Multi-Port-Speichermanager für die Java-Plattform SHAP

Martin Zabel, Peter Reichel, Rainer G. Spallek
{martin.zabel, peter.reichel, rgs}@inf.tu-dresden.de
Institut für Technische Informatik, Technische Universität Dresden

Kurzfassung

Der in Hardware implementierte Speichermanager der eingebetteten Java-Plattform SHAP stellt zusammen mit dem integriertem *Garbage Collector* einen automatisch verwalteten Heap zur Verfügung. Dieser Artikel erläutert die Erweiterung des Speichermanagers, sodass mehrere Komponenten, wie DMA-Kontroller und weitere CPUs/ CPU-Kerne, gleichzeitig und unabhängig voneinander auf Objekte im Speicher zugreifen können. Aufgrund des verwendeten Indirektionskonzepts müssen mehrere Ports—einer je Komponente—implementiert werden, die jeweils die Basisadresse der aktiven Referenz für folgende Adressberechnungen zwischenspeichern und ggf. nach der Verschiebung eines Objekts aktualisieren. Die von den Ports erzeugten Low-Level-Kommandos werden arbitriert und über die Adressberechnung zum Speicherkontroller gesendet. Getrennte Busse für den Versand des Kommandos/der Schreibdaten und den Empfang der Lesedaten sowie der Einsatz von Tags ermöglichen ein Pipeling und eine Verschachtelung von Speicherzugriffen zwecks optimaler Ausnutzung der Speicherbandbreite.

1 Einleitung

Für Hardwaresysteme die objektorientierte Konzepte unterstützen, wie bspw. Java-Prozessoren, ist es zweckmäßig den Heap direkt mit einem Speichermanager zu koppeln. Dieser verwaltet autonom die abgelegten Objekte und stellt High-Level-Operationen für die Allokation sowie den Lese- und Schreibzugriff zur Verfügung. Die Freigabe von Objekten kann dabei automatisch durch einen integrierten *Garbage Collector* (GC) erfolgen. Für die Anbindung mehrerer Komponenten (Multi-Core, DMA) sind entsprechend mehrere Ports bereitzustellen, die im Idealfall unabhängig voneinander agieren können. Dieser Artikel erläutert den Aufbau eines Multi-Port-Speichermanagers am Beispiel der Java-Plattform SHAP [Zab+07], die sich aus der SHAP-Mikroarchitektur (SHAP-MA) und der Software-Toolchain SHAP-Linker zusammensetzt.

2 Objektorienter Speicherzugriff

In einem objektorientiertem Hardwaresystem erfolgt der Speicherzugriff seitens der CPU oder anderer Komponenten mittels Objektreferenz und Objektoffset. Die Berechnung der endgültigen Speicheradresse erfolgt dabei erst im Speichermanager und nicht bereits in der CPU wie bei klassischen Architekturen.

Die Objektreferenz führt eine weitere Abstraktionsebene ein, um einen automatisch verwalteten Heap implementieren zu können. Die Referenz kann dabei sowohl direkt der Basisadresse des Objekts entsprechen, oder über eine Tabelle o. ä. indirekt auf die Basisadresse verweisen (Abb. 1). Die Entscheidung für eine der beiden Varianten hängt maßgeblich davon ab, ob zur Kompaktierung des Heap-Speichers der GC Objekte im Speicher verschieben muss oder nicht.

Ist eine Verschiebung von Objekten nicht notwendig (*Non-Copying Garbage Collection*), so kann die Referenz immer der Basisadresse entsprechen. Die Addition von Basisadresse und Offset kann in diesem Fall auch in die CPU oder dergleichen verlagert werden.

Müssen im Gegensatz dazu jedoch Objekte verschoben werden (*Copying Garbage Collection*), ist eine Aktualisierung der Basisadresse erforderlich. Entspricht die Referenz der Basisadresse, so sind folglich alle Referenzen zu aktualisieren. Bei Einsatz des Indirektionskonzepts ist dagegen nur der Tabelleneintrag zu aktualisieren.

In der SHAP-MA wird eine *Copying Garbage Collection* eingesetzt, um die Allokation von Objekten in konstanter und kurzer Zeit zu gewährleisten (vgl.

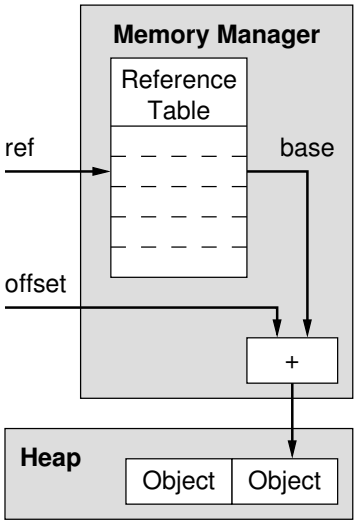

Abbildung 1: Indirektionskonzept für den Zugriff auf Objekte

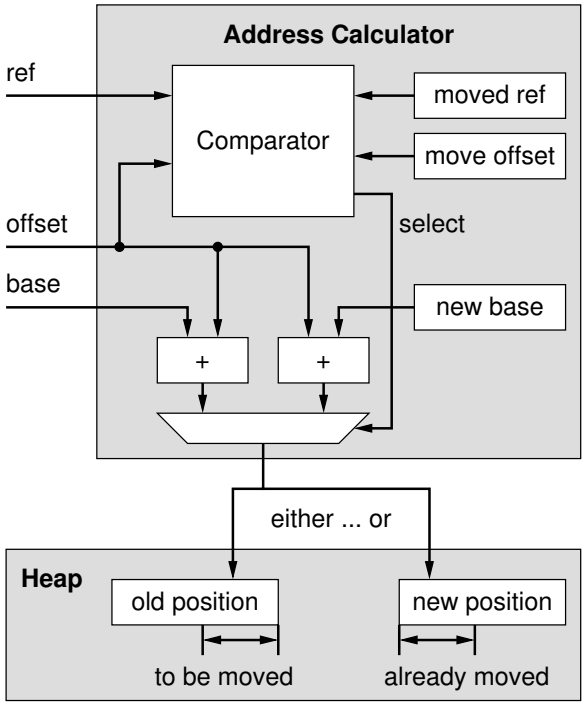

Abbildung 2: Implementierung der Speicheradressberechnung bei nebenläufigen Objektverschiebungen durch den GC.

[Reic07]). Da jedoch eine Aktualisierung aller Referenzen nach einer Objektverschiebung zu aufwändig ist, wird außerdem das Indirektskonzept eingesetzt.

Diese Variante hat jedoch den Nachteil, dass bei jedem Objektzugriff die Referenz zunächst in eine Basisadresse aufgelöst werden muss; im Folgenden als *Aktivierung einer Referenz* bezeichnet. Die zusätzlich notwendige Laufzeit für diese Aktivierung kann dabei in der realen Anwendung des Konzepts zum großen Teil „versteckt“ werden, da Referenz und Objektoffset i.Allg. getrennt vorliegen. In der SHAP-MA wird die Referenz direkt vom Stack aus aktiviert, während parallel der korrekte Offset aus Bytecode-Direktwerten bzw. Feldindizes bestimmt wird. Nach der Aktivierung einer Referenz können beliebig viele Lese- und Schreibzugriffe an beliebigen Offsets erfolgen, bis eine andere Referenz aktiviert wird. Dies wird insbesondere für Methodenaufrufe, *Object Locking* und Feldzugriffe (Überprüfung der Feldgrenzen) genutzt. Wird, z. B. bei der Verarbeitung von Feldern, mehrfach hintereinander auf das gleiche Objekt zugegriffen, so wird die erneute Auflösung automatisch durch den Speichermanager übersprungen.

Für den Einsatz in Echtzeitsystemen, bietet die Mikroarchitektur außerdem kurze Speicherzugriffszeiten unabhängig davon, ob gerade ein Objekt im Speicher verschoben wird oder nicht. Die nebenläufige Objektverschiebung wird dabei automatisch unterbrochen und an gleicher Stelle wieder fortgesetzt. Wird nun gerade auf das in der Verschiebung befindliche Objekt lesend oder schreibend zugegriffen, so wird der Objektzugriff automatisch auf die neue Position umgelenkt, sofern der Offset sich im bereits verschobenen Bereich des Objektes befindet.

Die zugehörige Adressberechnung ist schematisch in Abb. 2 dargestellt. Der *Comparator* entscheidet dabei über die Umlenkung des Speicherzugriffs. Die Basisadresse *base* am Eingang ist bereits bekannt, denn das Objekt wurde bereits aktiviert.

Zwecks Optimierung des kombinatorischen Pfades in der Hardware-Implementierung, werden wie im Bild angegeben zwei Addierer eingesetzt, die die beiden möglichen Speicheradressen parallel berechnen. Zeitgleich dazu entscheidet der *Comparator* über die zu verwendende Adresse.

3 Single-Port-Speichermanager

Eine erste, einfache Implementierung des Speichermanagers sah lediglich einen Zugriffs-Port vor. Solch ein Speichermanager-Port übersetzt die High-Level-Kommandos—Allokation, Referenzaktivierung, Lesen und Schreiben—der CPU in interne Kommandos und speichert außerdem die Basisadresse der aktuell aktivierten Referenz für Adressberechnungen zwischen.

Für eine effiziente Abarbeitung des Java-Programms, ist in die SHAP-MA ein Methoden-Cache integriert [Pre+07], der die aktuell ausgeführte Java-Methode

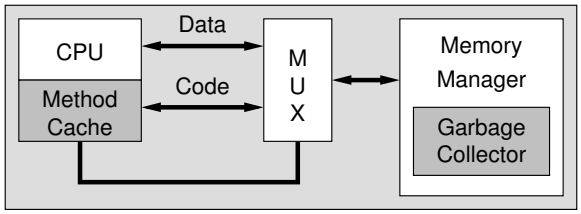

Abbildung 3: Single-Port-Speichermanager

zwischenspeichert. Dieser Cache benötigt jedoch ebenfalls einen direkten Zugriff auf den Speichermanager. Da jedoch zunächst nur ein Zugriffs-Port zur Verfügung stand, wurde eine einfache Multiplexer-Schaltung nach Abb. 3 implementiert. Die Steuerung des Multiplexers erfolgte dabei ausschließlich durch den Methoden-Cache, der den Port wieder „freigab“, sobald das Caching beendet war.

Ein wesentlicher Nachteil dieses Prinzips ist jedoch, dass CPU und Methoden-Cache nur auf demselben Objekt arbeiten können, da nur ein einziger Port vorhanden ist, indem eine Referenz aktiviert werden kann. Dies begrenzte die möglichen Caching-Strategien des Methoden-Caches auf ein einfaches Prinzip, bei dem nach Aufforderung durch die CPU die komplette Methode am Stück in den Cache zu laden war.

4 Multi-Port-Speichermanager

Für die Anbindung mehrerer Komponenten wie bspw. DMA-Kontroller, weitere CPUs oder CPU-Kerne, an den gemeinsam genutzten Heap-Speicher und damit an den selben Speichermanager, ist die Fortführung des, beim Single-Port-Speichermanager eingesetzten, Multiplexer-Prinzips jedoch nicht effizient. Zwei gravierende Nachteile wären präsent.

Die Arbitrierung zwischen den Komponenten müsste atomare Operationen aus Objektaktivierung und ggf. mehrfachem Lese- und Schreibzugriff vorsehen, da nur eine Referenz aktiviert werden kann. Dies führt jedoch zu einer ineffizienten Ausnutzung der Speicherbandbreite, da Lese- und Schreibzugriffe verschiedener Komponenten, bedingt durch die Atomizität, nicht verschachtelt werden könnten.

Zweitens wären Einsparungen bei mehrfacher Verwendung der gleichen Referenz durch eine Komponente nur noch selten möglich. Durch den Wechsel zwischen den Komponenten müssten ständig Referenzen zu Basisadressen aufgelöst werden, da jede Komponente typischerweise mit einem anderen Objekt arbeitet.

Eine alternative Lösung besteht daher darin den Speichermanager mit mehreren Ports, einem pro zugreifender Komponente, zu versehen. Jeder Port kann nun seine eigene Referenz aktivieren und halten. Die Arbitrierung erfolgt erst auf der Ebene der Low-Level-Kommandos: Ermitteln der Basisaddresse, Lesen und Schreiben. Das selektierte Kommando wird dann an die Adressberechnung übergeben, die ggf. den Speicherzugriff an die neue Adresse umlenkt. Für die Allokation von Objekten sprechen die Ports direkt die Objektverwaltung an, wobei ein separater Arbiter zum Einsatz kommt. Der schematische Aufbau hierzu ist in Abb. 4 dargestellt.

In dieser Lösung kann nun die Speicherbandbreite effektiv genutzt werden, indem der Speicher über zwei getrennte Kanäle angebunden wird: Einer übermittelt das Kommando sowie die Schreibdaten (*Request*) an den Speicher und der andere transferiert die Lesedaten (*Reply*) zurück an den Port. Zwecks Zuordnung von Lesedaten zu Kommando/Port wird ein Tag eingesetzt, wie es bspw. auch beim Bussystem „AMBA AXI“ [ARM04] der Fall ist. Somit ergibt sich ein Pipelining der *Requests* sowie eine Verschachtelung der *Requests* verschiedener Ports und damit eine effiziente Abarbeitung.

Jegliche Bandbreite die nicht von den Ports genutzt wird, steht nach wie vor automatisch dem GC für das Scannen und Verschieben von Objekten zur Verfügung.

Da die Basisadressen in den einzelnen Ports zwischengespeichert werden, müssen sie nach dem Verschieben eines Objektes ggf. angepasst werden. Dafür ist ein weiterer Kanal verantwortlich, der die betreffende Referenz und die neue Basisadresse übermittelt. Jeder Port bestimmt nun selbstständig, ob er seine zwischengespeicherte Basisadresse aktualisieren muss.

Diese Variante wurde so in der SHAP-MA implementiert und gleich zur Optimierung der Methoden-Cache-Anbindung genutzt (Abb. 5). Die Ausführung einer Java-Methode kann nun bereits dann starten, nachdem die ersten Befehle im Cache zwischengespeichert wurden. (Der genaue Zeitpunkt hängt von mehreren Faktoren ab.) Das Prinzip der getrennten Kanäle für *Request* und *Reply* wurde ebenfalls auf der Verbindung von Methoden-Cache und Speichermanager-Port eingesetzt.

Die erste reale Anwendung des Multi-Port-Speichermanagers war jedoch die Bereitstellung eines DMA-Kanals, damit die GPU (*Graphics Processing Unit*) direkten Zugriff auf den Objektspeicher erhält. Da auf den eingesetzten Entwicklungsboards kein separater SRAM für den Bildspeicher zur Verfügung steht, wurde dieser direkt in einem Objekt (Integer-Feld) abgelegt. Über einfache Feldmanipulationen mittels Java kann nun der Bildspeicher modifiziert und damit Grafikroutinen implementiert werden. Ein ähnliche Anwendung wäre das Senden und Empfangen von Ethernet-Paketen über einen Ethernet-MAC.

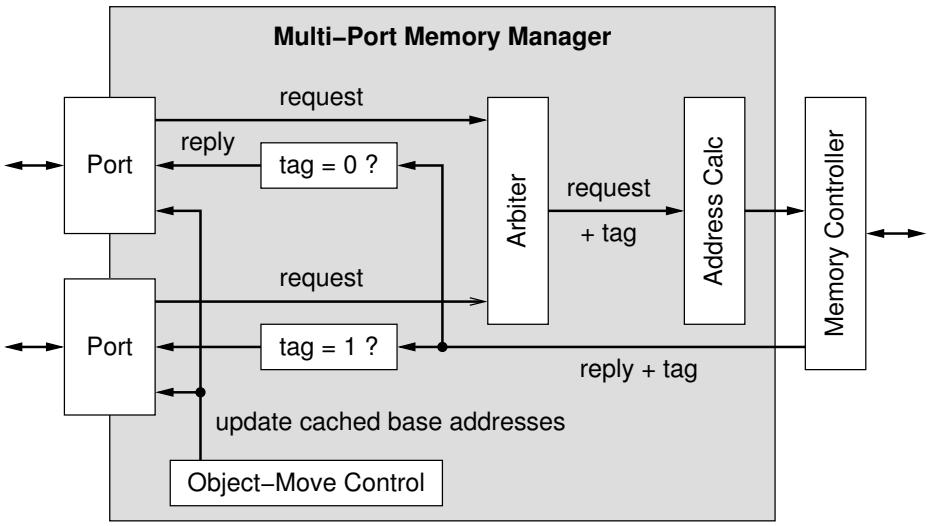

Abbildung 4: Schematischer Aufbau des Multi-Port-Speichermanagers
(Die Arbitrierung von Allokationskommandos wurde der Übersichtlichkeit halber weggelassen.)

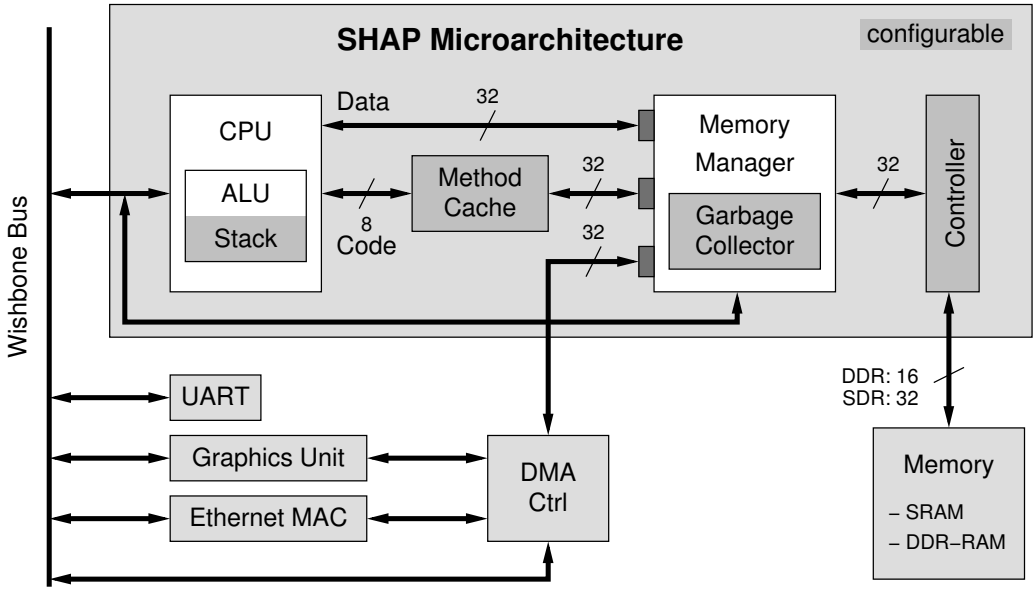

Abbildung 5: Aufbau der SHAP-Mikroarchitektur mit Multi-Port-Speichermanager

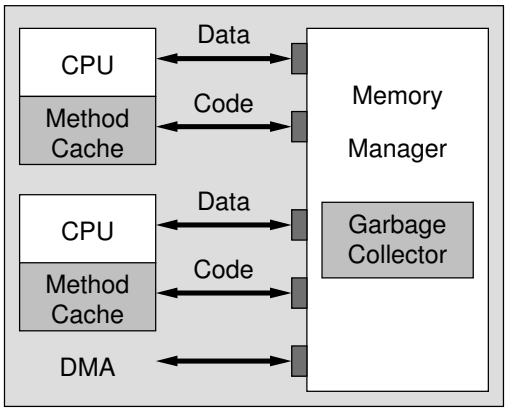

Abbildung 6: Anwendung des Multi-Port-Speichermanagers in einem Multi-CPU-System mit gemeinsamen Speicher

Die nächste Ausbaustufe der SHAP-MA sieht mehrere CPU-Kerne vor, die auf einem gemeinsam genutzten Heap arbeiten. Dazu bekommt jede CPU und jeder Methoden-Cache einen eigenen Port am Speichermanager wie in Abb. 6 dargestellt. Geplant ist außerdem ein Multi-Core-SHAP, bei dem die einzelnen Kerne zusätzlich auf einem gemeinsamen Stack arbeiten.

Die aktuell implementierte Lösung bietet noch Raum für Optimierungen. Die Verwaltungsdaten (u.a. Basisaddressen) zu den Referenzen, werden ebenfalls im Speicher in einem separatem Bereich abgelegt, da ansonsten zu viel On-Chip RAM benötigt würde. Um nun aber das Ermitteln von Basisadressen zu beschleunigen kann ein *Translation Look-aside Buffer* eingesetzt werden. Neben der Logik für eine adäquate Ersetzungsstrategie sind zusätzliche Vergleicher notwendig, um die zwischengespeicherten Basisadressen nach der Verschiebung eines Objektes zu aktualisieren oder zu invalidieren. Aktuell ist eine reduzierte Version dieses Verfahrens implementiert, bei der neben der Basisadresse für die aktuell aktivierte Referenz noch die Basisadresse der davor aktivierten Referenz zwischengespeichert wird.

5 Zusammenfassung

Es wurde ein Speichermanager mit mehreren Ports für die SHAP-Mikroarchitektur implementiert, der mehreren Komponenten gleichzeitigen und unabhängigen Zugriff auf die Objekte im Heap gewährt.

Ein wesentliches Problem dabei war das Indirektionskonzept für Objektreferenzen, bestimmt durch den in die Hardware integrierten, nebenläufigen *Garbage Collector*. Die Auflösung der Referenzen zu den Basisadressen der Objekte erfolgt jetzt für jede Komponente getrennt. Zusammen mit den getrennten Kanälen für Lese-/Schreibanfragen und Leseantworten, ergibt sich eine effiziente Ausnutzung der zur Verfügung stehenden Speicherbandbreite. Als Anwendung wurden DMA für I/O-Geräte sowie Multi-CPU/Multi-Core-Systeme vorgestellt.

Literatur

[ARM04] ARM Ltd.: *AMBA AXI Protocol, Specification*. v1.0. 3/2004

[Pre+07] PREUSSER, T. B.; ZABEL, M.; SPALLEK, R. G.: *Bump-Pointer Method Caching for Embedded Java Processors*. In: *The 5th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES 2007*, 2007

[Reic07] REICHEL, P.: *Entwurf und Implementierung verschiedener Garbage-Collector-Strategien für die Java-Plattform SHAP*, 2007

[Zab+07] ZABEL, M.; PREUSSER, T. B.; REICHEL, P.; SPALLEK, R. G.: *Secure, Real-Time and Multi-Threaded General-Purpose Embedded Java Microarchitecture*. In: *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, IEEE Press, August/2007