die vier am häufigsten verwendeten sind:

LRU (least-recently-used): Ersetzt wird der am längsten unreferenzierte Block. Da die zuletzt referenzierten Blöcke am wahrscheinlichsten in der Zukunft wieder referenziert werden, darf man vermuten, dass dieser Algorithmus die niedrigste miss-rate ergibt. Implementiert werden kann er mit Hilfe von zusätzlichen Statusbits pro Zeile[5].

FIFO (first-in-first-out): Ersetzt wird der Block, der am längsten im Cache war. Die Implementation kann einfach z.B. mit einem round-robin-schedule erfolgen

[5]Für eine Set-Größe von zwei z.B. mit einem hot/cold (toggle)Bitt. Allgemein beschreibt [Smith82] eine Implementierung mit einer *upper-left triangular matrix*, die bei einer Set-Größe von k mit $k(k-1)/2$ Statusbits auskommt ($\lceil \log 2k! \rceil$ ist das theoretische Minimum), einfach in Hardware zu implementieren und besonders schnell ausführbar ist. Die Menge der zusätzlichen Statusbits ist bis zu einer Set-Größe von 8 akzeptabel. Für größere Sets werden Approximationen an LRU verwendet.

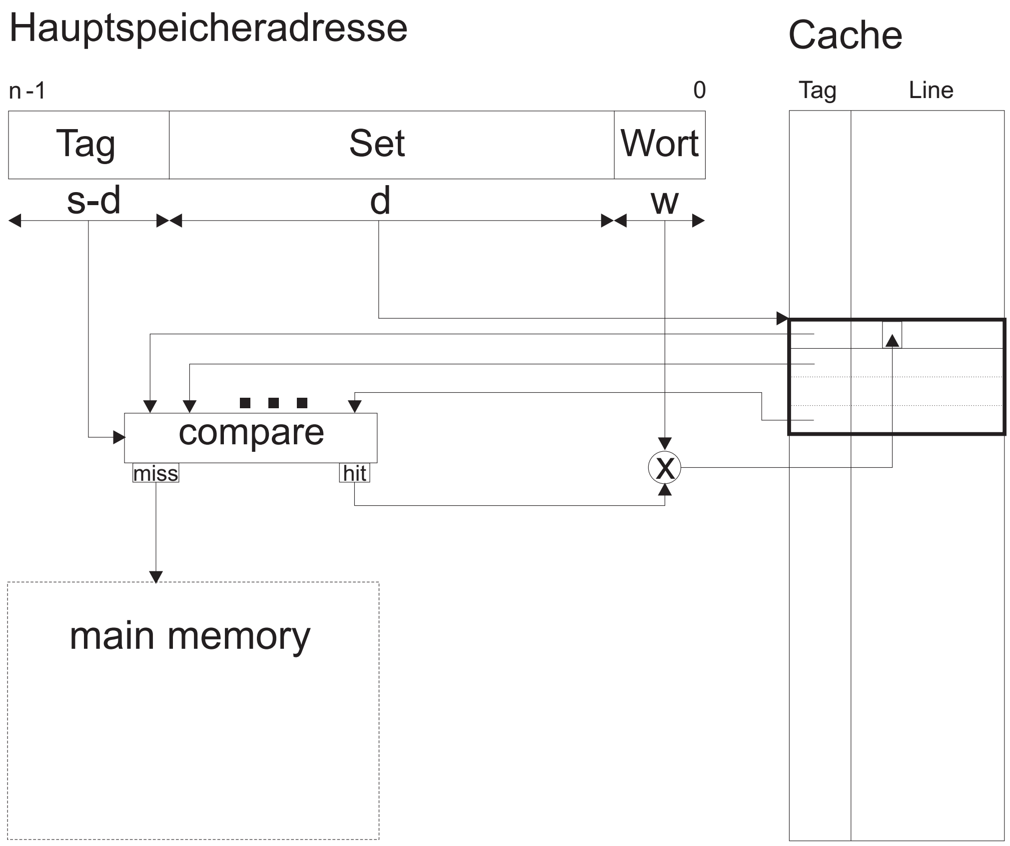


Abbildung 2.5: Set Associative Mapping: Bitselektion, Cachezugriff

LFU (least-frequently-used): Ersetzt wird der Block, der am wenigsten benutzt wurde. Bei der Implementierung muß ein Zähler pro Zeile hinzugefügt werden.

Random: Ersetzt wird ein (pseudo-) zufällig ausgewählter Block. Dieser Nicht-Nutzungsbasierte Algorithmus kann bei einer Set-Größe von k z.B. mit einem einzigen $\mathrm{mod}k$-Zähler, der durch beliebige Systemereignisse incrementiert wird und den zu ersetzenden Block auswählt, einfach und effizient implementiert werden.

Simulationsstudien zeigen, dass ein Random-Ersetzungs-Algorithmus nur eine geringfügig schlechtere Performance ergibt, als nutzungsbasierte Algorithmen [Smith82].

2.4.4 Schreibstrategie - Cache-Konsistenz - Allocate-On-Write

Alle Schreibzugriffe des Prozessors müssen irgendwann, spätestens aber dann, wenn eine Cachezeile überschrieben werden soll, vom Cache an den Hauptspeicher weitergegeben werden. Der Cache kann dabei unterschiedliche Strategien verfolgen, die einer der folgenden beiden Gruppen angehören:

write-through, store-through: Hier werden alle Schreiboperationen vom Cache direkt und unmittelbar auf den nachfolgenden Speicher durchgeschrieben. Diese Technik kann relativ einfach und ohne größere Konsistenzprobleme implementiert werden. Der Nachteil dieser Strategie ist ein hoher Cache-Hauptspeicher-Verkehr, der die Gesamtperformance verschlechtern kann, wenn es viele Schreibzugriffe gibt. Simulationsstudien zeigen, dass durchschnittlich 15% aller Speicherreferenzen Schreiboperationen sind, wobei das Spektrum von 5% bis 35% reicht, also recht breit gestreut ist, wie Smith in [Smith82] zeigt.

write-back, copy-back: Diese Strategie minimiert den Cache-Hauptspeicher-Verkehr, indem sie alle Schreiboperationen zuerst nur im Cache durchführt und erst dann, wenn eine Zeile aus dem Cache entfernt werden soll, diese komplett zurückkopiert. Für die Implementierung wird dazu pro Cache-Zeile ein weiteres Statusbit (*update-bit* oder auch *dirty-bit* genannt) benötigt, das gesetzt wird, sobald eine Schreiboperation auf eines der Worte dieser Zeile erfolgt.

Diese Strategien vermeiden zwar den Nachteil von write-through, dafür ergeben sich aber nichttriviale Konsistenzprobleme, da sich Teile des nachfolgenden Speichers bis zum endgültigen Zurückschreiben in einem ungültigen Zustand befinden.

Cache-Kohärenz

Sobald mehr als eine Einheit Zugriff auf den Systemspeicher hat, können sich Cache-Konsistenzprobleme ergeben. Es könnte z.B. ein Ein-/Ausgabegerät in der Lage sein, direkt, also am CPU-Cache-Speicher-System vorbei, Schreib-/Leseoperationen auf dem Systemspeicher durchzuführen. Liest das Gerät Daten aus dem Systemspeicher, könnten diese, sofern sie nur im Cache upgedatet wurden, ungültig sein. Umgekehrt können Schreiboperationen des Geräts zu Inkonsistenzen im Cache führen. Noch komplexer werden die Probleme, wenn mehr als eine CPU, jeweils mit ihrem eigenen Cache, einen gemeinsamen Speicher benutzen.

Es wird auf verschiedene Weise versucht, mit diesen Problemen umzugehen[6]. Die entsprechenden Lösungen sind aber meist noch sehr aufwendig. Ein System, das Cache-Inkonsistenzen verhindert, wird auch *cache-koherent* genannt. Das Problem der Cache-Kohärenz ist ein aktives Forschungsgebiet und es gibt vielleicht zukünftig effektivere Wege, es zu lösen (z.B. in [Muk98] oder [Con00]).

Allocate-On-Write

Eine weitere Designentscheidung bei Schreibvorgängen betrifft das Ladeverhalten. Soll der Hauptspeicherblock, auf den sich der Schreibvorgang bezieht, in den Cache geladen werden, oder nicht? Die erste Variante nennt man *Allocate On Write*, die zweite entsprechend *No Allocate On Write*. Bezüglich der miss-ratio hat im allgemeinen keine der Varianten wesentliche Vorzüge. Üblicherweise kombiniert man einen write-through

[6] z.B. mit Kohärenzprotokollen wie *I/O-through-Cache, Bus-Watching-With-Write-Through, Hardware-Transparency, Non-Cacheable-Memory*

Cache mit einer No-Allocate-On-Write (siehe Abbildung 2.6) und einen write-back Cache mit einer Allocate-On-Write Strategie (siehe Abbildung 2.7).

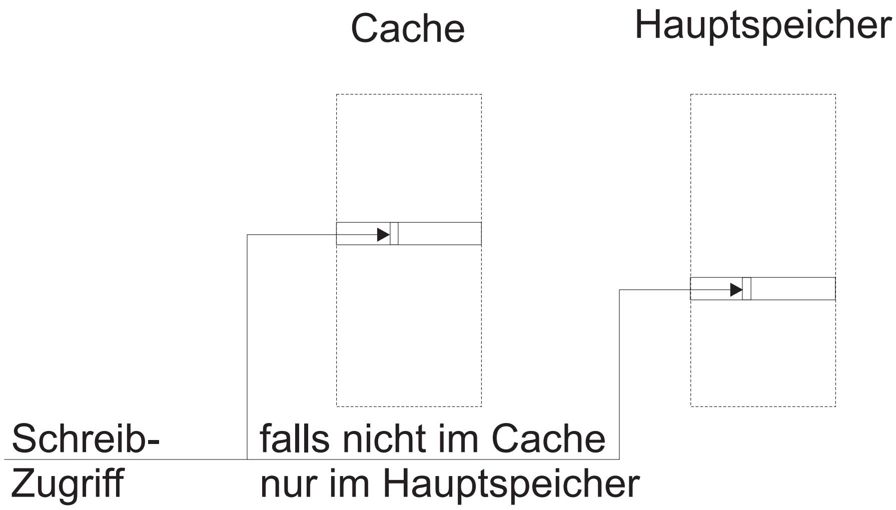


Abbildung 2.6: write-through mit No-Allocate-On-Write Strategie

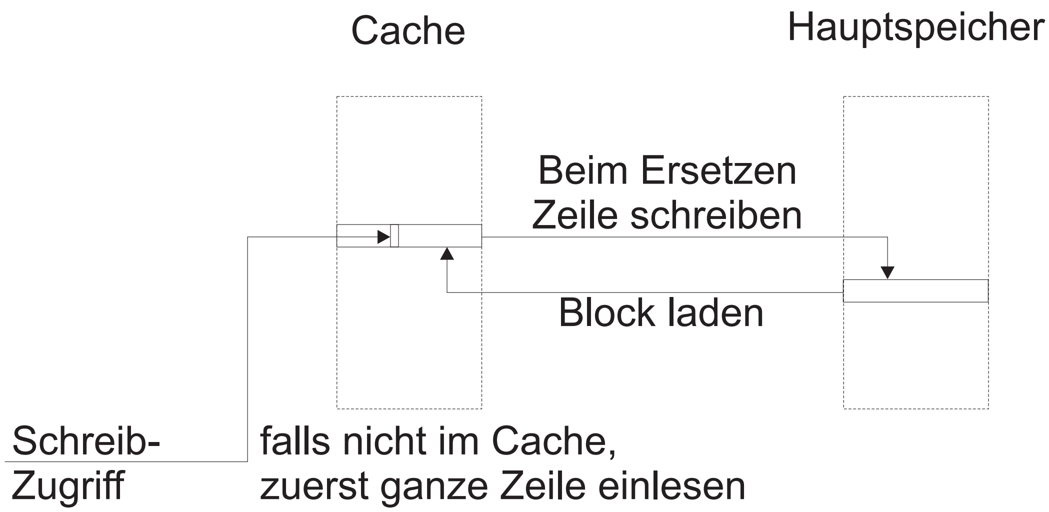


Abbildung 2.7: write-back mit Allocate-On-Write Strategie

2.4.5 Zeilengröße

Die Größe einer Cache-Zeile spielt in Bezug auf die Cache-Performance eine wichtige Rolle. Wie im Abschnitt 2.2 *Lokalitätseigenschaft von Referenzen* dargelegt, ist die Wahrscheinlichkeit, dass eine Hauptspeicheradresse zukünftig referenziert wird, umso größer, je näher diese bei der aktuellen Referenz liegt. Da die Zeilengröße bestimmend dafür ist, welche Umgebung einer Referenz mit in den Cache geladen wird, hat sie einen großen Einfluß auf die miss-ratio.

Anfänglich wird eine größere Zeile eine bessere miss-ratio bedeuten, da mehr Daten in den Cache geladen werden, auf die sich zukünftige Referenzen wahrscheinlich beziehen. Ab einem bestimmten Punkt bewirkt eine weitere Vergrößerung jedoch das Gegenteil, da größere Zeilen erstens bedeuten, dass es insgesamt weniger Zeilen im Cache gibt und diese dadurch schneller wieder überschrieben werden und zweitens jedes Wort, das weiter von der aktuellen Referenz entfernt ist, auch weniger wahrscheinlich in der Zukunft referenziert wird[7].

Die Komplexität der Abhängigkeit der Beziehung zwischen Zeilengröße und miss-ratio von den Lokalitätseigenschaften des jeweiligen Programms, macht es unmöglich, eine optimale Größe zu finden. Diverse Studien zeigen aber, dass eine Zeilengröße zwischen 4 und 8 Worten heute nahe bei einem Optimum liegt [Stall96].

2.4.6 Cache-Anzahl

Bei der Einführung von Cache-Speichern hatte jedes „typische“ System einen einzigen Cache. In modernen Systemen gibt es dagegen meist mehrere davon. Dieser Abschnitt beschreibt kurz zwei dieser Designvarianten.

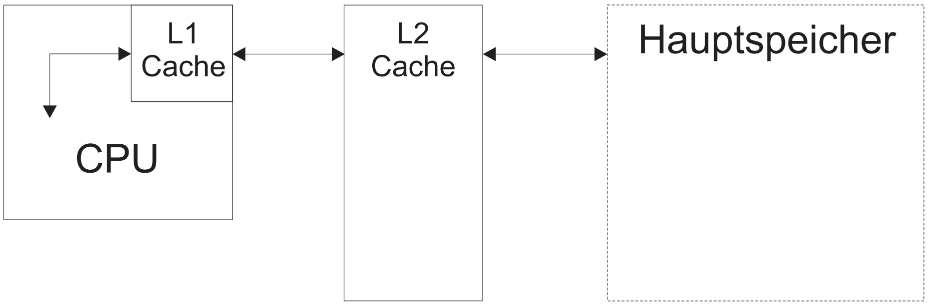


Abbildung 2.8: Two Level Cache

Single-/Two-Level-Caches

Im Laufe der Zeit wurde es durch die zunehmende Logikdichte bei der IC Herstellung möglich, Cache-Speicher auf dem Prozessorchip unterzubringen. Dies bietet natürlich eine Reihe von Vorzügen, da dieser ohne externe Buszugriffe erreicht werden kann und so die Performance erheblich steigert. Die Größe des Cache ist dabei durch die verfügbare Chipfläche begrenzt und in der Regel eher klein, was eine nicht allzugute miss-ratio vermuten läßt.

[7] Dieses Phänomen wird auch *Cache-Pollution* genannt

Man benutzt deshalb häufig noch eine weitere, externe Cache-Stufe, die diesen Beschränkungen nicht unterliegt und dafür sorgt, dass ein miss im Onchip-Cache trotzdem schnell bedient werden kann (Abbildung 2.8 auf der vorherigen Seite). Wenn die Implementierug schnell genug ist und die Daten aus der zweiten Stufe geliefert werden können, kann das sogar eine zero-wait-state-transaction, also der schnellstmögliche Buszugriff sein.

Die beiden Stufen werden *Level 1* oder *L1* Cache bei der Onchip-Variante und *Level 2* oder *L2* beim externen Cache genannt. Die möglichen Performance-Steigerungen durch den Einsatz eines Two-Level-Cache hängen von den miss-ratios beider Stufen ab und sind durchaus beachtlich.

Unified- / Split-Cache

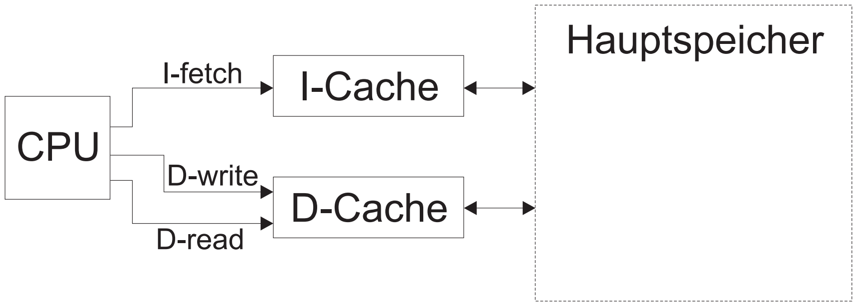


Abbildung 2.9: Split-Cache: Instruction-Cache und Data-Cache

Eine weitere Designalternative ist die Unterteilung eines Caches in einen *Instruction-* und einen *Data-Cache* (siehe Abbildung 2.9). Es ist zwar so, dass bei gegebener Cache-Größe ein *Unified-Cache* eine niedrigere miss-ratio als ein geteilter hat, da er seinen Speicherplatz automatisch dem aktuellen workload anpassen kann. Die meisten modernen Architekturen bevorzugen aber trotzdem die zweite Alternative.

Ein getrennter Instruction-Cache hat besonders bei superscalaren Prozessoren Vorzüge, da er eine, von der execution unit ungestörte, effiziente Nutzung der Instruction-Pipeline ermöglicht.

2.5 Trace-gesteuerte Cache-Simulation

Trace-gesteuerte Cache-Simulation ist eine sehr effiziente Art, das Verhalten einer Speicherstruktur zu untersuchen. Ein Trace ist dabei eine Aufzeichnung aller Speicherreferenzen eines ausgeführten Programmes, die jeweils mit einer Marke versehen sind, die angibt, um welche Art von Zugriff[8] es sich handelt.

Mit solchen Traces steuert man dann ein Simulationsmodell der gewünschten Speicher-

[8] z.B. instruction-fetch, data-fetch, data-store, ...

struktur (Cache, Hauptspeicher, . . .). In diesem Simulationsmodell kann man dann sehr einfach beliebige Designparameter testen.

Trace-gesteuerte Simulation wird oft einer direkten Messung[9] vorgezogen, da diese den Zugang zur konkreten Maschine erfordert, sehr viel aufwendiger und kostspieliger ist, und es auch kaum ermöglicht, Hardware-Designparameter zu variiren, was bei einer Simulation dagegen problemlos möglich ist. Trace-gesteuerte Simulationen sind außerdem beliebig reproduzierbar. Sie erfordern keinen Zugang zur untersuchten Hardware, ja noch nicht einmal deren Existenz.

Messungen werden dagegen oft zur Verifikation der Ergebnisse benutzt, da sie wiederum einen realitätsnahen workload (incl. supervisor code, interrupts, context switches, etc.) repräsentieren, der mit Traces sehr schwer zu modellieren[10] ist.

Die in dieser Arbeit entwickelten Simulations-Komponenten verwenden ebenfalls Traces zur Simulationssteuerung.

[9] Hardwaremonitoring

[10] Multitasking kann durch ein „Mischen“ mehrerer Traces oder ein flushen des Caches in regelmäßigen Abständen simuliert werden.

3 Simulator Beans

Das Ziel dieser Arbeit ist, wie bereits im Kapitel 1 *Einleitung* dargestellt, die Entwicklung von Softwarekomponenten, aus denen trace-gesteuerte, visuelle Simulatoren für eine breite Palette von Cache-Architekturen möglichst einfach und mit Hilfe visueller Entwicklungswerkzeuge aufgebaut werden können. Diese Komponenten müssen natürlich wiederverwendbar und möglichst einfach zu konfigurieren sein.

Über die eigentlichen Funktionen eines Cache-Simulators hinaus, sollte es zudem möglich sein, die Komponenten, oder zumindest Teile von ihnen, später in ein größeres Simulationsprojekt[1] einzubinden. Der Simulator kann sich also nicht nur darauf beschränken, die für die Statistik und Visualisierung relevanten Aspekte abzubilden. Er muß sich wie ein „realer" Cache verhalten und im Gegensatz zu den meisten anderen Cache-Simulatoren, wirklich Daten speichern[2].

Für die Implementierung wurde aus diesen Gründen Java™ gewählt und die Softwarekomponenten werden gemäß der JavaBeans™-Spezifikation entwickelt, was eine Verwendung mit allen visuellen Java™-Entwicklungsumgebungen, die JavaBeans™ unterstützen, ermöglicht. Der Aufbau von komplexen Cache-Konfigurationen gestaltet sich mit Hilfe dieser Werkzeuge einfach und übersichtlich. Eine „klassische" Verwendung der Komponenten ist natürlich ebenfalls möglich. Dazu wird eine entsprechende Klassenbibliothek bereitgestellt.

Um Perfomanceeinschränkungen, die bei Java™ Programmen im allgemeinen zur Zeit noch vorhanden sind, etwas abzumildern, werden alle Komponenten jeweils in einer visuellen und in einer nichtvisuellen Form getrennt bereitgestellt.

Als Entwicklungsumgebung und Testplattform kommt das Java™ 2 SDK sowie die BeanBox aus dem BDK 1.1 der Firma Sun Microsystems, Inc.[3] zum Einsatz. Im Vergleich zu anderen Java™-Entwicklungsumgebungen sind diese zwar recht spartanisch ausgestattet, dafür aber für eine Vielzahl von Hardwareplattformen kostenlos verfügbar [4].

Im Folgenden werden zunächst kurz JavaBeans™ im allgemeinen, der abstrakte Aufbau des Cache-Simulators im Überblick und schließlich die einzelnen Simulator-Beans im Detail dargestellt.

[1]Modellierung und Simulation eines Rechnersystems

[2]Was natürlich für eine reine trace-gesteuerte Cache-Simulation überhaupt keinen Sinn macht, da dabei nur die Statusinformationen sowie die Tags benötigt werden.

[3]http://www.sun.com/

[4]http://java.sun.com/products/jdk/1.2/index.html

3.1 JavaBeans™

Die Idee einer *Komponententechnologie* für Software ist schon Jahrzehnte alt. Da es unproduktiv ist, Standardaufgaben bei jedem Softwareprojekt neu zu entwickeln, liegt der Wunsch nach wiederverwendbaren Softwarekomponenten nahe. Die Entwicklung von Funktions- und Klassenbibliotheken war ein erster Schritt in diese Richtung. Ein weiterer sind Entwicklungssysteme, die es über eine grafische Benutzeroberfläche erlauben, Softwarekomponenten visuell, wie mit einem Baukasten zusammenzusetzen, persistent zu konfigurieren und notwendige Interaktionen zwischen ihnen mit relativ wenig Aufwand zu realisieren[5].

Um eine entsprechende Komponententechnologie auch für Java™ zu ermöglichen, erstellte Sun die JavaBeans™-Spezifikation. Darin werden allgemeine Anforderungen an *Beans*, wie die entsprechenden Softwarekomponenten treffenderweise genannt werden, festgelegt, die wie *Entwurfsmuster* verstanden werden sollen und bei ihrer Einhaltung eine Zusammenarbeit mit visuellen Entwicklungssystemen unterschiedlicher Hersteller und Plattformen ermöglichen.

Beans, die dieser Spezifikation entsprechen, müssen mindestens die folgenden Kriterien erfüllen:

- Bereitstellung eines öffentlichen, parameterlosen Konstruktors.
- Alle Eigenschaften der Bean, sogenannte *Properties*, müssen über öffentliche *getter*- und *setter*-Methoden abgefragt und zugewiesen werden können, deren Signaturen ebenfalls vorgegeben sind.
- Eine Serialisierung und Deserialisierung muß möglich sein.
- Eine Kommunikation mit anderen Beans oder einer Umgebung wird über das Event-Modell von Java™ 1.1 realisiert, die dazu benötigten öffentlichen Methoden und Schnittstellen sollten ebenfalls einem vorgegebenen Signaturschema entsprechen.

In visuellen Entwicklungssystemen können solche Beans dann, wie in einem Baukasten verwendet und aus ihnen komplette Anwendungen oder Applets erstellt werden, ohne eine einzige Codezeile schreiben zu müssen.

Eine detaillierte Darstellung der Bean-Technologie und ihrer Anwendungsmöglichkeiten kann man z.B. in [Rod98, Doh99, Wil99] finden. Die JavaBeans™-Spezifikation von Sun liegt normalerweise dem BDK bei, kann aber auch aus dem Web[6] bezogen werden.

Die in dieser Arbeit entwickelten Softwarekomponenten entsprechen ebenso wie alle *AWT*- und *SWING*-Komponenten der JavaBeans™-Spezifikation. Zur vereinfachten Anwendung als Klassenbibliothek werden jedoch noch weitere, über den Rahmen der Spezifikation hinausgehende, öffentliche Methoden und Konstruktoren implementiert.

[5] Man spricht dann auch nicht mehr unbedingt von *programmieren* sondern von *zusammenklicken*.

[6] http://java.sun.com/beans/spec.html

3.2 Abstrakter Aufbau eines Cache-Simulators

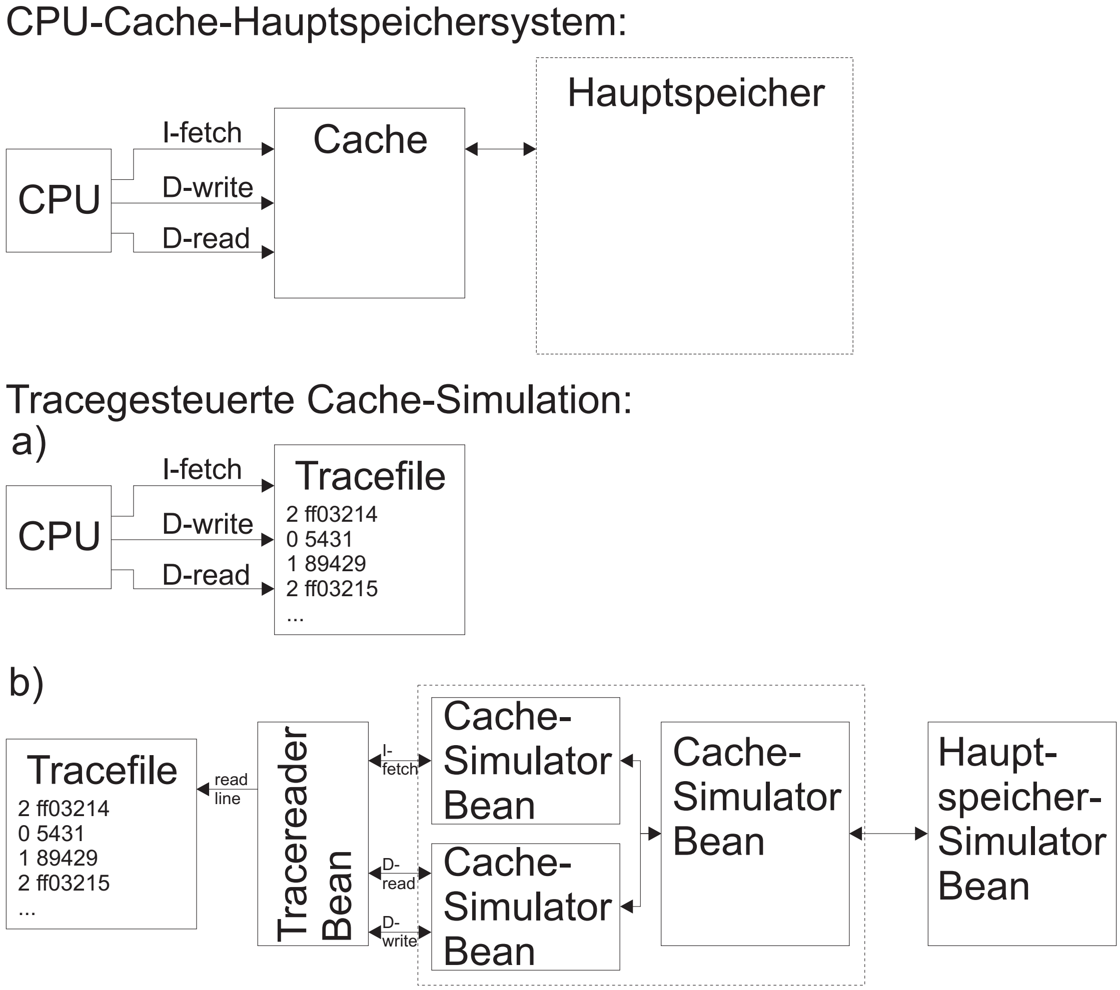


Abbildung 3.1: Tracegesteuerter Cache-Simulator: Abstrakter Aufbau

Der abstrakte Aufbau eines tracegesteuerten Cache-Simulators wird in Abbildung 3.1 dargestellt. Der eigentliche Simulator wurde dabei in drei Komponenten (Beans) aufgeteilt:

Tracereader: Bean zum Lesen eines Tracefiles und zur Weitergabe der entsprechend aufbereiteten Speicherreferenzen, was je nach Konfiguration getrennt in *Instruction*- und *Data*-Referenzen erfolgen kann.

Cache-Simulator: Konfigurierbare Modellierung eines Caches. Diese Bean nimmt Datenanforderungen entgegen, bearbeitet und leitet sie gegebenenfalls an eine nachfolgende Bean weiter.

Hauptspeicher-Simulator: Bean zur Simulation eines Hauptspeichers. Sie nimmt Datenanforderungen entgegen und bestätigt sie.

Die Kommunikation zwischen den einzelnen Beans erfolgt gemäß der JavaBeans™-Spezifikation über *Events* und dem Austausch von sogenannten *MemoryItems*.

3.3 Eventmodell

Mit Java™ 1.1 wurde das sogenannte *Delegation Event Model*[7] eingeführt, dass ein effizientes Eventhandling ermöglicht. Seine Basis beruht auf drei Konzepten, für die jeweils eine entsprechende abstrakte Klasse existiert:

Event Object: In diesem Objekt wird die Eventbenachrichtigung zusammen mit relevanten Daten gekapselt und von einem Sender (*event source*) zu den interessierten Empfängern (*event listeners*) übertragen.

Event Source: Ein Objekt, das den betreffenden Event generiert und Mechanismen zur Registrierung interessierter Zuhöhrer, den sogenannten *event listeners*, bereitstellt.

Event Listener: Objekte, die dieses Interface implementieren und sich bei der entsprechenden event source registrieren, werden beim Auftreten des Events benachrichtigt.

Events werden in diesem Modell nur noch dann erzeugt, wenn interessierte listeners vorhanden sind und auch nur an diese gesendet, was eine wesentliche Verbesserung zum alten Java™ 1.0 Eventmodell darstellt. Außerdem bietet es sowohl *single-* als auch *multicasting* von Events.

Für die Kommunikation der Simulator Beans werden die folgenden Event Objekte und Schnittstellen definiert:

DemandEvent: Dieses Event Objekt wird bei der Anforderung von Speicherzugriffen von entsprechenden event sourcen (ReadTrace oder Cache) erzeugt und enthält die Art des Zugriffs, den aktuellen Takt und das betreffende *MemoryItem*[8], das geschrieben oder gelesen werden soll.

DemandPerformedEvent: Wenn eine Anforderung eines Speicherzugriffs erfolgreich bearbeitet wurde, generieren entsprechende event sourcen (Cache oder Memory) dieses Event Objekt. Es enthält analog zum DemandEvent Objekt die Art des Zugriffs, den Tackt der Anforderung und das betreffende *MemoryItem*, das geschrieben oder gelesen wurde.

[7] Event sources werden in diesem Modell an interessierte listeners *delegiert* - daher der Name.

[8] Klasse zur Kapselung von Hauptspeicherdaten, enthält sowohl die Hauptspeicheradresse als auch den Inhalt der Speicherzelle.

ReaderEvent: Zur Kommunikation einer ReadTrace Bean mit einer eventuellen Umgebung wird dieses Event Objekt genutzt. Es enthält verschiedene Statusinformationen, wird nur von ReadTrace Objekten erzeugt und in definierten Intervallen an registrierte Listener gesendet.

DemandEventListener: Schnittstelle, die von Objekten implementiert werden muß, die Speicherzugriffsanforderungen jeder Art bearbeiten. Sie müssen eine öffentliche Methode *handleDemandEvent* bereitstellen.

DemandInstructionListener: Objekte, die Instructionfetches behandeln, implementieren diese Schnittstelle und stellen ebenfalls eine öffentliche Methode *handleDemandEvent* bereit.

DemandPerformedListener: Schnittstelle zur Rückmeldung einer erfolgreich bearbeiteten Speicherzugriffsanforderung. Objekte die DemandEvents generieren, sollten auch dieses Interface implementieren und in einer öffentlichen Methode *handleDemandPerformedEvent* auf die Rückmeldung reagieren.

ReaderListener: Am Status der Tracebearbeitung interessierte Instanzen, können diese Schnittstelle implementieren und in einer öffentlichen Methode *checkReader* die entsprechenden Informationen entgegennehmen.

Abbildung 3.2 zeigt ein Beispiel einer Kommunikation der Simulator Beans mit den definierten Event Objekten und Schnittstellen. Neben der Festlegung der gewünschten Cache-Designparameter beschränkt sich die reale Verknüpfung der Beans, also der Aufbau eines konkreten Simulationsmodells, mit einem visuellen Entwicklungstool wie zum Beispiel der BeanBox auf das „Zeichnen" der Pfeile.

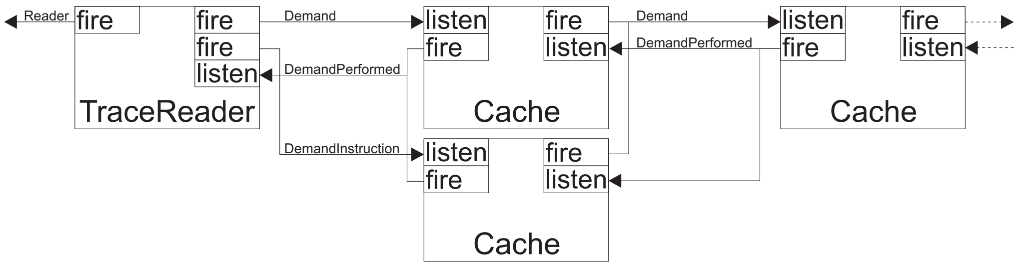


Abbildung 3.2: Beispiel Bean-Kommunikation mit Events

Im Reset dieses Kapitels werden die einzelnen Simulator Beans jeweils in ihrer visuellen und nichtvisuellen Variante näher dargestellt. Ihre konkrete Anwendung zur Modellierung von unterschiedlichen Cache-Architekturen wird dann im Kapitel 4 *Anwendung und Ergebnisse* beschrieben.

3.4 Simulator Beans im Detail

3.4.1 ReadTrace

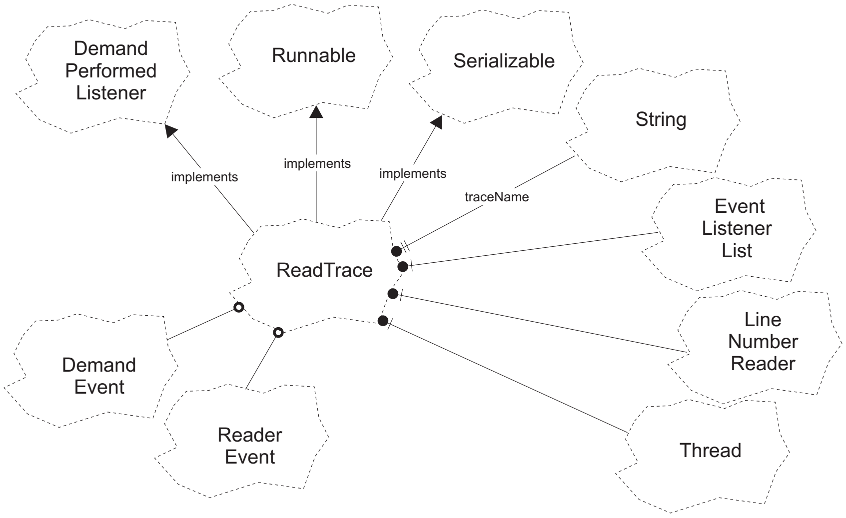


Abbildung 3.3: Klassendiagramm: ReadTrace

Die ReadTrace Bean (Abbildung 3.3) implementiert eine nichtvisuelle Komponente zur Steuerung eines Simulationsmodells unter Verwendung eines Tracefiles im verbreiteten *din* Format von *dineroIII*[9]. In diesem Format sind alle Speicherreferenzen mit einem Label versehen, der angibt, um welche Art von Zugriff[10] es sich handelt. ReadTrace liest das File zeilenweise, decodiert die Zugriffsart und generiert daraus jeweils ein entsprechendes DemandEvent Objekt.

Bean Properties

traceName: Über diese Property wird der Dateiname des zu bearbeitenden Tracefiles bestimmt.[11]

Registrierbare Listener

- DemandEventListener
- DemandInstructionListener
- ReaderListener

[9] dineroIII ist seit vielen Jahren der Standard für tracegesteuerte Cache-Simulation.
[10] 0-read data, 1-write data, 2-instruction fetch, 4-cache flush
[11] Spezifikationsgemäß werden für alle Bean Properties entsprechende *getter*- und *setter*-Methoden implementiert. Ihre Signaturen entsprechen dem folgenden Schema: *public void setPropertyName(PropertyType t)* bzw. *public PropertyType getPropertyName()*.

Implementierte Listener-interfaces

- DemandPerformedListener

Besonderheiten

ReadTrace filtert beim Decodieren der Speicherzugriffsart *instruction fetches* aus und sendet sie über die *DemandInstructionListener-Schnittstelle* zur Bearbeitung an interessierte Listener, falls solche registriert sind. Ist das nicht der Fall, wird der Event über die DemandListener Schnittstelle weitergeleitet. Dadurch wird es sehr einfach und ohne zusätzlichen Aufwand möglich, Split-Cache-Architekturen zu modellieren.

3.4.2 VisualReadTrace

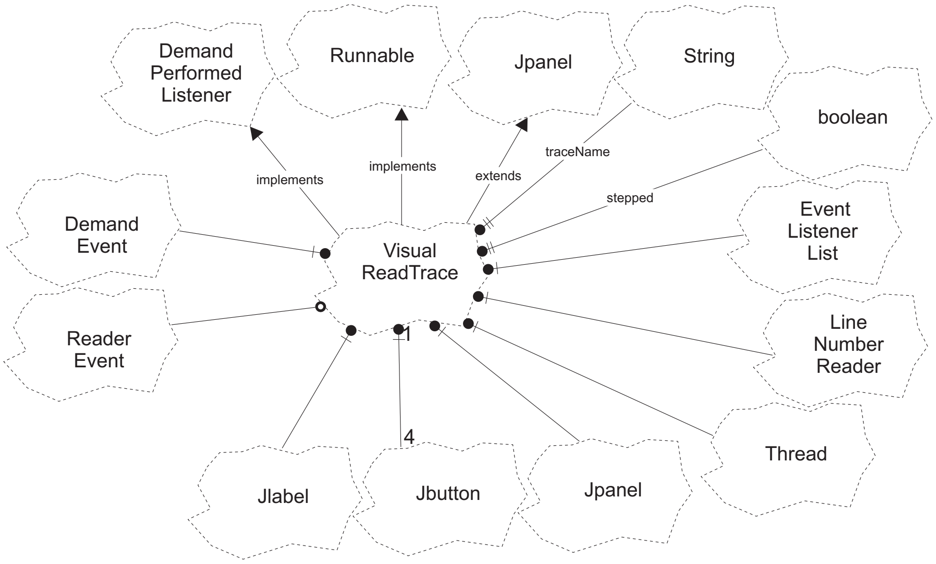


Abbildung 3.4: Klassendiagramm: VisualReadTrace

Die Bean VisualReadTrace (Abbildung 3.4) bietet die gleiche Funktionalität wie ReadTrace, erweitert um eine grafische Anzeige und Steuerungsmöglichkeiten. Dabei kann zwischen einer einzelschrittweisen und einer fortlaufenden Simulation gewählt werden.

Bean Properties

traceName: Über diese Property wird der Dateiname des zu bearbeitenden Tracefiles bestimmt.

stepped: Property zur Auswahl zwischen Einzelschritt (stepped = true) und kontinuierlicher Simulation (stepped = false),

Registrierbare Listener

- DemandEventListener
- DemandInstructionListener

Implementierte Listener-Interfaces

- DemandPerformedListener

3.4.3 Cache

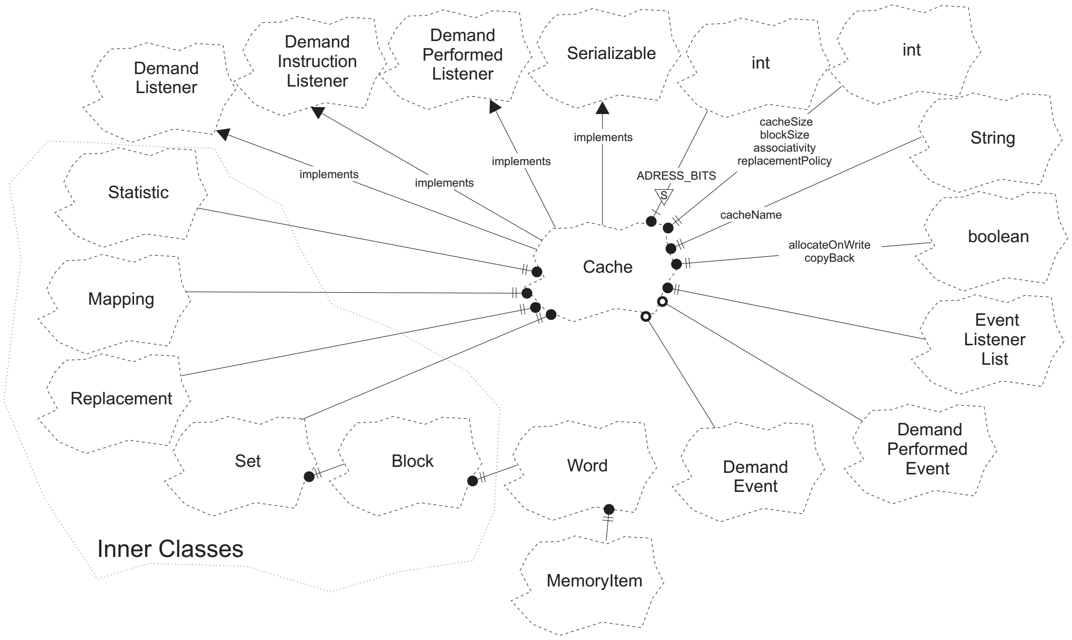


Abbildung 3.5: Klassendiagramm: Cache

Die Cache Bean (Abbildung 3.5) implementiert ein universelles, über seine Properties konfigurierbares Cachemodul. Speicherzugriffsanforderungen werden über die definierten Schnittstellen entgegengenommen, bearbeitet und gegebenenfalls weitergeleitet. Der Cache wird dabei nicht nur rein statistisch simuliert, sondern es werden Daten (in *MemoryItems* gekapselt) gespeichert und zwischen den Komponenten über das Eventsystem weitergereicht, um eine später geplante Verwendung in einem größeren Simulationsprojekt zu ermöglichen.

Die Steuerung der Simulation wird durch eine (in der Eventkette) vorhergehende Einheit geregelt. Im Moment stehen dafür die Komponenten *ReadTrace* und *VisualReadTrace* zur Verfügung, die eine tracegesteuerte Simulation ermöglichen. Da die Cache Bean davon aber nicht abhängig und ein definiertes Eventmodell vorhanden ist, könnten durchaus auch andere Simulationsarten mit ihr realisiert werden.

Bean Properties

cacheName:	Name der Cachekomponente
cacheSize:	Simulierte Cachegröße in Byte
blockSize:	Größe eines simulierten Hauptspeicherblocks (= Zeilengröße im Cache) in Byte
associativity:	Assoziativität des simulierten Caches (*direct mapped*: 1; *fully associative*: cacheSize/blockSize)
replacementPolicy:	Auswahl des Ersetzungsalgorithmus (0-LRU; 1-Random; 2-FiFo; 3-LFU)
allocateOnWrite:	Ladeverhalten bei Schreibzugriffen
copyBack:	Rückschreibstrategie

Registrierbare Listener

- DemandEventListener
- DemandInstructionListener
- DemandPerformedListener

Implementierte Listener-Interfaces

- DemandEventListener
- DemandInstructionListener
- DemandPerformedListener

3.4.4 VisualCache

VisualCache (Abbildung 3.6 auf der nächsten Seite) ist eine Bean mit gleicher Funktionalität wie Cache, ergänzt um eine grafische Darstellung des Cachedesigns. Schreib- und Lesevorgänge werden pro Speicherzelle dargestellt. Dabei kann über die Property *resetWordDisplay* bestimmt werden, ob und nach welchem Zeitintervall diese Anzeige automatisch wieder gelöscht werden soll.

Um die Vorgänge in den Kontrollstrukturen hervorzuheben, werden pro Cachezeile zudem der Zeitpunkt (Takt) der ersten und letzten Referenz, ein Referenzierungszähler und das dirty flag angezeigt.

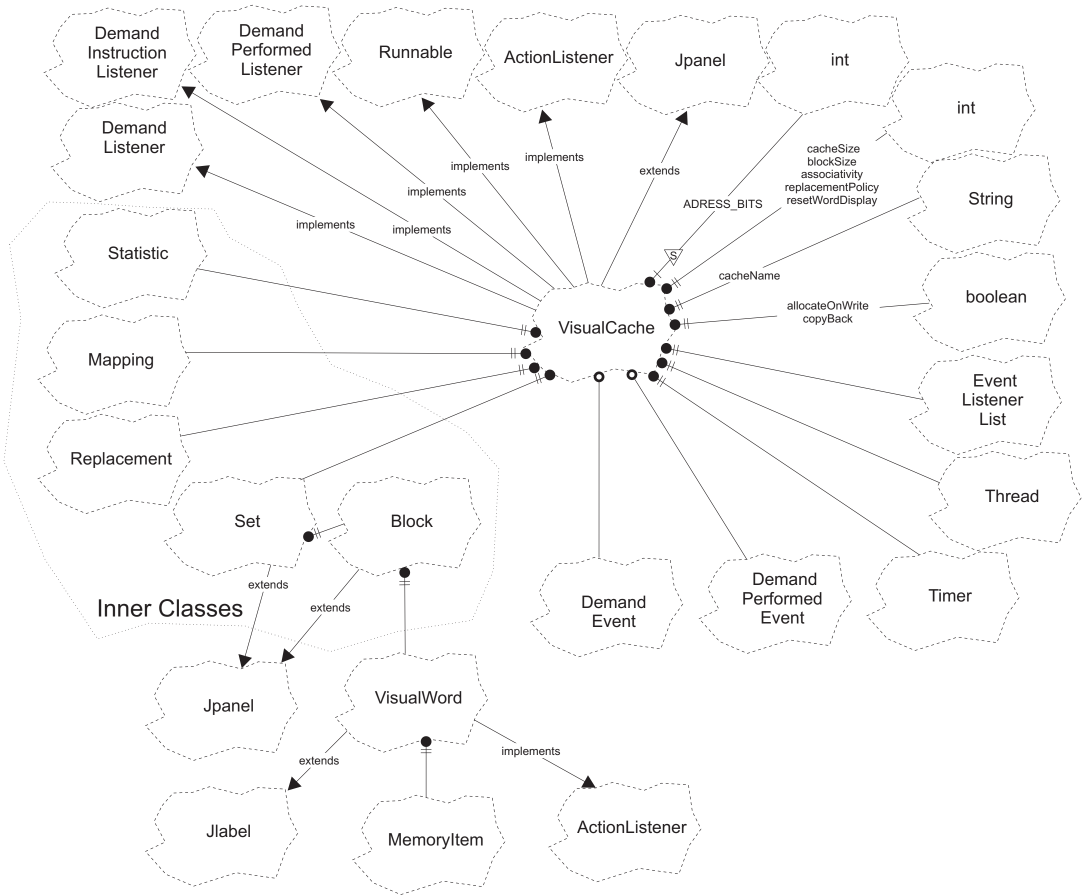


Abbildung 3.6: Klassendiagramm: VisualCache

Bean Properties

cacheName:	Name der Cachekomponente
cacheSize:	Simulierte Cachegröße in Byte
blockSize:	Größe eines simulierten Hauptspeicherblocks (= Zeilengröße im Cache) in Byte
associativity:	Assoziativität des simulierten Caches (*direct mapped*: 1; *fully associative*: cacheSize/blockSize)
replacementPolicy:	Auswahl des Ersetzungsalgorithmus (0-LRU; 1-Random; 2-FiFo; 3-LFU)
allocateOnWrite:	Ladeverhalten bei Schreibzugriffen
copyBack:	Rückschreibstrategie

resetWordDisplay: Automatisches Löschen der Zugriffsdarstellung pro Speicherzelle (0-kein automatisches Löschen)

Registrierbare Listener

- DemandEventListener
- DemandInstructionListener
- DemandPerformedListener

Implementierte Listener-Interfaces

- DemandEventListener
- DemandInstructionListener
- DemandPerformedListener
- (ActionListener)[12]

3.4.5 Memory

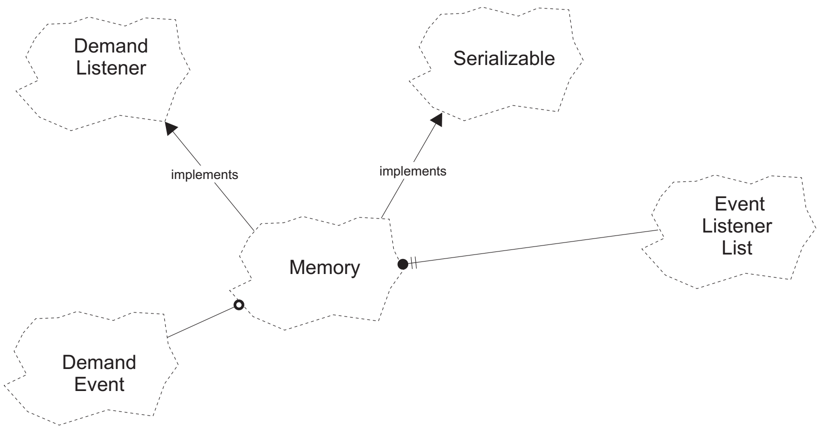


Abbildung 3.7: Klassendiagramm: Memory

Die Memory Bean (Abbildung 3.7) dient in erster Linie einem „definierten Abschluß" des Simulationsmodells. Sie bietet keine Funktionalität im eigentlichen Sinne. Demand-Events werden entgegengenommen und mittels DemandPerformedEvents bestätigt. Die erhaltenen Daten werden nicht gespeichert.

[12]nur für interne Zwecke

Im Rahmen eines umfangreicheren Simulationsmodells wird diese Bean sicherlich erweitert bzw. ersetzt.

Bean Properties

Diese Bean besitzt dieser Version keine Properties.

Registrierbare Listener

- DemandPerformedListener

Implementierte Listener-Interfaces

- DemandEventListener

3.4.6 VisualMemory

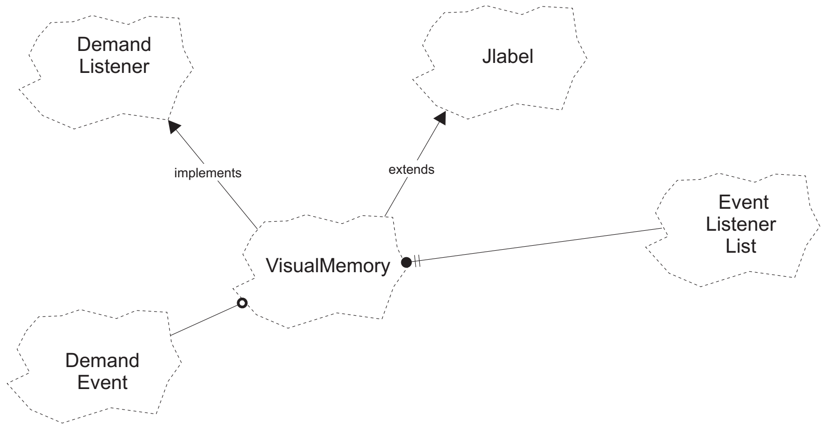


Abbildung 3.8: Klassendiagramm: VisualMemory

VisualMemory (Abbildung 3.8) ist wie Memory eine Bean zum „definierten Abschluß“ des Simulationsmodells. Sie bietet die gleiche (also keine) Funktionalität. DemandEvents werden entgegengenommen und mittels DemandPerformedEvents bestätigt. Die erhaltenen Daten werden nicht gespeichert.

Bean Properties

Diese Bean besitzt in der aktuellen Version keine Properties.

Registrierbare Listener

- DemandPerformedListener

Implementierte Listener-Interfaces

- DemandEventListener

4 Anwendung und Ergebnisse

Die vorgestellten Simulator Beans eignen sich zum Aufbau von Simulatoren unterschiedlicher Cachedesigns. Es können sowohl visuelle Simulatoren, für die didaktische Aufbereitung klein bemessener Caches, als auch nichtvisuelle für die statistische Analyse größerer Caches erzeugt werden. Dieses Kapitel zeigt die Benutzung der Simulator Beans an einigen konkreten Beispielen und präsentiert die Ergebnisse.

4.1 Anwendung der Klassenbibliothek

Um die Simulatorkomponenten als normale Java^{TM}-Klassen zu verwenden, muß das package *net.morgenstern.cachesim* entsprechend der gewählten Java^{TM}-Enwicklungsumgebung installiert und eventuell in den lokalen *CLASSPATH* aufgenommen werden. Nähere Informationen zur konkreten Installation enthält normalerweise die Dokumentation der Entwicklungsumgebung. Eine ausführliche Dokumentation der Klassenbibliothek ist ebenfalls im package *net.morgenstern.cachesim* enthalten.

4.1.1 Ein nichtvisueller Splitcache

Als erste Anwendung soll ein nichtvisueller Splitcache erstellt und getestet werden. Es werden die folgenden Designparameter gewählt:

- Cachegröße jeweils 8192 Byte
- Blockgröße 32 Byte
- Direct mapped
- Ersetzungsalgorithmus LRU
- Allocate-On-Write
- Schreibstrategie copy back

Neben den für die Anwendung mit visuellen Entwicklungstools geforderten parameterlosen Konstruktoren und Setter-methoden enthalten die Simulator Beans zusätzlich auch Konstruktoren, denen alle benötigten Properties als Parameter übergeben werden können. Diese Java^{TM}-Anwendung (Abbildung 4.1 auf der vorherigen Seite) nutzt diese Möglichkeit bei der Instanziierung der Cache Objekte.

```
import net.morgenstern.cachesim.*;

class Beispiel1 implements ReaderListener {
    //Simulationskomponenten instanziieren
    final Cache iCache = new Cache("Instructioncache", //cacheName
                                   8192,               //cacheSize
                                   32,                 //blockSize
                                   1,                  //associativity
                                   0,                  //replacementPolicy
                                   true,               //allocateOnWrite
                                   true);              //copyBack
    final Cache dCache = new Cache("Datacache",8192,32,1,0,true,true);
    final ReadTrace reader = new ReadTrace();
    final Memory memory = new Memory();

    private void initialize(String traceName) {
        //Eventkette aufbauen
        reader.addDemandInstructionListener(iCache);
        reader.addDemandListener(dCache);
        reader.addReaderListener(this);

        iCache.addDemandListener(memory);
        iCache.addDemandPerformedListener(reader);

        dCache.addDemandListener(memory);
        dCache.addDemandPerformedListener(reader);

        memory.addDemandPerformedListener(iCache);
        memory.addDemandPerformedListener(dCache);
        //Simulation starten
        reader.setTraceName(traceName);
        reader.startReading();
    }
    //Einstiegsfunktion, Tracefilename als Parameter
    public static void main(String[] args) {
        Beispiel1 simulator = new Beispiel1();
        simulator.initialize(args[0]);
    }
    //Behandlung von ReaderEvents
    public void checkReader(ReaderEvent e) {
        if (e.isFinished()) {
           iCache.printStatistic(dCache); //Kombinierte Statistikausgabe
           System.exit(0);
        }
    }
}
```

Abbildung 4.1: Beispiel 1: Ein nichtvisueller Splitcache, Sourcecode

Die Klasse *Beispiel1* instanziiert die benötigten Simulatorkomponenten mit den gewünschten Properties als Membervariablen[1] und verbindet sie in ihrer Methode *initialize()* über das definierte Eventsystem. Allein durch die Registrierung des Objekts *iCache* an der *DemandInstructionListener* Schnittstelle wird der Aufbau eines Splitcache erreicht. Diese Registrierung der Eventlistener entspricht dem „Pfeile zeichnen“ in grafischen Entwicklungstools und stellt die eigentliche Funktionalität des Simulators bereit.

Ein Testlauf der Applikation mit dem Testtrace der *dineroIII* Installation ergibt die Ergebnisse[2] in Abbildung 4.2.

```
>javac Beispiel1.java

>java Beispiel1 eg.din

Instructioncache + Datacache:
             Metrics     Total    Instrn      Data      Read     Write
----------------------------------------------------------------------
      Demand Fetches      1000       470       530       234       296
       Demand Misses        59        26        33        16        17
                        0,0590    0,0553    0,0623    0,0684    0,0574
```

Abbildung 4.2: Beispiel 1: Simulationsergebnis

4.1.2 Ein visueller Splitcache

Auch visuelle Simulatoren können ohne großen Aufwand über die Klassenfunktionen erstellt werden. Das folgende Beispiel implementiert einen visuellen Splitcache mit folgenden Designparametern:

- Cachegröße jeweils 256 Byte
- Blockgröße 16 Byte
- four way set associative
- Ersetzungsalgorithmus LRU
- Allocate-On-Write
- Schreibstrategie write trough

[1] was keinesfalls ein Muß ist...

[2] die genau den *dineroIII* Ergebnissen entsprechen. Generell wurde *dineroIII* zur Validierung der Ergebnisse benutzt.

Die Simulation soll im Einzelschrittmodus erfolgen und die beiden Cache Objekte sollen, um Bildschirmplatz zu sparen, jeweils auf einer eigenen TabPage untergebracht werden. Im Sourcecode in Abbildung 4.3 werden nur die für den Simulator wesentlichen Stellen gezeigt.

```
...
import net.morgenstern.cachesim.*;
class Beispiel2 extends JFrame {
    //Simulationskomponenten instanziieren
    final VisualCache iCache = new VisualCache("Instructioncache",
              256,16,4,0,true,false,0);
    final VisualCache dCache = new VisualCache("Datacache",
              256,16,4,0,true,false,0);
    final VisualReadTrace reader = new VisualReadTrace();
    final Memory memory = new Memory();
    public Beispiel2() {super("trace-driven cache-simulation");}
    void initialize(String traceName) {
        JTabbedPane tp = new JTabbedPane();
        ...
        reader.setTraceName(traceName);
        reader.setStepped(true);
        //Visuelle Simulatorkomponenten platzieren
        tp.add(iCache, iCache.getName());
        tp.addTab(dCache.getName(),dCache);
        contentPane.add(reader,BorderLayout.NORTH);
        contentPane.add(tp,BorderLayout.CENTER);
        //Eventkette aufbauen
        reader.addDemandInstructionListener(iCache);
        reader.addDemandListener(dCache);
        iCache.addDemandListener(memory);
        iCache.addDemandPerformedListener(reader);
        dCache.addDemandListener(memory);
        dCache.addDemandPerformedListener(reader);
        memory.addDemandPerformedListener(iCache);
        memory.addDemandPerformedListener(dCache);
    }
    public static void main(String[] args) {
        Beispiel2 simulator = new Beispiel2();
        simulator.initialize(args[0]);
    }
}
```

Abbildung 4.3: Beispiel 2: Ein visueller Splitcache, Sourcecode

Wie dieses Beispiel zeigt, ist es über das definierte Eventmodell durchaus möglich (und manchmal sogar sinnvoll), visuelle mit nichtvisuellen Komponenten zu verknüpfen.

Den Simulator dieses Beispiels in Aktion zeigt Abbildung 4.4 auf der nächsten Seite.

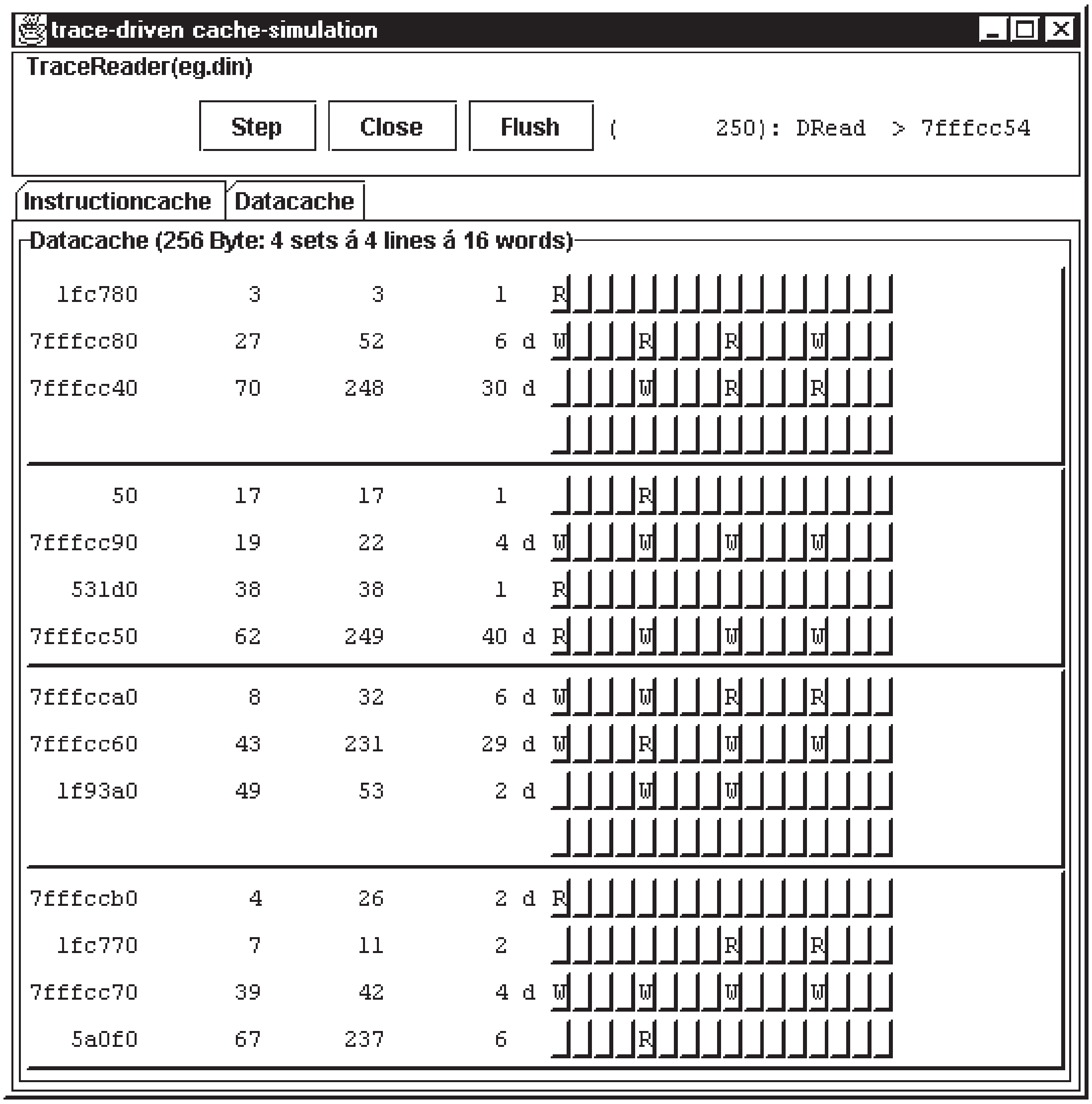


Abbildung 4.4: Beispiel 2: Ein visueller Splitcache in Aktion

Visuelle Simulatoren können natürlich auch statistische Ausgaben generieren. Das Verfahren dazu verläuft analog zu Beispiel1 über den Aufruf der Methode *printStatistic()*.

4.2 Anwendung mit grafischen Entwicklungsumgebungen

Wie bereits erwähnt, ist es mit grafischen Java™-Entwicklungsumgebungen besonders einfach, mit den Simulator Beans zu arbeiten. Dieser Abschnitt beschreibt das prinzipielle Vorgehen dabei.

Die Simulator Beans müssen einer grafischen Entwicklungsumgebung vor ihrer ersten Benutzung zunächst bekannt gemacht werden. Das Verfahren dazu differiert dabei von Tool zu Tool. Java^{TM}-Standard ist allerdings, dass alle dazu benötigten Resourcen in Form einer *jar*-Datei[3] vorliegen müssen. Diese wird in ein bestimmtes Verzeichnis[4] kopiert und dann automatisch, oder über einen speziellen Menüpunkt in das Entwicklungssystem integriert.

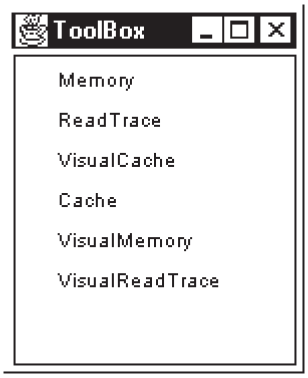


Abbildung 4.5: BeanBox Toolbox mit Simulator Beans

Die Entwicklungsumgebung analysiert dabei jede Bean mit dem Java^{TM}-Reflection-Package oder verwendet entsprechende BeanInfo Klassen, falls sie vorhanden sind[5]. Ist dies erfolgreich abgeschlossen, stehen die Beans dann als Komponenten in einer Form von *Toolbox* zur Verfügung (Abbildung 4.5).

Analog zu CAD Programmen können sie dann ausgewählt und einem Design hinzugefügt werden. Für jede definierte bzw. in der zugehörigen BeanInfo Klasse dazu vorgesehene Property, stellt die Entwicklungsumgebung dann einen *Property Editor* bereit, mit dem die Konfiguration der Bean erfolgen kann. Abbildung 4.6 auf der nächsten Seite zeigt die Property Editoren der BeanBox für die VisualCache Bean.

Die Kommunikation zwischen den einzelnen Beans, wird meist auch grafisch durch Verbinden einer Event Quelle mit einem passenden Listener festgelegt. In der BeanBox wird dazu zunächst eine Bean des aktuellen Designs als event scource ausgewählt und dann über den Menüpunkt *Edit->Events* die Eventschnittstelle bestimmt, über die die Verbindung erfolgen soll (Abbildung 4.7 auf Seite 39).

Danach zieht man im aktuellen Design eine Linie zum gewünschten Event Listener.

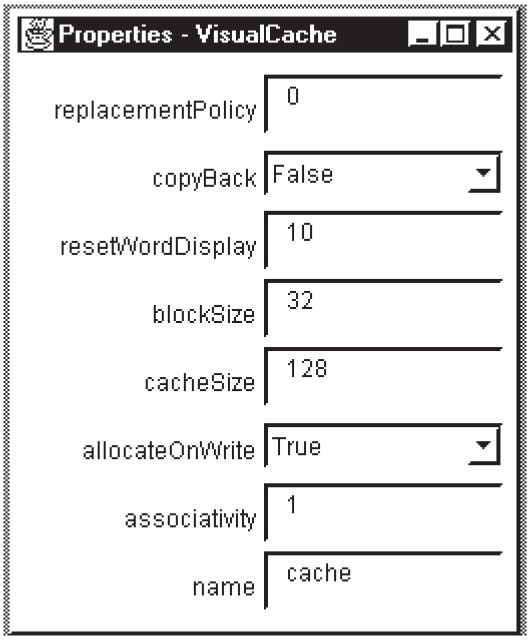


Abbildung 4.6: BeanBox Property Editoren für VisualCache Bean

Jetzt muß noch die passende Methode des Listeners aus einem Menü der verfügbaren Methoden ausgewählt werden (Abbildung 4.8 auf der nächsten Seite) und die Eventverbindung ist definiert.

Die Entwicklungsumgebung generiert dabei automatisch sogenannte *Adapter Klassen*, die quasi zwischen Event Quelle und Listener stehen und das betreffende Event Objekt weiterleiten.

Durch die Serialisierbarkeit von Beans ist es möglich, angepasste Komponenten zu erstellen und persistent zu speichern. Auch ganze Designs können natürlich gespeichert und wieder geladen werden. Zur Erstellung kompletter Applets oder Applikationen aus entsprechenden Designs gibt es in den Entwicklungsumgebungen meist auch geeignete Tools. Bei der hier verwendeten BeanBox können in der aktuellen Version allerdings nur Applets generiert werden.

Ein fertiges Design kann meist auch gleich in der Entwicklungsumgebung getestet werden. Es ist also nicht immer unbedingt nötig, „richtige" Anwendungen oder Applets zu erstellen und zu kompilieren, um einen Cache zu simulieren. Das bietet sich zum „Experimentieren" mit den Simulator Beans geradezu an.

[3] Java^{TM}-Version einer *zip*-Datei

[4] Bei der hier verwendeten BeanBox muß dies *.../Bdk1.1/jars* sein

[5] Alle hier vorgestellten Simulator Beans besitzen entsprechende BeanInfo Klassen.

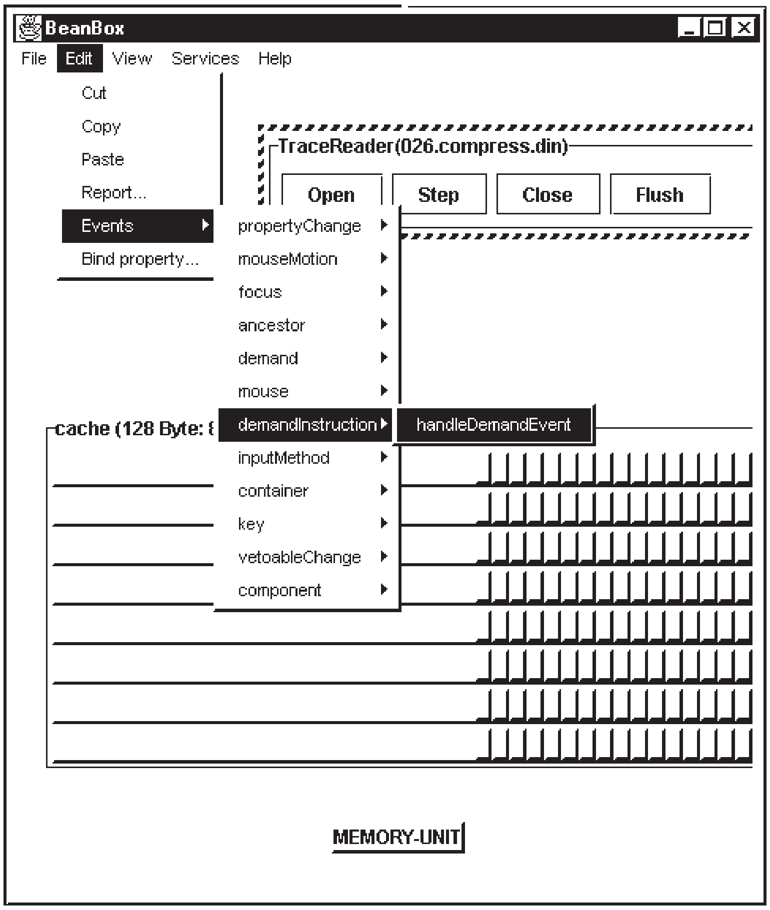


Abbildung 4.7: BeanBox Eventschnittstellen

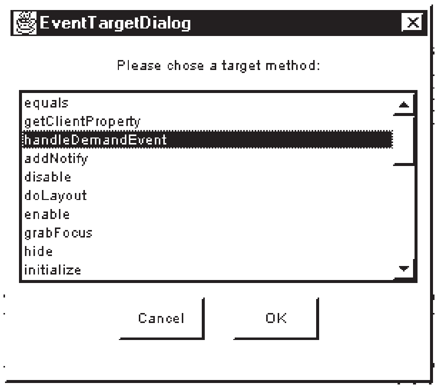
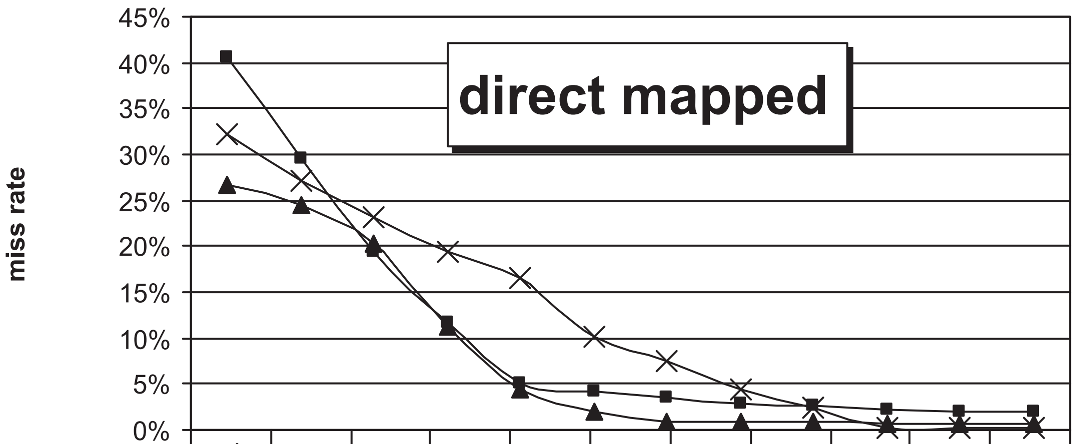


Abbildung 4.8: BeanBox Eventlistener Auswahl

4.3 Überprüfung der Ergebnisse

Wie bereits erwähnt, wurde zur Validierung der Simulationsergebnisse der Cachesimulator *dineroIII* verwendet. Es konnten natürlich nicht alle Kombinationen der unterschiedlichen Designs modelliert, simuliert und verglichen werden. Aber diese Einschränkung gilt selbst für einen so anerkannten Simulator wie *dineroIII*[6]. In vergleichbaren Designs ergaben sich dabei auch übereinstimmende Simulationsergebnisse.

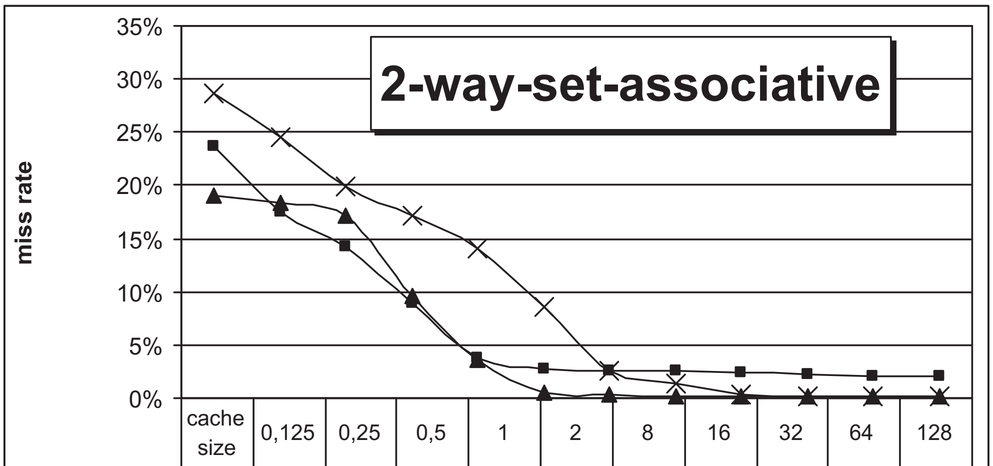
Im Rest dieses Kapitels werden beispielhaft noch einige statistische Simulationsergebnisse präsentiert. Dazu wurde mit den beschriebenen Methoden ein einstufiger, nichtvisueller Cachesimulator aufgebaut und Traces der Programme *compress*, *ora* und *alvinn* simuliert. Als Designparameter kamen eine Blockgröße von 32 Worten, ein LRU replacement und eine write allocate Strategie zum Einsatz. Die Cachegröße wurde zwischen 128 und 256k Worten variiert und jeweils mit einem direct mapping, einem 2-way und einem 4-way set associative mapping kombiniert.



	cache size	0,125	0,25	0,5	1	2	8	16	32	64	128
compress	40,602	29,581	19,324	11,762	5,105	4,085	2,809	2,593	2,295	2,091	2,067
ora	26,789	24,516	20,229	11,271	4,519	2,035	0,841	0,785	0,771	0,768	0,768
alvinn	32,277	27,103	23,189	19,509	16,451	10,242	4,426	2,324	0,225	0,208	0,208

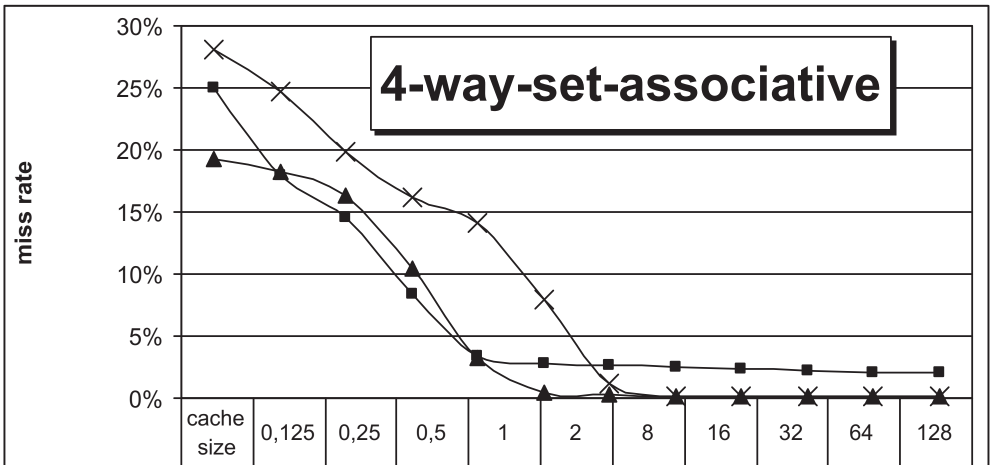
Abbildung 4.9: Simulation: Cachegröße, direct mapped

[6] siehe *dineroIII* Dokumentation von Mark D. Hill [Hil89] „Bugs: Not all combination of options have been throughly tested."



cache size	0,125	0,25	0,5	1	2	8	16	32	64	128
compress	23,676	17,566	14,307	8,989	3,717	2,816	2,501	2,334	2,200	2,031	1,990
ora	19,055	18,333	17,077	9,651	3,562	0,442	0,222	0,201	0,189	0,189	0,188
alvinn	28,641	24,554	19,905	17,095	14,014	8,500	1,289	0,332	0,157	0,157	0,157

Abbildung 4.10: Simulation: Cachegröße, 2-way-set-associative



cache size	0,125	0,25	0,5	1	2	8	16	32	64	128
compress	24,969	17,896	14,562	8,331	3,402	2,748	2,467	2,309	2,170	2,028	1,987
ora	19,288	18,216	16,379	10,472	3,199	0,379	0,198	0,190	0,188	0,188	0,188
alvinn	28,124	24,632	19,788	16,106	14,044	8,002	0,159	0,157	0,157	0,157	0,157

Abbildung 4.11: Simulation: Cachegröße, 4-way-set-associative

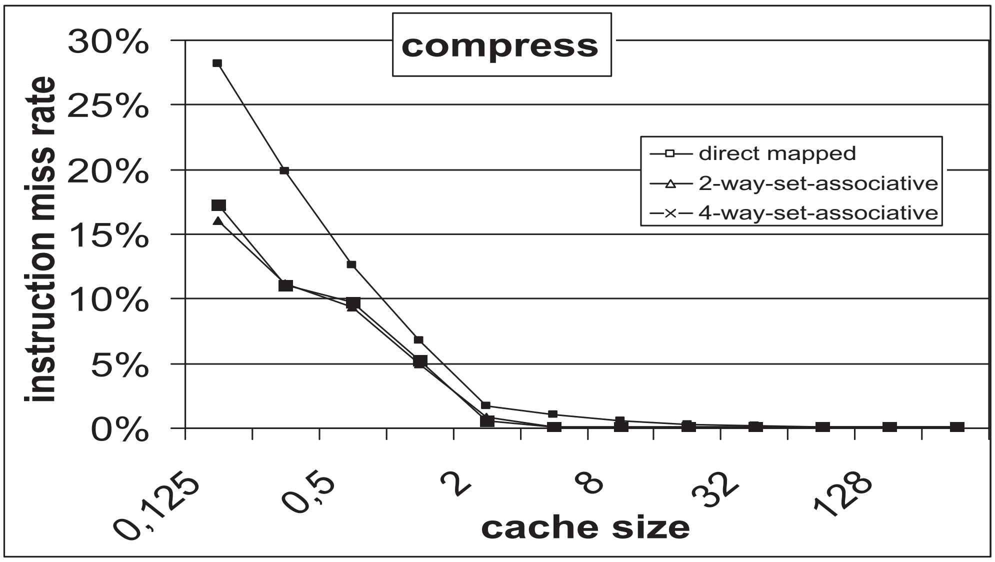


Abbildung 4.12: Simulation von Cache Größe und Assoziativität (compress, inst. miss rate)

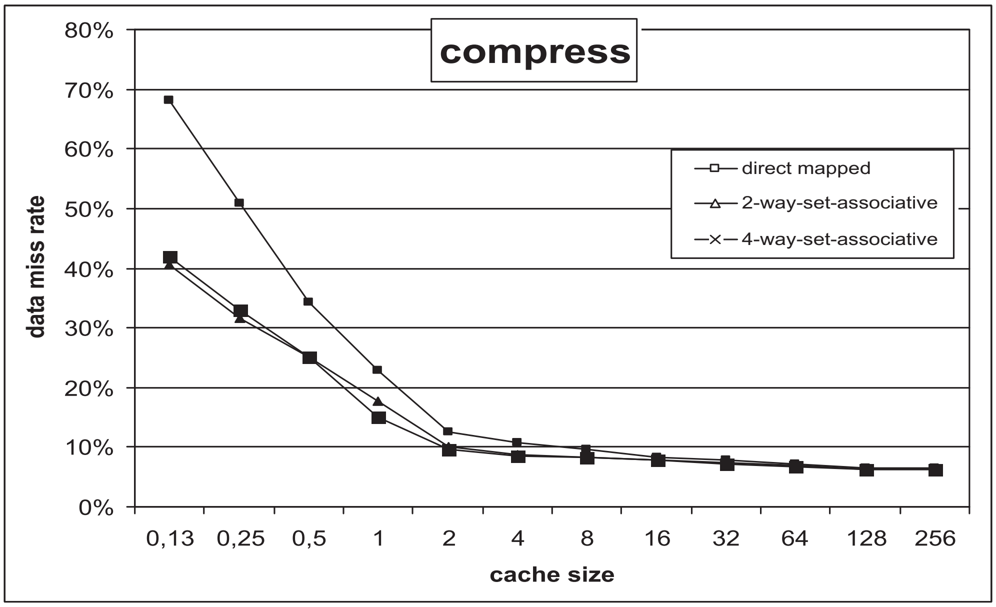


Abbildung 4.13: Simulation von Cache Größe und Assoziativität (compress, data miss rate)

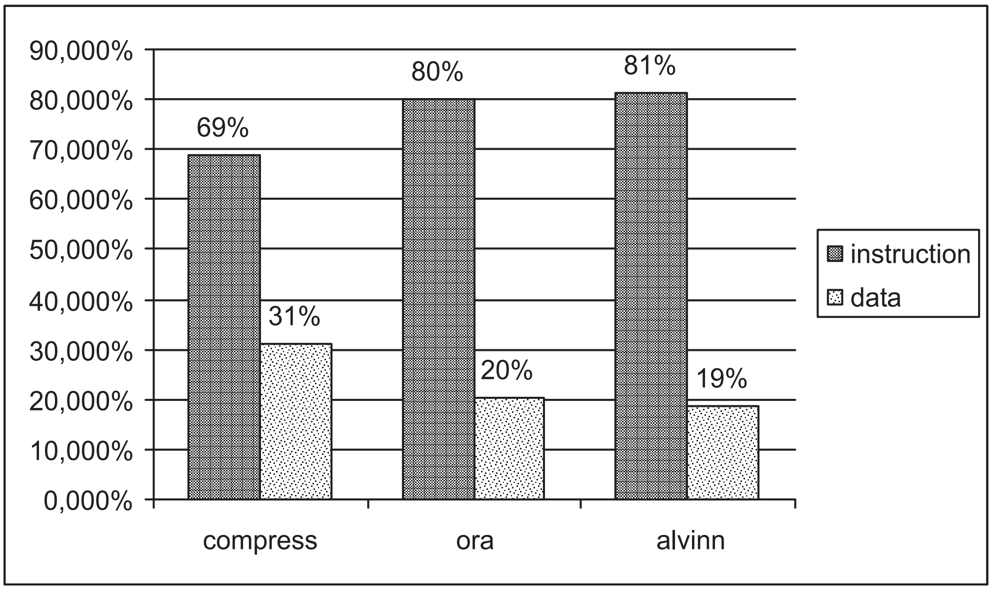


Abbildung 4.14: Simulation: Anteil Daten/Instruktionen

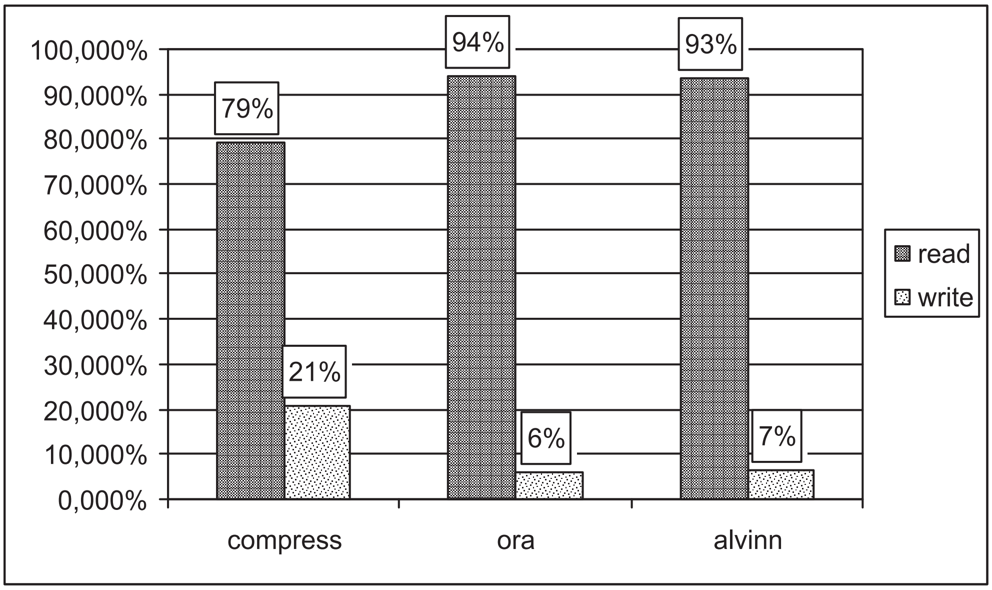


Abbildung 4.15: Simulation: Anteil Schreib-/Lesezugriffe

5 Zusammenfassung

In der vorliegenden Arbeit wurde zunächst ein Überlick der theoretischen Grundlagen von Cache Speichern allgemein und den für die hier behandelten Simulator Beans wichtigen Cache Designparametern gegeben, da er zum Verständnis ihrer Funktion nötig ist. Es gibt natürlich noch weitere Designparameter wie z.B. *sub-block placement* oder *prefetching* Strategien, die hier nicht besprochen und implementiert wurden, da dies den Rahmen dieser Arbeit gesprengt und vielleicht auch nicht ganz in ihr Konzept gepasst hätte.

Mit den entwickelten Beans kann ein breites Spektrum von Cache Architekturen modelliert und simuliert werden. Ihre Anwendung ist dank Java^{TM} und der Bean Technologie denkbar unkompliziert. Die noch vorhandenen Performanceeinschränkungen spielen bei visuellen Simulatoren für didaktische Zwecke nur eine untergeordnete Rolle und können durch neuere Java^{TM}-Technologien wie *Just-In-Time-Compiler* und verbesserte *Virtual Machines* zukünftig verringert werden.

Eventuell werden die Simulator Beans zukünftig in ein größeres Simulationsprojekt eingebettet. Einige Voraussetzungen dafür wurden bereits in der jetzigen Implementierung geschaffen. So dient das Handling und die tatsächliche Ablage von Speicherzellen (*MemoryItems*), die vom definierten Eventsystem sowie den Beans bereitgestellt wird, allein diesem Zweck.

Je nachdem, welcher Zweck mit den erstellten Simulatoren verfolgt wird, sind noch bestimmte Erweiterungen bzw. Optimierungen der Simulator Beans denkbar. Für nichtvisuelle, rein statistische Simulatoren wären eine *simultane* Simulation eines ganzen Designspektrums (z.B. mit variierter Cachegröße etc.) sicher sinnvoll, für visuelle Simulatoren dagegen überhaupt nicht. Anpassungen und Erweiterungen der Beans sind durch die definierten Klassenschnittstellen zukünftig hoffentlich ohne größeren Aufwand möglich.

Literaturverzeichnis

[Con00] Condon, Anne E., Sorin, Daniel, Plakal, Manoj, Hill, Mark D., Martin, Milo M., Wood, David A.: *Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol*, University of Wisconsin Technical Report CS-TR-2000-1412, 2000,
http://www.cs.wisc.edu/multifacet/papers/tr1412_lamport.pdf

[Denn72] Denning, P.J.: *On modeling program behavior*, In Proc. Spring Joint Computer Conference, vol. 40, AFIPS Press, Arlington, Va., 1972, Seiten 937-944.

[Dil00] Dillmann, R.: *Technische Informatik II: Speicherhierarchie*, Universität Karlsruhe, 2000,
http://wwwiaim.ira.uka.de/users/asfour/TI/TI-2/Vorlesung/Folien/vor-06.07-2F.pdf

[Doh99] Doherty, Don, Leinecker, Rick, et al.: *Java Beans Unleashed*, SAMS Publishing, Indianapolis, 1999

[Lin00] Lindenstruth, Volker: *Einführung in die Technische Informatik: Cache*, Ruprecht-Karls-Universität Heidelberg, 2000,
http://web.kip.uni-heidelberg.de/Hardwinf/I2WS00/folien/l11_cacheb.pdf

[Hen95] Hennessy, John L., Patterson, David A.: *Computer Architecture: A Quantitative Approach.* Second Edition, Morgan Kaufmann Publishers, 1995, Web-Enhanced:
http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-329-8

[Hen97] Hennessy, John L., Patterson, David A.: *Computer Organization and Design: The Hardware/Software Interface, Second Edition.* Morgan Kaufmann Publishers, 1997, Web-Enhanced:
http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-428-6

[Hil89] Hill, Mark D.: *Dinero III Manpage*, University of Wisconsin, 1989,
http://www.cs.dal.ca/ jost/CSCI4121/DineroManpage.pdf

[Muk98] Mukherjee, Shubhendu S., Hill, Mark D.: *Using Prediction to Accelerate Coherence Protocols*, International Symposium on Computer Architecture (ISCA), Barcelona, 1998,
ftp://ftp.cs.wisc.edu/wwt/isca98_cosmos.pdf

[Rod98] Rodrigues, L.H.: *The Awesome Power of Java Beans*, Manning Publications Co., Greenwich, 1998.

[Smith82] Smith, A.J.: *Cache Memories*, In: Computer Surveys, vol. 14/3, Association for Computer Machinery, 1982, S. 473ff

[Stall96] Stallings, W.: *Computer Organization and Architecture: Design for Performance*, 4th edition, Prentice Hall, New Jersey, 1996, Kapitel *Cache Memory* ab Seite 121

[Wil99] Wilhelms G., Kopp, M.: *Java professionell*, MITP-Verlag, Bonn, 1999