KESO: Konstruktiver Speicherschutz für Eingebettete Systeme

Der Technischen Fakultät der
Universität Erlangen-Nürnberg
zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von
Christian Walter Alois Wawersich

Erlangen 2009

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung: 27.10.2008
Tag der Promotion: 05.03.2009
Dekan: Prof. Dr.-Ing. habil. Huber Johannes
Erster Berichterstatter: Prof. Dr. Wolfgang Schröder-Preikschat
Zweiter Berichterstatter: Prof. Dr. Michael Philippsen

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Eingebettete Systeme finden eine immer stärkere Verbreitung in unserem Alltag. Viele Dinge, die wir im Alltag nutzen, werden von Rechnersystemen gesteuert, wie zum Beispiel das Handy, das Navigationsgerät, aber auch die Heizung, die Spülmaschine und der Wäschetrockner.

Als eingebettete Systeme werden Rechnersysteme bezeichnet, die in einem Gerät fest eingebettet sind und für eine bestimmte Aufgabe konzipiert wurden. Wie stark diese Rechnersysteme im Vergleich zum PC verbreitet sind, kann an den Stückzahlen der verkauften Mikroprozessoren abgelesen werden. Ein Studie aus dem Jahr 2002 [99] weist darauf hin, dass nur etwa 2% aller Mikroprozessoren in einem PC oder einem größeren Rechner verbaut werden.

Die Abbildung 1.1 zeigt die Aufteilung der verkauften Mikroprozessoren, aufgeschlüsselt nach verschiedenen Prozessortypen. Über 50% aller Mikroprozessoren besaßen danach im Jahr 2002 noch eine Wortbreite von 8 Bit. Interessant ist, dass die 32-Bit-Mikroprozessoren nur einen Anteil von 9% von allen verkauften Prozessoren ausmachten. Der Großteil von diesen wird wiederum in eingebetteten Systemen verbaut und nur 2% aller Prozessoren befinden sich als CPU in einem PC.

Die kleinen 8-Bit-Mikroprozessoren befinden sich in den vielen kleinen Mikrocontrollern, wie dem 8051, dem C166 und dem 6805. Mikrocontroller sind auf das Steuern von Geräten spezialisiert. Ein Mikrocontroller vereint einen Prozessorkern, einen persistenten Programmspeicher von einigen Kilobyte, einen kleinen Arbeitsspeicher von einigen wenigen 100 Byte und Steuerelektronik für Peripheriegeräte auf einem Chip. Da alle benötigten Komponenten auf einem Chip untergebracht sind, ermöglichen sie

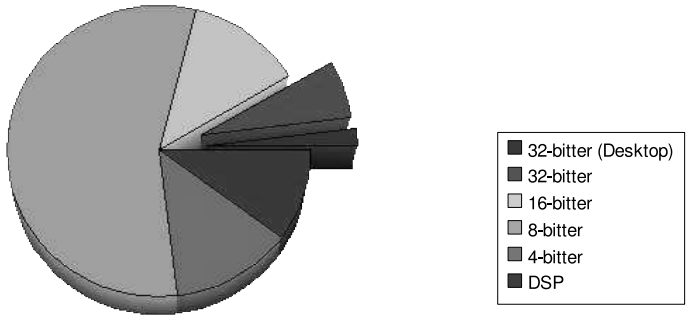

Abbildung 1.1: Marktanteil nach Stückzahl von verschiedenen Prozessorarchitekturen

ein einfaches Gerätedesign. Die kleinen Mikrocontroller verbrauchen im Vergleich zu heutigen Desktopprozessoren weniger Energie und kosten auch deutlich weniger.

Die Software der meisten Steuergeräte wird speziell für ihr Einsatzgebiet und genau für einen Mikrocontroller maßgeschneidert. Sie wird mit dem Gerät als eine Einheit ausgeliefert. Entwicklungsprozesse mit intensiven Funktionstests stellen dabei sicher, dass die Software auf genau dieser Hardware zuverlässig und korrekt funktioniert. Ein Speicherschutz wird unter diesen Umständen als nicht notwendig betrachtet und der zusätzliche Ressourcenbedarf, der für einen Speicherschutz benötigt würde, wird als zu teuer bewertet.

Der Grund für diese Bewertung liegt in der hohen Stückzahl des Endproduktes. Würde zum Beispiel aufgrund eines zusätzlichen Speicherschutzes ein teurerer Mikrocontroller benötigt, so würden zusätzliche Kosten für jedes produzierte Endprodukt entstehen. Bei großen Stückzahlen summieren sich auch kleinste Beträge zu großen Summen und die zusätzlichen Entwicklungskosten für eine speziell angepasste und aufwändig getestete Software erscheinen im Vergleich hierzu gering.

Viele der Mikrocontroller steuern und regeln technische Abläufe. Dieses sind zeitkritische Aufgaben, da die korrekte Funktionsweise vom Einhalten fester Reaktionszeiten abhängt. So muss zum Beispiel das Signal für die Zündung von einer Motorsteuerung im richtigen Augenblick erfolgen und kann nicht verschoben werden. Die in eingebetteten Systemen eingesetzten Betriebssysteme sind daher überwiegend echtzeitfähig.

Um Speicherressourcen zu sparen, kommt neben Assembler häufig C als Programmiersprache zum Einsatz. Die Programmiersprache C benötigt besonders wenige zusätzliche Systemressourcen und eignet sich sehr gut für eine hardwarenahe Softwareentwicklung. Außerdem gibt es für nahezu jeden Mikrocontroller einen passenden Übersetzer, was

auch eine Migration von Software auf andere Mikrocontroller erleichtert und dem Softwareentwickler eine einheitliche Programmiersprache bietet. Der Einsatz von C steigert die Produktivität in der Softwareentwicklung, ohne dass ein deutlicher Mehrbedarf an Speicher und damit verbundene Mehrkosten bei der Hardware entstehen.

Objektorientierte Programmiersprachen wie C++, Ada oder Java benötigen oft mehr Speicherressourcen und Rechenleistung als C und es gibt nicht für alle Mikrocontroller passende Übersetzer. Der zusätzliche Mehrbedarf an Hardwareressourcen, welcher durch den Einsatz dieser Programmiersprachen entstehen würde, führt dazu, dass auch sie aus Kostengründen gemieden werden.

Dies geschieht, obwohl der Einsatz einer objektorientierten Programmiersprache eine höhere Produktivität bei der Softwareentwicklung versprechen würde. So zeigen Studien, dass bei der Verwendung von Java als Programmiersprache die Anzahl von Programmierfehlern [78] im Vergleich zu C++ deutlich verringert und die Produktivität in der Softwareentwicklung gesteigert wurde [81].

Es gibt jedoch auch Möglichkeiten für den Einsatz von objektorientierten Sprachen im Umfeld von eingebetteten Systemen. Ein Beispiel dafür ist die SmartCard „Sm@rtCafé® Expert“ von G+D [93], die kleine Java-Programme nachladen kann, um so die Funktionalität der Karte zu erweitern. Die nachgeladenen Programme dürfen auf keinen Fall die Funktionalität der SmartCard beeinträchtigen und bestimmte Speicherbereiche der Karte weder auslesen noch verändern, da ansonsten sicherheitsrelevante Informationen missbraucht werden könnten. Ein weiteres vergleichbares Beispiel ist auch ein Java-fähiges Handy.

Die Integrität von Speicher und Systemsoftware wird durch eine typsichere Kodierung dem Java Bytecode gewährleistet. Dieser wird vor der Ausführung überprüft und anschließend von einer sicheren Laufzeitumgebung interpretiert. Bei Handys kommt für diesen Zweck eine Java Micro Edition [60] zum Einsatz und auf der Sm@rtCafé® Expert-Geldkarte eine JavaCard [95] Laufzeitumgebung.

Die genannten Anwendungsbeispiele haben gemeinsam, dass die Produkte das Nachladen von nicht vertrauenswürdigen Erweiterungen unterstützen sollen, und dass zum Schutz vor einem möglichen Missbrauch, der Bedarf für einen geeigneten Speicherschutz besteht.

Bedarf für Speicherschutz besteht jedoch nicht nur zum Schutze vor Missbrauch, auch die Steigerung von Zuverlässigkeit und Produktivität kann den Einsatz von Speicherschutz motivieren. Ein Bereich, für den zur Zeit ein Speicherschutz für diesen Zweck diskutiert wird, sind die Steuergeräte im Automobil.

Der Anteil an eingebetteten Systemen im Automobil ist in den letzten Jahren stark angestiegen und die Zulieferer der Automobilbranche erwarten, dass bis zu 90% der zukünftigen Innovationen im Elektronikbereich erfolgen werden und damit auch zu einem Großteil im Bereich der elektronischen Steuergeräte (ECU) [77].

Die Innovationen reichen von verbesserter Motorsteuerung über Fahrerassistenzsysteme bis zur intelligenten Scheibenwischerautomatik. Zur Zeit werden in einem Automobil der Oberklasse bereits mehr als 40 unterschiedliche Steuergeräte verbaut. Die meisten dieser Steuergeräte basieren auf kleinen 8-Bit-Mikrocontrollern. Je nach Aufgabengebiet werden auch leistungsfähigere 16- oder 32-Bit-Architekturen verbaut. Die Geräte sind über verschiedene Netzwerktechnologien wie CAN, MOST, LIN oder FlexRay untereinander verbunden und bilden so ein komplexes Netzwerk von Rechnern.

Die Innovationen führen dazu, dass der Bedarf an leistungsfähigeren Mikrocontrollern wächst. Auch gibt es die Bestrebung, unterschiedliche Funktionalität auf einzelnen Mikrocontrollern zusammenzufassen, um die Anzahl der Mikrocontroller zu reduzieren. Durch die Reduktion verspricht man sich eine verbesserte Auslastung der Mikrocontroller, was wiederum hilft Kosten zu sparen. Außerdem erweisen sich die Steckverbindungen zwischen den Mikrocontrollern als eine häufige Fehlerquelle, weshalb eine geringere Anzahl an ECUs einen robusteren Aufbau verspricht.

Daher ist eine Zunahme von 16-Bit- und 32-Bit-Mikrocontrollern im Automobil zu erwarten. Wie Elektroniknet [46] im Oktober 2008 verlauten lässt, rechnen die Analysten damit, dass die 32-Bit-Mikrocontroller im Automobil bis zum Jahr 2015 einen Marktanteil von 58% erreichen werden.

Die 8-Bit Mikrocontroller werden jedoch weiterhin einen bedeutenden Anteil der ECUs stellen, da auch das Automobil ein Massenprodukt ist und auch hier versucht wird, die Kosten für das Endprodukt möglichst niedrig zu halten. Neue Funktionen werden dabei häufig zuerst in der Oberklasse mit einer verhältnismäßig geringen Stückzahl von Fahrzeugen eingeführt und später bei der Migration auf Produktklassen mit größeren Stückzahlen auf kleinere und preiswerte Mikrocontroller übertragen.

Die Migration erfordert dabei eine aufwändige Anpassung der Software, um eine vergleichbare Funktionalität mit geringeren Ressourcen zu erreichen. So werden bereits umgesetzte Funktionen teilweise komplett neu programmiert, um eine vergleichbare Funktionalität auf wesentlich kleineren Mikrocontrollern umsetzen zu können.

Die Automobilindustrie steht bei der Steuergeräteentwicklung in einem ständigen Spannungsfeld zwischen dem Zwang zu neuen und innovativen Funktionen und möglichst niedrigen Produktionskosten. Niedrigere Produktionskosten können auch durch die Nutzung von Standardkomponenten erreicht werden. Diese Komponenten werden von Zulieferern für mehrere Automobilbauer und daher in größeren Stückzahlen und preisgünstiger gefertigt. Die Automobilbauer sind bestrebt, durch enge Zusammenarbeit mit den Zulieferern und der Konkurrenz die Produktion von einheitlichen Komponenten zu fördern.

Dieses Konzept wird auch auf die Software übertragen. Geht es nach dem Wunsch der Automobil-Hersteller, so soll Software in Form von Softwarekomponenten, die

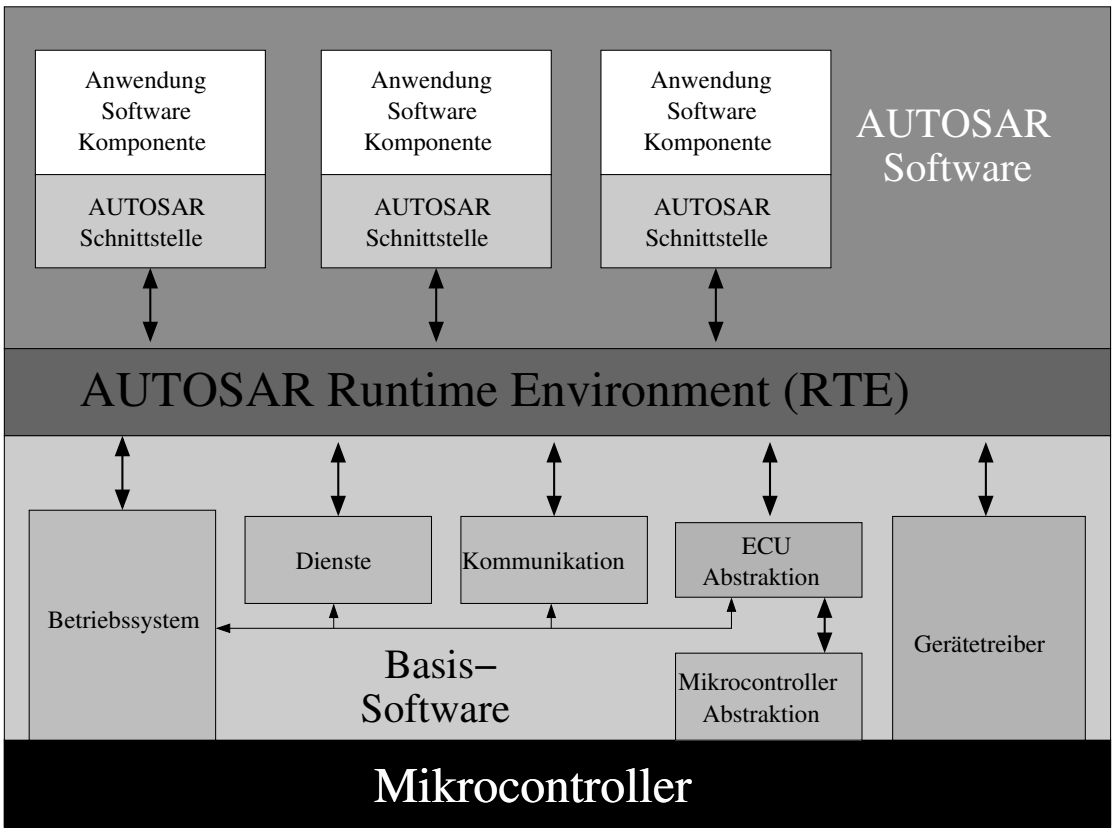

Abbildung 1.2: AUTOSAR Architekturüberblick

jeweils eine bestimmte Funktionalität bieten, von Zulieferern gefertigt werden und so dem Automobilhersteller möglichst preiswert zur Verfügung stehen. Beispiele für die von der Automobilindustrie und ihren Zulieferern betriebene Standardisierung im Softwarebereich sind OSEK/VDX [73] und AUTOSAR [14].

Während mit OSEK/VDX eine einheitliche Systemsoftware für die im Fahrzeug verbauten Steuergeräte definiert wurde, werden mit AUTOSAR die Systemsoftware weiter ausgebaut und Architektur spezifiziert. Die Anwendungsschicht sieht Softwarekomponenten vor, die über eine einheitliche Kommunikationsschicht, dem *Virtual Funktional Bus* (VFB) kommunizieren. Diese in Abbildung 1.2 dargestellte Architektur zeigt klar die Bestrebung der Automobilhersteller, die Software nach ihrer Funktion in Komponenten zu gliedern, welche auf einer einheitlichen Systemschicht aufsetzen sollen. Die Entwicklung dieser Softwarekomponenten soll durch unterschiedliche Zulieferer erfolgen.

Trotz der Verwendung von standardisierten und oft auch identischen Hardware- und Softwarekomponenten ist es für den Automobilhersteller weiterhin wichtig, sich von der

Konkurrenz abgrenzen zu können. Es muss daher für den Automobilhersteller möglich sein, den Standardkomponenten noch einen entscheidenden Mehrwert hinzuzufügen. Daher ist ein aufwändiger Integrationsprozess durch den Automobilhersteller vorgesehen, bei dem die Standardkomponenten vom Automobilhersteller noch spezifisch konfiguriert und kombiniert werden. Auf diese Weise ist es für den Hersteller möglich, ein spezifisches Image für die jeweilige Automarke aufzubauen und zu pflegen.

Wichtig für das Image ist auch die Zuverlässigkeit der Software. Eine nicht zuverlässig funktionierende Komponente kann zu teuren Rückrufaktionen führen und das Image einer Marke nachhaltig schädigen. Es ist daher nicht nur wichtig, die Kosten in der Softwareentwicklung zu senken und die Produktivität zu steigern, sondern es müssen auch zusätzliche Maßnahmen getroffen werden, welche die zu erwartenden Probleme bei der Integration von Softwarekomponenten unterschiedlicher Hersteller auf einem gemeinsam genutzten Mikrocontroller bewältigen zu können.

Mit der Integration mehrerer Softwarekomponenten von unterschiedlichen Herstellern auf einem Mikrocontroller entstehen neue Fehlerquellen. Während früher ein Hersteller für Hardware und Software eines Steuergerätes verantwortlich war, kommen nun Softwarekomponenten von unterschiedlichen Herstellern auf einem Mikrocontroller zusammen. Die physikalische Isolation zwischen den Softwarekomponenten, die durch getrennte Mikrocontroller bestand, geht hierbei verloren. Programmierfehler können daher die Funktionsweise auch von fremden Softwarekomponenten negativ beeinflussen und auch dort zu Fehlern führen. Als Folge davon kann eine Fehlfunktion nur noch schwer einem bestimmten Hersteller zugeordnet werden. Ist ein Fehler nicht reproduzierbar, so ist eine Zuordnung möglicherweise gänzlich ausgeschlossen.

In AUTOSAR ist daher ein optionaler Speicherschutz vorgesehen, welcher auf einer Erweiterung [50] für OSEK/VDX aufbaut. Dass dieser Speicherschutz optional ist, veranschaulicht erneut den bereits erwähnten hohen Druck zur minimalen Hardwareausstattung. Nur wenige der zur Zeit im Automobil eingesetzten Mikrocontroller besitzen die benötigten Voraussetzungen für einen hardwarebasierten Speicherschutz.

1.2 Zielsetzung der Arbeit

In dieser Arbeit wird ein Speicherschutzkonzept für statisch konfigurierte Systeme vorgestellt. Dieser Speicherschutz ist speziell für den Einsatz auf kleinsten eingebetteten Systemen geeignet. Der Speicherschutz verhindert wirkungsvoll fehlerhafte Speicherzugriffe, wie sie durch Programmierfehler entstehen, und somit die Ausbreitung von Fehlern über die Grenzen von Komponenten hinaus. Als Motivation dient eine auf mehrere Hersteller verteilte Softwareentwicklung und eine möglichst preiswerte, am Bedarf orientierte Hardwareausstattung.

An den Speicherschutz werden daher folgende Anforderungen gestellt:

Geringer Ressourcenbedarf: Der zusätzliche Bedarf an Hardwareressourcen für den Speicherschutz muss so gering wie möglich ausfallen, um den Einsatz auch auf möglichst preiswerter Hardware zu ermöglichen.

Hardwareunabhängigkeit: Der Speicherschutz muss auf einer Vielzahl von unterschiedlichen Architekturen realisierbar sein, da je nach Anforderung der Anwendung unterschiedliche Mikrocontroller und Prozessorarchitekturen zum Einsatz kommen.

Echtzeitfähigkeit: Die mit dem Speicherschutz verbundenen Algorithmen und Aktionen müssen ein zeitlich vorhersagbares Verhalten aufweisen, da eingebettete Systeme überwiegend Echtzeitanforderungen erfüllen müssen.

Die Wahl fiel dabei auf einen rein softwarebasierten Ansatz, da dieser plattformunabhängig auf unterschiedlichen Mikrocontrollern eingesetzt werden kann.

1.3 Aufbau der Arbeit

Im nachfolgenden Kapitel 2 sollen unterschiedliche Isolations- und Speicherschutzkonzepte und Ihre Vor- und Nachteile kurz beleuchtet werden, um die Entscheidung für eine softwarebasierten Lösung zu begründen. Im anschließenden Kapitel 3 wird die Architektur einer Java-Laufzeitumgebung beschrieben, die einen softwarebasierten Speicherschutz für eingebettete Systeme bietet. Außerdem werden einzelne Architekturentscheidungen im Hinblick auf die besonderen Randbedingungen dargestellt. Kapitel 4 und Kapitel 5 beschreiben die Datenstrukturen und den Aufbau des Java-Bytecode-zu-C-Übersetzers, welcher im Rahmen dieser Arbeit entwickelt wurde. Im Kapitel 6 werden die Kosten für den Speicherschutz an Hand von prototypischen Anwendungen diskutiert. Kapitel 7 fast die Ergebnisse der Arbeit zusammen.

Kapitel 2

Speicherschutzkonzepte

2.1 Einleitung

2.1.1 Softwarekomponenten und Schutzräume

Die Funktionalität und der Umfang der Software im Automobil nehmen immer weiter zu, und mit dem Umfang steigt auch die Komplexität der Abläufe. Eine Technik, um die Komplexität eines größeren Systems zu beherrschen, beruht darauf, das Problem in kleinere überschaubare Komponenten aufzuteilen und anschließend aus diesen Komponenten ein größeres Gesamtsystem zu entwickeln. Bei der Hardware wird dieser Ansatz bereits sehr erfolgreich umgesetzt und Softwarekomponenten [69, 97] sollen in der Softwareentwicklung eine vergleichbare Rolle übernehmen.

Wie bei der Hardware sollen auch bei der Software die Komponenten von unterschiedlichen Herstellern entwickelt werden. Anschließend werden die Komponenten zusammengeführt. Treten bei der Integration von Softwarekomponenten Fehler auf, so können sich diese auf andere Softwarekomponenten auswirken. Die Zuordnung von Fehlern zu einer bestimmten Komponente und damit zu einem konkreten Hersteller ist zeitaufwändig und teuer.

Um die Fehlerausbreitung zu beschränken, ist es daher sinnvoll, jeder Komponente einen eigenen Speicherbereich zuzuordnen. Mit Hilfe eines Speicherschutzes können diese Speicherbereiche voneinander isoliert werden, und so kann verhindert werden, dass sich Speicherfehler unbemerkt über Komponentengrenzen hinweg ausbreiten. Die durch einen Speicherschutz voneinander isolierten Speicherbereiche werden im Weiteren als Schutzräume bezeichnet.

2.1.2 Vertrauenswürdigkeit von Speicherschutz

Alle nachfolgenden Speicherschutzkonzepte beruhen zumindest zu einem Teil selbst auf Software [23]. Ein Fehler in dieser Softwarekomponente kann daher immer den Speicherschutz selbst in Frage stellen. Es ist daher nötig, dass dieser Komponente ein größeres Maß an Vertrauen entgegengebracht werden kann.

2.1.3 Konservative Softwareentwicklung

Eine empirische Studie, die auf der Analyse von mehr als 1000 Fehlern in Linux und OpenBSD [28] basiert und diese Fehler über einen längeren Zeitraum rückwirkend untersucht hat, hat gezeigt, dass Softwarekomponenten, welche länger im Einsatz sind und häufig verwendet werden, zu weniger verbleibenden Fehlern tendieren. Dieses muss unter der Voraussetzung gesehen werden, dass gefundene Fehler auch immer fortlaufend bereinigt werden und die Komponente ansonsten keiner starken Neuentwicklung unterliegt.

2.1.4 Granularität von Schutzräumen

Eine weitere Beobachtung, die von der Studie ebenfalls bestätigt wurde, ist, dass die Anzahl von Fehlern in einer Softwarekomponente mit ihrer Größe und Komplexität zunimmt. Es ist daher ratsam, die Softwarekomponente, welche den Speicherschutz implementiert, von allen anderen Softwarekomponenten zu isolieren.

Eine Strategie, um ein möglichst stabiles und vertrauenswürdiges Gesamtsystem zu bekommen, ist es, den Umfang der Software, der vertraut werden muss, möglichst gering zu halten und die Teile der Software, die einer starken fortlaufenden Veränderung unterworfen sind oder komplexe Aufgaben erledigen, auf getrennte Schutzräume aufzuteilen.

Die Abbildung 2.1 veranschaulicht schematisch die Unterteilung der auf einem System laufenden Software. Die Basissoftware stellt die üblichen Dienste eines Betriebssystems zur Verfügung, wie zum Beispiel die Auftragsplanung, die Ressourcenverwaltung und die Gerätetreiber. Die Anwendungssoftware übernimmt die eigentlichen anwendungsspezifischen Aufgaben.

Monolithische Betriebssysteme kennen keine Untergliederung der Basissoftware und vertrauen daher allen Basiskomponenten. Um noch robustere Systeme zu schaffen, wurde die Mikrokernarchitektur entwickelt, bei der auch Teile der Basissoftware vom Speicherschutz profitieren. Bekannte Beispiele für Systeme mit einem hardwarebasierten

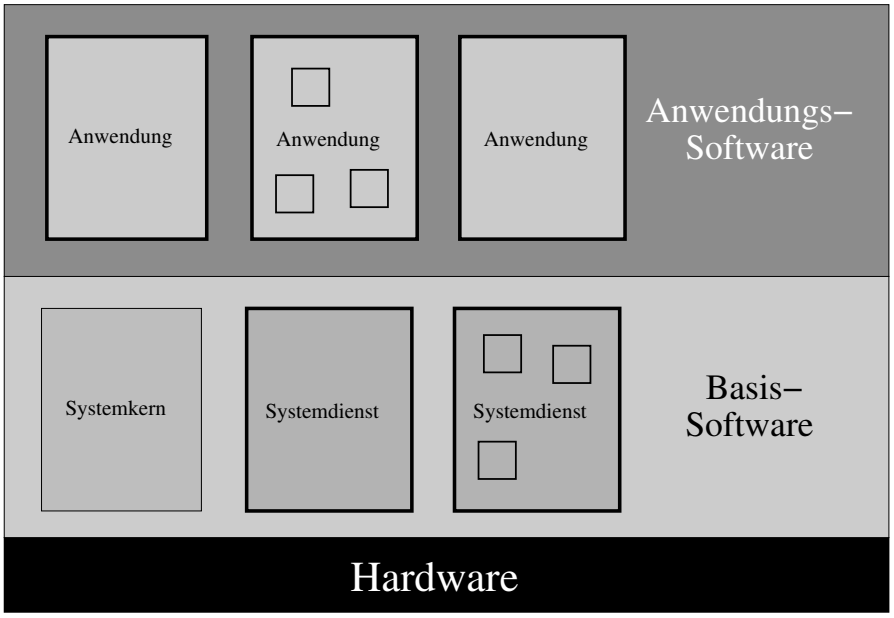

Abbildung 2.1: Granularität von Schutzarchitekturen.

Speicherschutz und Mikrokernarchitektur sind MACH [4], KeyKOS [47], EROS [89] und L4 [65].

Der größte Kritikpunkt an Mikrokernarchitekturen mit hardwarebasiertem Speicherschutz sind die zusätzlichen Kosten für die Kommunikation zwischen den Komponenten der Basissoftware. So ist ein auf einem MACH-3 basierendes Unix-Betriebssystem um 50% langsamer als ein monolithisches Unix [29, 48].

2.1.5 Kommunikation zwischen Schutzräumen

Die auf die Schutzräume aufgeteilten Softwarekomponenten müssen sinnvollerweise auch mit der Umwelt und anderen Softwarekomponenten kommunizieren. Für die Kommunikation ist es nötig, dass ein Austausch von Daten zwischen den Schutzräumen stattfindet.

Fehlerhafte Daten oder fehlerhaftes Verhalten bei der Kommunikation kann auch zur Ausbreitung von Fehlern führen. Ein Beispiel für derartige Fehler sind Pufferüberläufe, welche auch in Systemen mit Speicherschutz häufig zur Fehlerausbreitung führen können und die häufigste Technik sind, um den Speicherschutz auch böswillig und gezielt zu umgehen [31, 79].

2.1.6 Auswahlkriterien

Im Weiteren sollen unterschiedliche Speicherschutzkonzepte betrachtet werden, die für die Aufgabe in Betracht kommen. Ein geeigneter Speicherschutz sollte die in der Einleitung genannten Anforderungen erfüllen — geringer Ressourcenbedarf, Hardware unabhängig und echtzeitfähig. Weitere Eigenschaften, die berücksichtigt werden sollen, sind die Kommunikation zwischen Schutzräumen und die Möglichkeit eine Softwarekomponente auf möglichst einfache Art auf einem Mikrocontroller mehrfach verwenden zu können.

Die Kommunikation zwischen Schutzräumen dient dazu, die Hardware ansprechen zu können und Daten mit anderen Schutzräumen auszutauschen. In dem Anwendungsfall ist davon auszugehen, dass die Softwarekomponenten relativ klein ausfallen und ein häufiger Datenaustausch stattfindet. Daher sollte die Kommunikation ohne großen Mehraufwand für das Wechseln von Schutzräumen auskommen.

Um die Wiederverwendbarkeit von Softwarekomponenten zu steigern ist es sinnvoll, wenn eine Komponente mehrfach auf einem Mikrocontroller verwendet werden kann. Einige Speicherschutzkonzepte unterstützen dies, z.B. virtuelle Adressräume, die es ermöglichen, dass jeder Anwendung ein eigener Adressraum zur Verfügung steht.

2.2 Hardwarebasierter Speicherschutz

2.2.1 Das einfache Grundprinzip

Bei hardwarebasiertem Speicherschutz werden die Speicherzugriffe durch die Hardware überwacht. Dieser Ansatz funktioniert auch, wenn die nicht vertrauenswürdige Softwarekomponente direkt in einer Maschinensprache programmiert wurde. Die überwachten Speicherzugriffe umfassen nicht nur die Lade- und Speicherbefehle, sondern oft auch das Laden der Maschinenbefehle vor der Ausführung. Die Hardware kann daher in den meisten Fällen zwischen den drei Zugriffsrechten **Lesen**, **Schreiben** und **Ausführen** unterscheiden.

Schlägt die Überprüfung beim Speicherzugriff fehl, so unterbricht die Hardware den Ablauf des aktuellen Maschinenprogramms und wechselt zu einer Unterbrechungsbehandlung in die Systemsoftware. Die Systemsoftware entscheidet anschließend über das weitere Vorgehen.

Damit die Hardware weiß, welche Rechte für welche Speicheradresse gelten, verfügt sie über zusätzliche Systemregister. Diese Systemregister beschreiben entweder direkt oder indirekt über eine komplexere Datenstruktur den Speicherbereich und die geltenden Rechte. Die Anzahl der Systemregister ist dabei immer gering und reicht in der

Regel nicht zur Beschreibung aller benötigten Schutzräume aus, so dass bei der Existenz von mehreren Schutzräumen die Systemregister durch die Systemsoftware umkonfiguriert werden müssen und die Systemsoftware eine separate Verwaltungstruktur für die Zugriffsrechte benötigt.

Damit die Systemregister und die Systemdaten nicht von der Anwendung verändert werden können, ist der Zugriff auf die Register und die Daten vor dem Zugriff der im Schutzraum laufenden Anwendung geschützt. Bei einer Unterbrechung oder bei einem Wechsel zur Systemsoftware wird dieser Schutz implizit ausgeschaltet, so dass die Daten anschließend durch die Systemsoftware verändert werden können. Hierzu unterscheidet die Hardware zwischen mindestens zwei Betriebsmodi, einem eingeschränkten, unprivilegierten Modus und einem privilegierten Modus. Bei einer Unterbrechung oder beim Aufruf einer Systemroutine findet ein Moduswechsel in den privilegierten Modus statt und beim Rücksprung wird wieder in den unprivilegierten Modus geschaltet. Im unprivilegierten Modus werden neben den Lade-/Speicherbefehlen noch die Ausführung von weiteren Maschinenbefehlen unterbunden, wie zum Beispiel die Befehle zum Verändern der Systemregister.

Eine für hardwarebasierten Speicherschutz geeignete CPU benötigt daher eine Hardware, die mindestens folgende Funktionen bietet:

- Die Hardware besitzt mindestens zwei Betriebsmodi, einen privilegierten und einen unprivilegierten Betriebsmodus, und nur im privilegierten Modus kann die Konfiguration für den Speicherschutz verändert werden.
- Es gibt die Möglichkeit, für einen oder mehrere Speicherbereiche Zugriffsrechte festzulegen, welche im unprivilegierten Modus den Zugriff auf die Speicherbereiche einschränken.
- Alle Speicherzugriffe werden im unprivilegierten Modus überprüft und bei einer Zugriffsverletzung wird der Programmablauf unterbrochen und an einer vordefinierten Stelle im privilegierten Modus fortgesetzt.

Die dargestellten Voraussetzungen für hardwarebasierten Speicherschutz wurden bewusst möglichst allgemein gehalten, da der Speicherschutz in existierenden Prozessoren sehr unterschiedlich umgesetzt wird. Die genannten Voraussetzungen sollten jedoch für einen Speicherschutz ausreichen. Je nach zusätzlicher Funktionalität ist der hardwarebasierte Speicherschutz auf Speicherseiten, Segmente oder Speicherbereiche ausgerichtet.

2.2.2 Seitenbasierter Speicherschutz

In den größeren 32-Bit-Prozessorarchitekturen ist der Speicherschutz mit der Adressumrechnung verbunden. Die zuständige Hardwareeinheit auf der CPU wird als *Memory*

Managment Unit (MMU) bezeichnet, da sie über den reinen Speicherschutz hinaus auch Funktionalität für die Speicherverwaltung bietet.

Bei der seitenorientierten Speicherverwaltung wird der physikalische Speicher in Kacheln gleicher Größe unterteilt. Für jede Seite wird ein Eintrag in einer Seitenkacheltabelle verwaltet, in dem die Zugriffsrechte und weitere Daten für die Adressumrechnung gespeichert sind. Die Zugriffsrechte gelten für den von der Seite abgedeckten Speicherbereich.

Eine lineare Seitenkacheltabelle, die einen 32-Bit-Adressraum vollständig abdeckt und eine Seitengröße von 4096 Zeichen besitzt, benötigt 1048576 Einträge.[1] Üblicherweise besitzt ein Eintrag in der Seitenkacheltabelle die gleiche Größe wie die Speicheradresse. Im Falle einer 32-Bit-Rechnerarchitektur sind es 32 Bit oder 4 Byte. Der Speicherbedarf für eine vollständige lineare Seitenkacheltabelle liegt daher bei 4 Megabyte. Zur Beschreibung von einem Schutzraum wird jeweils eine Seitenkacheltabelle benötigt.

Der volle Adressraum wird jedoch nur sehr selten ausgenutzt, daher ist es sinnvoll, nur die genutzten Einträge in einer Datenstruktur zu speichern. Anstelle einer linearen Seitenkacheltabelle werden daher oft andere Datenstrukturen verwendet, wie z.B. eine hierarchische Seitenkacheltabelle[2] oder eine Hashtabelle[3]. All diese Datenstrukturen finden in der MMU selbst keinen Platz. Sie werden daher im Arbeitsspeicher abgelegt oder, wenn vorhanden, sogar auf einem Hintergrundspeicher.

Ein Speicherzugriff führt im Idealfall einer linearen Seitenkacheltabelle, zu mindestens zwei Speicherzugriffen, einem zum Lesen des Eintrags aus der Seitenkacheltabelle und anschließend dem eigentlichen Speicherzugriff. Bei den anderen genannten Verfahren ist der Aufwand entsprechend größer. Um einen schnellen Speicherzugriff zu ermöglichen speichert die MMU daher ausgewählte Einträge nach dem ersten Zugriff in einem schnellen Zwischenspeicher, dem *Translation Lookaside Buffer* (TLB) [4] . Der TLB ist als voll assoziativer Speicher implementiert, die MMU kann daher einen Eintrag im TLB in konstanter Zeit finden. Die zusätzliche Zeit, die für das Nachschlagen im TLB benötigt wird, kann durch zusätzliche Stufen in der Prozessor-Pipeline abgearbeitet werden. So wird im idealen Fall der Durchsatz an ausgeführten Maschinenbefehlen nicht durch die MMU begrenzt.

Wenn eine seitenorientierte Speicherverwaltung nicht aus einem anderen Grund benötigt wird, so ist ein seitenorientierter Speicherschutz für die Anforderungen von eingebetteten Systemen ungeeignet. Die Speicherressourcen für die benötigte Seitenkacheltabelle steht

[1] Die Anzahl der Tabelleneinträge berechnet sich aus der Anzahl der Adressen (2^{32}) geteilt durch die Seitengröße (2^{12}). $\frac{2^{32}}{2^{12}} = 2^{20} = 1048576$ Einträge.

[2] Die x86-Architektur von Intel besitzt ab dem Modell 80286 eine dreistufige Seitenkacheltabelle.

[3] Hashtabellen werden überwiegend im Zusammenhang mit inversen Seitenkacheltabellen verwendet.

[4] Die Anzahl der Einträge im TLB ist in der Regel auf einige wenige bis ca. tausend Einträge begrenzt (4–128 TriCore, 1024 Opteron). Sind alle Einträge belegt und wird ein neuer Eintrag benötigt, so wird ein alter Eintrag mit Hilfe einer einfachen Verdrängungsstrategie ausgewählt und verdrängt.

in keinem sinnvollen Verhältnis zum Nutzen. Die feste Seitengröße führt bei großen Seiten zu einem zusätzlichen großen Speicherverschnitt und bei einer kleinen Seitengröße zu noch größeren Seitenkacheltabellen. Der zusätzliche Aufwand zum Lesen der Einträge führt im schlechtesten Fall zu langen Ausführungszeiten von Lade-/Speicherbefehlen und als Folge zu nicht mehr sinnvollen Aussagen über die Ausführungszeit von Maschinenprogrammen. Daher ist der TLB keine akzeptable Alternative für eine Anwendung mit harten Echtzeitanforderungen. Des weiteren ist der TLB als voll-assoziativer Speicher für den Einsatz in preiswerten Mikrocontrollern zu teuer.

2.2.3 Segmentbasierte Speicherschutz

Neben einer Speicherverwaltung mit gleich großen Seiten, gibt es Architekturen, die den Speicher in Segmente aufteilen und diese Segmente zur Adressumrechnung nutzen. Die Hardwareeinheit wird auch hier als *Memory Managment Unit* (MMU) bezeichnet. Die Adressen beim Speicherzugriff können als Index innerhalb des Segmentes betrachtet verwenden. Das Segment, auf das sich die Adresse bezieht, wird beim Speicherzugriff je nach Architektur und Befehl implizit oder explizit mit angegeben.

Segmente besitzen im Gegensatz zu Speicherseiten eine konfigurierbare Basisadresse und Segmentlänge. Beim Zugriff wird überprüft, ob die Adresse die Länge des Segmentes nicht überschreitet und anschließend wird die Basisadresse zur Adresse addiert und so die physikalische Adresse bestimmt.

Die Anzahl der zur Verfügung stehenden Segmente ist je nach Architektur auf 2–8 Segmente begrenzt. Die MMU benötigt für jedes Segment einen Segmentdeskriptor bestehend aus der Basisadresse, der Segmentlänge und den Zugriffsrechten. Die Segmentdeskriptoren liegen je nach Architektur im Arbeitsspeicher oder direkt in der MMU. Finden die Segmentdeskriptoren platz in der MMU, so wird kein TLB benötigt. Eine segmentbasierte Speicherverwaltung ist für den Einsatz in eingebetteten System interessant, da keine großen Datenstrukturen im Arbeitsspeicher verwaltetet werden müssen und im einfachen Fall auch kein TLB benötigt wird.

Die nicht lineare Adressierung des Arbeitsspeichers hat Vor- und Nachteile. So ermöglicht eine segmentbasierte Speicherverwaltung die Adressierung von einem Speicher mit mehr als 2^{16} Adressen mit Hilfe einer 16 Bit Adresse, da die physikalische Adressen noch um die Basisadresse aus dem Segementdeskriptor erweitert wird. Die nicht lineare Adressierung von Speicher führt jedoch auch zu zusätzlichen Aufwand bei Sprüngen im Maschinenprogramm und beim Kopieren von Daten, wenn dieses über Segmentgrenzen hinaus geschieht.

2.2.4 Speicherschutz mit Begrenzungsregistern

Soll nur ein einfacher Speicherschutz unterstützt werden, so reicht eine Hardware aus, welche Begrenzungsregister bereitstellt. Die entsprechende Hardwareeinheit wird als *Memory Protection Unit* (MPU) bezeichnet, da sie keine weiteren Speicherverwaltungsaufgaben übernimmt. Eine MPU beschreibt die Zugriffsrechte für einen Speicherbereich mit Hilfe von Bereichsdeskriptoren oder Begrenzungsregistern, welche einen Speicherbereich und dessen Zugriffsrechte beschreiben. Ein Bereichsdeskriptor beinhaltet für diesen Zweck jeweils die erste und letzte gültige Adresse eines Speicherbereiches und die für diesen Bereich geltenden Zugriffsrechte.

Der in dieser Arbeit verwendete TriCore Mikrocontroller TC1796 besitzt eine MPU mit Begrenzungsregistern, welche in dieser Architektur als *Memory Protection Register* (MPR) bezeichnet werden. Die Anzahl dieser Systemregister ist wie üblich begrenzt. Der TC1796 besitzt insgesamt 12 Register, wobei diese auf zwei Konfigurationen aufgeteilt sind. Daher sind immer nur 6 Register aktiv. Die Systemsoftware kann zwischen diesen Konfigurationen umschalten, und bei mehr als zwei Schutzräumen muss die Konfiguration jeweils entsprechend verändert werden.

Begrenzungregister sind eine schlanke Alternative zu einer aufwändigen Speicherverwaltung. Sie sind ausreichend, um den gewünschten Speicherschutz zwischen Softwarekomponenten zu realisieren. Die zusätzlichen Arbeitsschritte für die Überprüfung der Speicherbereiche wird wie bei den anderen Verfahren durch einen zusätzlichen Schritt in der Prozessor-Pipeline abgearbeitet, so dass auch hier der Speicherschutz den Ablauf vom Maschinenprogramm nur in selten Fällen verlangsamt.

2.3 Der konstruktive Speicherschutz

2.3.1 Grundprinzip

Ein Speicherschutz, der auf jedem Mikrocontroller implementiert werden kann, ist der softwarebasierte Speicherschutz oder konstruktive Speicherschutz. Beim softwarebasierten Speicherschutz werden die Speicherzugriffe nicht implizit durch die Hardware überprüft, sondern die Überprüfung findet vor der Ausführung der Speicherzugriffe explizit durch die Software statt.

Eine Überprüfung des Speicherzugriffes kann erfolgen, sobald alle benötigten Daten für die Überprüfung bekannt sind. Dies ermöglicht unterschiedliche Ansatzpunkte für einen konstruktiven Speicherschutz. Zum einen ist es möglich, dass eine Überprüfung erst auf Maschinenprogrammebene, kurz vor dem Speicherzugriff in Form eines kurzen

Maschinenprogramms erfolgen muss, zum anderen können Überprüfungen auch schon zum Zeitpunkt der Maschinenprogrammerzeugung erfolgen.

Das Vorabwissen für die Überprüfung hängt dabei stark davon ab, ob als Ausgangspunkt ein Maschinenprogramm oder eine höhere Programmiersprache dient.

2.3.2 Konstruktiver Speicherschutz auf Maschinenprogrammebene

Bei einem softwarebasierten Speicherschutz auf Maschinenprogrammebene werden die Überprüfungen direkt vor der Ausführung des entsprechenden Maschinenbefehls vorgenommen. Es gibt dabei zwei unterschiedliche Ansätze. Eine Möglichkeit besteht darin, die Maschinenbefehle nicht direkt durch die Hardware ausführen zu lassen, sondern einen Emulator zu nutzen. Die Interpretation der Maschinenbefehle führt jedoch zu einem großen zusätzlichen Mehraufwand und die Maschinenbefehle müssen bei einer Harvard-Architektur im Datenspeicher liegen. Daher scheidet die Emulation als Speicherschutztechnik für das in dieser Arbeit angestrebte Einsatzgebiet aus.

Alternativ zur Emulation kann die Überprüfung durch eine Instrumentalisierung des Maschinenprogramms erfolgen. Für die Überprüfung werden zusätzliche Maschinenbefehle direkt vor dem Speicherzugriff eingefügt. Die Instrumentalisierung kann beim Laden der Software auf den Mikrocontroller geschehen.

Auch hierbei kommt es noch zu einem deutlichen Mehraufwand. Für eine sinnvolle Überprüfung reicht im allgemeinen Fall eine Bereichsüberprüfung aus. Eine solche Überprüfung umfasst auf den meisten Architekturen zwei zusätzliche Maschinenbefehle[5] pro Speicherzugriff. Rechnet man für den eigentlichen Speicherzugriff nur einen Maschinenbefehl, so führt eine Überprüfung zu einem dreimal höheren Aufwand.

Diese Abschätzung ist jedoch recht pessimistisch. Viele der Überprüfungen können entfallen, wenn klar ist, dass das Maschinenprogramm die Bedingungen nicht verletzen kann. Dies gilt im Besonderen für das Laden der Maschinenbefehle. Nur bei indirekten Sprüngen ist hier eine Überprüfung notwendig, da ansonsten die Adressen bereits beim Instrumentalisieren des Maschinenprogramms überprüft werden können. Der tatsächliche Mehraufwand hängt auch davon ab, wie viele Speicherzugriffe in der Anwendung vorkommen. Ein aktuelles Forschungsprojekt, welches einen softwarebasierten Speicherschutz auf diese Weise umgesetzt hat, kommt bei der Programmgröße auf einen Zuwachs von 30%–65% und beim Datenspeicher zu einem Mehrbedarf von 5,8%–9,5% [59].

[5] Die zwei zusätzlichen Maschinenbefehle sind eine Vergleichsoperation und ein bedingter Sprung. Das Laden der zu vergleichenden Argumente wird in diesem Fall nicht mitgezählt, da ein Wert konstant sein kann und der andere Wert die beim Speicherzugriff benötigte Adresse darstellt.

Auch wenn die Zahlen im Vergleich zur Emulation schon recht vielversprechend erscheinen, so gibt es dennoch weitere Nachteile. Die Instrumentalisierung der Maschinenprogramme muss für jede Prozessorarchitektur separat entwickelt werden, da jede Architektur unterschiedliche Maschinenbefehle und unterschiedliche Adressierungsarten kennt. Darüber hinaus ist es nicht immer möglich, dass Maschinenbefehle und Daten zuverlässig unterschieden werden können. Das ist insbesondere der Fall bei indirekten Sprüngen, welche möglicherweise noch von Eingabedaten abhängen können, die Analyse und die Anpassung unmöglich machen [92].

2.3.3 Konstruktiver Speicherschutz auf programmiersprachlicher Ebene

Der konstruktive Speicherschutz kann auch auf der programmiersprachlichen Ebene angesetzt werden. Hierbei ergibt sich der große Vorteil, dass mehr Wissen über das Datenmodell und den Programmfluss besteht. Besonders geeignet sind dafür typsichere Programmiersprachen, da diese eine zuverlässige Aussage über die Datenstrukturen zulassen.

Sind die Struktur der Daten und deren Grenzen bereits zum Übersetzungszeitpunkt bekannt, so kann die Überprüfung des Speicherzugriffs bereits zum Übersetzungszeitpunkt erfolgen und es gibt keinen zusätzlichen Aufwand zur Laufzeit. Etwas Ähnliches gilt auch für die Ausführung von Programmen. Wenn es möglich ist, vor der Laufzeit zu verifizieren, welche Programmadressen angesprungen werden, so kann auch diese Überprüfung zur Laufzeit entfallen.

Ist aufgrund von dynamischen Datenstrukturen und dynamischen Programmteilen eine Überprüfung vor der Laufzeit nicht möglich, so muss der Übersetzer entsprechende Überprüfungen in das Maschinenprogramm einfügen. Je weniger Überprüfungen zur Laufzeit noch erfolgen, desto geringer wird der benötigte zusätzliche Aufwand für den Speicherschutz.

Konstruktiver Speicherschutz auf programmiersprachlicher Ebene ermöglichen es, die Anzahl der benötigten Überprüfungen stärker zu reduzieren als es beim Instrumentalisieren von Maschinenprogrammen möglich ist. Auch ist dieser Ansatz weniger stark von der Zielarchitektur abhängig. Als Nachteile werden jedoch die Bindung an eine Programmiersprache und die Bereitstellung der Software als quellenoffene Komponente gesehen.

Diese Nachteile können durch eine Zwischensprache behoben werden. Im Übersetzerbau wurde bereits Anfang der 60er Jahre das Konzept einer universalen Zwischensprache ersonnen, die die Wiederverwendbarkeit von Übersetzern für unterschiedliche Rechnerarchitekturen verbessern sollte [30, 55].

Anstelle für N Programmiersprachen und M unterschiedliche Rechnerarchitekturen jeweils einen, daher NxM unterschiedliche, Übersetzer zu entwickeln, benötigt man bei der Verwendung einer universalen Zwischensprache nur N Übersetzer für die Abbildung aus der Programmiersprachen in die Zwischensprache und weitere M für die Abbildung aus der Zwischensprache in die entsprechende Maschinensprache, daher $N+M$ viele Übersetzer.

Es gibt zwar bis heute keine allgemein verwendete universale Zwischensprache für alle Programmiersprachen und alle Rechnerarchitekturen, aber es gibt einige Übersetzer, die die Idee einer architekturunabhängigen Zwischensprache umsetzen. Beispiele für entsprechende Zwischensprachen sind der *P-Code* [71], *EM-1* vom *Amsterdam Compiler Kit* (ACK) [98], ANDF [68], der *Java-Bytecode* [66], die *LLVM* [62] und die *Common Intermediate Language* (CIL) [1].

Die Zwischensprachen sind dazu bestimmt, durch einen Interpreter oder einen Übersetzer weiterverarbeitet zu werden. Sie lassen sich einfacher verarbeiten als die meisten Maschinenbefehlssätze, welche auf die Belange der Hardware ausgerichtet und aus Sicht der Übersetzer oder Interpreter unnötig kompliziert sind. Insbesondere die Befehlssätze von typsicheren Zwischensprachen eignen sich als Grundlage für einen konstruktiven Speicherschutz. Ein bekanntes Beispiel hierfür ist der Java-Bytecode, welcher seine erste stärkere Verbreitung als Zwischensprache zum Erweitern von Internet-Browsern fand.

Bereits 1973 wurde ein typsicherer Befehlssatz mit der Programmiersprache Mesa auf dem Alto-System als Schutzmechanismus eingesetzt. Das Alto System unterstützte vier Programmiersprachen: BCPL, Mesa, Smalltalk und Lisp. Jede Sprache hatte ihren eigenen Mikrobefehlssatz. Der Mikrobefehlssatz von Mesa wurde ebenfalls Mesa genannt [53, 61]. Pilot als ein Nachfolger des Alto Systems benutzte ausschließlich den Mesa-Mikrobefehlssatz, um so den Systemschutz sicherzustellen [82]. Weitere Beispiele für Systeme, die eine typsichere Sprache für den Speicherschutz nutzen, sind SPIN [22], Oberon [103], Inferno [36], JX [42], KaffeOS [15], JikesNODE [85], und Singularity [7].

2.4 Speicherschutz auf Steuergeräten im Automobil

Mit der Einführung von AUTOSAR wurde auch ein Speicherschutzkonzept für Steuergeräte im Automobil spezifiziert. Vorgesehen ist dabei ein hardwarebasierter Speicherschutz basierend auf Begrenzungsregistern, wie er in Kapitel 2.2.4 beschrieben wurde. Auf eine Adressumrechnung, die die Wiederverwendbarkeit von Komponenten erleichtert, wurde verzichtet. Es sollen durch die Hardware mindestens zwei Schutzräume unterstützt werden, einer für privilegierte Funktionen und einer für die Anwendungssoftware.

Viele aktuelle Mikrocontroller unterstützen keinen hardwarebasierten Speicherschutz. Daher ist der in AUTOSAR spezifizierte Speicherschutz als optionale Funktionalität vorgesehen. Die Spezifikation beschreibt nicht, wie die Umsetzung in Hardware zu erfolgen hat.

In dieser Arbeit soll ein anderer Ansatz verfolgt werden. Ausgehend von den Erfahrungen, die mit dem Java-Betriebssystem JX [42] gemacht wurden, soll eine Erweiterung für OSEK/VDX entstehen, die einen hardwareunabhängigen Speicherschutz bietet. Wie bei JX soll der Speicherschutz auf der Typsicherheit des Java-Bytecodes basieren.

Mit AJACS [8] gibt es bereits einen vergleichbaren Versuch, um eine JVM auf einem OSEK/VDX-Betriebssystem umzusetzen. Als Einsatzgebiet wurden überwiegend kleine 16- bzw. 32-Bit-Mikrocontroller im Infotainmentbereich genannt, mit einem Programmspeicher von mehr als 64 Kbytes.

Im Gegensatz dazu sollen in dieser Arbeit mehrere JVM-Instanzen eingesetzt werden und auch 8-Bit-Mikrocontroller mit weniger als 16 KByte Programmspeicher als mögliche Zielplattform dienen.

2.5 Entscheidung für eine Multi-JVM

Ausgehend von den am Anfang genannten Anforderungen hat ein konstruktiver Speicherschutz bei der Hardwareunabhängigkeit einen klaren Vorteil. Ein softwarebasierter Speicherschutz ist von der Hardware unabhängig, hardwarebasierter Speicherschutz ist es nicht.

Die zwei weiteren Anforderungen — geringer Ressourcenbedarf und Echtzeitfähigkeit — sind nicht so klar zu beantworten. Eine kleine JVM für eingebettete Systeme, wie z.B. die KVM [60] von Sun, benötigt ohne Anwendungssoftware bereits 40–80 KByte. Die Java-Klassen werden bei der KVM dynamisch geladen und benötigen dafür RAM. RAM ist jedoch im Vergleich zu Programmspeicher auf den Mikrocontrollern besonders knapp bemessen. Der in dieser Arbeit verwendete ATmega8535 besitzt einen Datenspeicher von nur 512 Byte.

Die automatische Speicherbereinigung von Java wird für den Einsatz in Systemen mit Echtzeitanforderungen als problematisch betrachtet [90]. Echtzeitfähige automatische Speicherbereinigungen [12, 17, 21, 27, 54, 84, 91] und die RTSJ [56] bieten hier jedoch mögliche Lösungsansätze.

Auch der Mehraufwand für einen hardwarebasierten Speicherschutz wird auf den Steuergeräten oft nicht akzeptiert. Die AUTOSAR-Spezifikation sieht daher nur einen Speicherschutz basierend auf Bereichsregistern vor. Aufwändigere Techniken, die mit einer Adressumrechnung kombiniert sind, werden nicht angestrebt.

Durch die Entscheidung zu einem konstruktiven Speicherschutz ergeben sich daher zwei große Herausforderungen. Zum einen ist es nötig eine Java-Laufzeitumgebung zu bauen, die es ermöglicht eine Java-Anwendung mit einem zu einer C-Anwendung vergleichbaren Speicherbedarf, laufen zu lassen. Außerdem muss diese JVM eine echtzeitfähige Speicherverwaltung besitzen.

Im nachfolgenden Kapitel wird die Architektur dieser Laufzeitumgebung beschrieben.

Kapitel 3

KESO – Die Systemarchitektur

3.1 Einleitung und Überblick

Im folgenden Kapitel wird die Architektur einer Laufzeitumgebung vorgestellt, die es ermöglicht, mehrere Java-Anwendungen auf einem Mikrocontroller koexistieren zu lassen, wobei für jede dieser Anwendungen eine eigene *Java Virtual Maschine* (JVM) bereit gestellt wird. Die Laufzeitumgebung wird im Weiteren als KESO bezeichnet, was ein Akronym für *konstruktiver eingebetteter Speicherschutz für OSEK* ist.

Die Abbildung 3.1 zeigt eine schematische Darstellung einer möglichen KESO-Konfiguration. Als Basis dient ein OSEK/VDX-konformes Betriebssystem. Auf diesem setzt die KESO-Laufzeitumgebung auf, welche die Systemdienste des darunter liegenden OSEK/VDX-Systems um weitere Dienste und Funktionen ergänzt und diese für die Anwendungen bereitstellt.

Die Schutzräume werden in KESO als Domäne bezeichnet. Jede Domäne stellt aus der Sicht der Anwendungen eine eigenständige JVM dar. Sie besitzt ihren eigenen Speicherbereich. Dieser kann nicht von einer anderen Domäne aus direkt verändert werden. Die Aktivitätsträger sind fest einer Domäne zugeordnet. Andere Dienste von OSEK/VDX wie Alarm, Events, Counter und Ressourcen können sowohl fest einer Domäne zugeordnet werden als auch global zur Verfügung stehen.

Die Konfiguration einer KESO-Laufzeitumgebung ist statisch. Diese statische Konfiguration orientiert sich an dem Generierungsprozess des zugrunde liegenden OSEK/VDX-Systems, welches ebenfalls eine statische Konfiguration mit einer eigenen Konfigurationssprache [74] vorsieht.

Die Domäne kann standardkonforme Java-Klassendateien ausführen, wie sie von einem Java-Übersetzer erzeugt werden. Zusammen mit der statischen Konfiguration erzeugt der

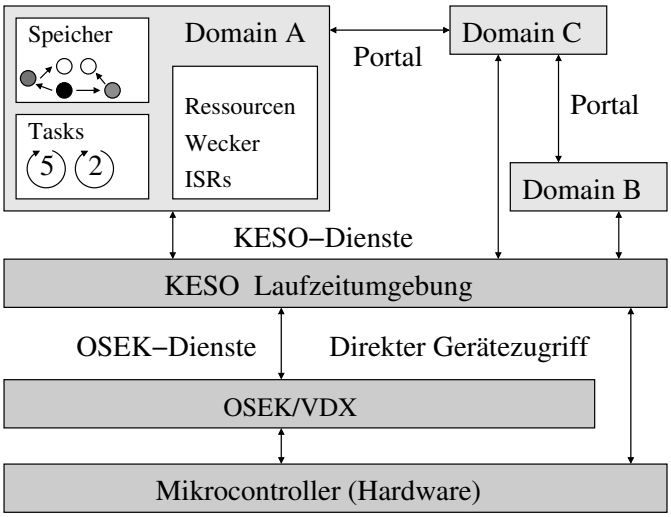

Abbildung 3.1: Die Architektur von KESO.

in Kapitel 5 beschriebene Generator eine Konfiguration für das OSEK/VDX-System und alle benötigten Dateien zum Erzeugen eines eigenständigen Maschinenprogramms, welches direkt auf dem Mikrocontroller lauffähig ist. Aufgrund der statischen Konfiguration ist es möglich, dass KESO und OSEK/VDX an die Anforderungen der Anwendungen automatisch angepasst werden.

Im Folgenden sollen die einzelnen Komponenten der Architektur genauer beschrieben werden.

3.2 OSEK/VDX als Basis für KESO

Als Basis für KESO dient ein OSEK/VDX-konformes Betriebssystem. OSEK/VDX ist eine offene Spezifikation für Betriebssysteme in Kraftfahrzeugen [75], welche speziell auf die Bedürfnisse der Steuergeräte und deren geringe Systemressourcen abgestimmt ist. Nachfolgend soll ein kurzer Überblick über die OSEK/VDX-Systemdienste gegeben werden. Sofern diese für die KESO-Architektur wichtig sind, sollen sie näher beleuchtet werden.

3.2.1 Überblick über die Systemdienste

KESO setzt als eine leichtgewichtige Laufzeitumgebung auf einem OSEK/VDX-System auf, mit dem Ziel, möglichst viele der positiven Eigenschaften von OSEK/VDX zu erhal-

ten und die bekannten Systemdienste dem Anwendungsentwickler über eine Bibliothek von Java-Klassen zur Verfügung zu stellen. Tabelle 3.1 zeigt die von der OSEK/VDX-Spezifikation definierten Dienste und nennt die gleichbedeutenden Klassen und Methoden aus der KESO-Klassenbibliothek.

Die OSEK/VDX-Spezifikation ist zwar unabhängig von der Programmiersprache, viele Eigenschaften der Schnittstelle wurden jedoch für die Programmiersprache C ausgelegt. Bei der Abbildung der C-Funktionsaufrufe auf die Programmiersprache Java wurden zum überwiegenden Teil statische Methoden verwendet und die Klassen dienen lediglich zur Gliederung der Dienste und nicht zum Instanzieren von Objekten. An einigen Stellen wurde jedoch auch dem objektorientierten Programmiermodell der Vortritt gewährt und es wurden Klassen und Objekte anstelle von primitiven Datentypen eingeführt.

Klassen anstelle von primitiven Datentypen zu verwenden kann Programmierfehler verhindern, da so der Übersetzer den Typ an der Schnittstelle überprüfen kann. Die Verwendung von Klassen führt jedoch auch zu einem größeren Speicherbedarf, da ein vollwertiges Java-Objekt mehr Speicher belegt als ein einfacher primitiver Datentyp. Ein Beispiel ist hier das Task-Objekt. Während in OSEK/VDX ein Task nur durch seine Task-ID repräsentiert wird, gibt es in KESO ein Task-Objekt. Das Task-Objekt verhindert, dass ein ungültiger Task bei einem Aufruf von **ActivateTask** angegeben werden kann. Das Objekt hat jedoch auch einen höhren Speicherbedarf, da es sowohl die Typinformation als auch die Task-ID enthält.

Die Kommunikation zwischen den Java-Anwendungen und den OSEK/VDX C Funktionen erfolgt über eine speziell für KESO entwickelte Schnittstelle. Diese wird in Kapitel 5.5 beschrieben. Im Vergleich zum üblicherweise verwendeten *Java Native Interface* (JNI) [64] ist diese Schnittstelle wesentlich leichtgewichtiger.

Nachfolgend sollen die für KESO wichtigen OSEK/VDX-Dienste, (Taskmodell, Synchronisation, Ereignisse und die Unterbrechungsbehandlung) noch genauer beschrieben werden, insoweit diese zum Verständnis für die Architektur von KESO wichtig sind. Für einen kompletten Überblick über die Funktionsweise von OSEK/VDX soll hier auf die entsprechende OSEK/VDX-Spezifikation verwiesen werden.

3.2.2 Das Taskmodell von OSEK/VDX

In einem OSEK/VDX-System wird die Einheit zur Auftragsplanung als Task bezeichnet. Jeder Task wird zum Generierungszeitpunkt fest angelegt. Ein Task kann während der Laufzeit weder erzeugt noch zerstört werden. Jedem Task ist eine feste Priorität zugewiesen, die sich, mit Ausnahme für die im Kapitel 3.2.3 noch beschriebene Synchronisation, nicht verändert. Ein einfacher Task, in der Spezifikation auch als **basic task** bezeichnet, kann, wie in Abbildung 3.2 gezeigt, nur die drei Zustände „**Suspendiert**“, „**Bereit**“ oder

OSEK/VDX	KESO			
Auftragsplanung	keso/core/TaskService	Task	ISR 1	ISR 2
ActivateTask	activate	•		•
TerminateTask	terminate	•		
ChainTask	chain	•		
Schedule	schedule	•		
GetTaskID	getTaskID	•		•
	getTaskByName	•		•
GetTaskState	getTaskState	•		•
Unterbrechungsbehandlung	keso/core/InterruptService			
DisableAllInterrupts	disableAll	•	•	•
EnableAllInterrupts	enableAll	•	•	•
SuspendAllInterrupts	suspendAll	•	•	•
ResumeAllInterrupts	resumeAll	•	•	•
SuspendOSInterrupts	suspendOS	•	•	•
ResumeOSInterrupts	resumeOS	•	•	•
Synchronisation	keso/core/RessourceService			
	getResourceByName	•		•
GetResource	getResource	•		•
	getScheduler	•		•
ReleaseResource	releaseResource	•		•
	releaseScheduler	•		•
Ereignisse	keso/core/EventService			
SetEvent	setEvent	•		•
GetEvent	getEvent	•		•
ClearEvent	clearEvent	•		
WaitEvent	waitEvent	•		
Alarmierung	keso/core/AlarmService			
GetAlarmBase	getAlarmBase	•		•
GetAlarm	getAlarm	•		•
SetRelAlarm	setRelAlarm	•		•
SetAbsAlarm	setAbsAlarm	•		•
CancelAlarm	cancelAlarm	•		•
Moduswechsel	keso/core/OSService			
GetActiveApplicationMode	getActiveApplicationMode	•		•
StartOS	startOS			
ShutdownOS	shutdownOS	•		•

Tabelle 3.1: Übersicht über die OSEK/VDX-Systemdienste
Übersicht über die OSEK/VDX-Systemdienste und die Entsprechungen in KESO

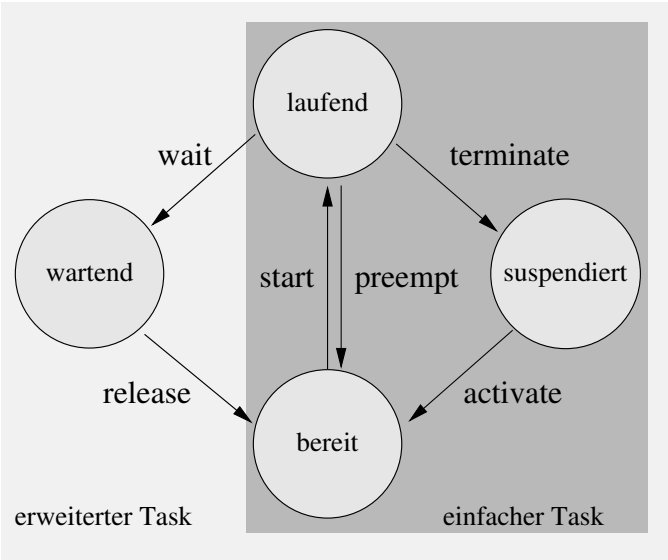

Abbildung 3.2: Zustandsmodell von OSEK/VDX und KESO.

„**Laufend**“ einnehmen. Soll ein Task zusätzlich noch die Funktionalität besitzen, auf ein Ereignis warten zu können, so muss dieser Task auch den Zustand „**Wartend**“ einnehmen können. Ein Task, der diesen zusätzlichen Zustand einnehmen kann, wird in OSEK/VDX als **extended task** bezeichnet und muss in der Konfiguration als solcher gekennzeichnet werden.

Ein Task wird immer strikt nach Priorität eingeplant. Gibt es mehrere Tasks mit gleicher Priorität, so werden diese in der Aktivierungsreihenfolge abgearbeitet. Ein OSEK/VDX-System kann die Einplanung wahlweise verdrängend oder nicht verdrängend vornehmen. In einer nicht verdrängenden Konfiguration gibt es genau vier Systemaufrufe, die einen Taskwechsel bewirken können. Ein Taskwechsel findet dann nur beim Beenden eines Tasks (**TerminateTask**, **ChainTask**), beim Warten auf ein Ereignis (**WaitEvent**) oder bei der expliziten Abgabe der Rechenzeit (**Schedule**) statt, jedoch nicht direkt nachdem ein höher-priorer Task lauffähig geworden ist.

Die von OSEK/VDX bereitgestellte Auftragsplanung erlaubt eine aufgabenorientierte Gliederung der Anwendungssoftware bei gleichzeitig sehr geringer Speichernutzung.

3.2.3 Synchronisation

In OSEK/VDX kann ein Task eine Ressource exklusiv belegen, indem er sie mit **GetResource** anfordert. Die Synchronisation wird dabei über das *Immediate Ceiling Priority Protocol* (ICPP) bewerkstelligt, welches eine Abwandlung vom *Original Ceiling Priority*

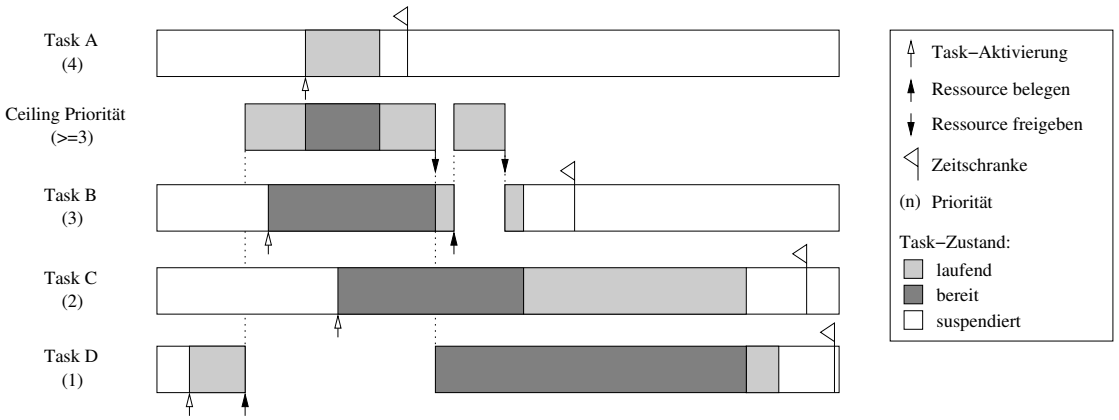

Abbildung 3.3: Immediate Ceiling Priority Protocol

Protocol (OCPP) [88] darstellt und dieses wesentlich vereinfacht. Bei dem ICPP wird die Task-Priorität unmittelbar bei einer Ressourcenbelegung für die gesamte Dauer der Belegung auf eine vorab bestimmte Ceiling-Priorität angehoben. Die Ceiling-Priorität muss dabei immer größer oder gleich der höchsten Priorität aller Tasks sein, die die Ressource zu einem beliebigen Zeitpunkt belegen können.

Aufgrund der angehobenen Priorität wird ein Task für die Dauer des kritischen Abschnittes nicht durch einen anderen Task mit gleicher oder niedrigerer Ceiling-Priorität verdrängt. In der Abbildung 3.3 konkurrieren der **Task B** und der **Task D** um eine gemeinsam genutzte Ressource. Der **Task D** fordert die Ressource dabei vor der Aktivierung von **Task B** an und wird daher auf mindestens die gleiche Priorität angehoben. Nach der Aktivierung von **Task B** läuft zunächst **Task D** weiter, aufgrund der angehobenen Priorität und der Tatsache, dass bei einem Task mit gleicher Priorität der zuerst aktivierte Task abgearbeitet wird. **Task D** wird erst verdrängt, wenn er die Ressource wieder freigibt oder wenn ein Task mit höherer Priorität aktiviert wird. In der Abbildung wäre dies **Task A**. Voraussetzung dafür ist jedoch, dass **Task A** die Ressource nie anfordert, ansonsten wäre die Ceiling-Priorität entsprechend höher.

Das ICPP verhindert daher, dass ein zweiter konkurrierender Task für die Dauer der Ressourcenbelegung in den Zustand „**Laufend**“ übergeht. Da der Task mit der Ceiling-Priorität nicht mehr verdrängt wird, ist es nicht nötig, dass ein auf eine Ressource wartender Task explizit verwaltet wird. Das Zustandsmodell benötigt aus diesem Grund für die Synchronisation auch nicht den Zustand „**Wartend**“. Der wartende Task bleibt im Zustand „**Bereit**“ und belegt zu diesem Zeitpunkt noch keinen Speicher für den Funktionsstack, und es muss auch keine Warteschlange für die Ressourcen im System verwaltet werden. Diese Art der Synchronisation hat daher einen sehr geringen Speicherbedarf.

Außerdem gelten für ICPP noch zwei weitere für das Anwendungsgebiet nützliche Ei-

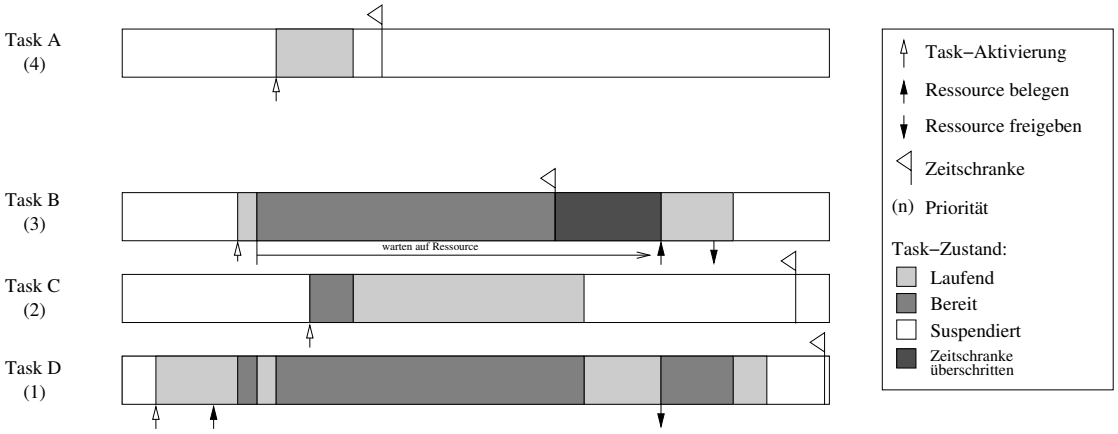

Abbildung 3.4: Synchronisation ohne Prioritätsvererbung

genschaften. Erstens verhindert das Protokoll wirkungsvoll eine Prioritätsumkehr. In der Abbildung 3.3 würde die Aktivierung von **Task C** bei einem Synchronisationsverfahren mit fester Priorität und ohne Prioritätsvererbung eine Prioritätsumkehr auslösen, da **Task C** den **Task D** verdrängen und so den eigentlich höher priorisierten **Task B** verzögern würde. Die Abbildung 3.4 zeigt diesen Fall der Prioritätsumkehr. Als Folge kommt es in diesem Fall zu einer Überschreitung der für **Task B** veranschlagten Zeitschranke.

Zweitens ist die Synchronisation mit dem ICPP verklemmungsfrei, da die Ceiling-Priorität der Ressource immer von der höchsten Priorität bestimmt wird, mit der ein Task eine Ressource anfordern kann. Daraus folgt, dass die Ceiling-Priorität für alle Ressourcen, die eine Gruppe von Tasks gemeinsam nutzt, gleich der höchsten Priorität aller in der Gruppe befindlichen Tasks ist. Werden zwei getrennte Ressourcen von einer Gruppe von Tasks gleichzeitig belegt, so ist deren Ceiling-Priorität gleich hoch und mit der Belegung der ersten Ressource ist auch die Belegung der zweiten sichergestellt, da kein konkurrierender Task mehr diese Ressource anfordern kann.

Die Ceiling-Priorität wird bei OSEK/VDX zum Generierungszeitpunkt für jede Ressource berechnet. Zu diesem Zweck müssen in der OSEK/VDX-Konfiguration die von einem Task verwendeten Ressourcen korrekt angegeben werden. Konfigurationsfehler werden von den Generierungswerkzeugen zum Generierungszeitpunkt nicht erkannt, da diese das C-Programm nicht analysieren können.

3.2.4 Systemereignisse

Eine weitere für die Architektur von KESO wichtige Funktionalität sind die Ereignisse. OSEK/VDX bietet die Möglichkeit, dass ein Task auf ein Ereignis wartet. Wie bereits

bei der Beschreibung des Taskmodells erwähnt, kann nur ein **extended task** diese Funktionalität nutzen. Die Ereignisse, auf die ein Task wartet, werden wieder zum Konfigurationszeitpunkt festgelegt. In der OSEK/VDX-Konfiguration müssen hierzu für jeden Task die verwendeten Ereignisse angegeben werden.

Ereignisse können von unterschiedlichen Quellen ausgelöst werden, wie z.B. einem Systemwecker oder einer Unterbrechungsbehandlung. Die Programmierschnittstelle für die Ereignisse ist wieder auf einen geringen Ressourcenbedarf abgestimmt. So muss beim Setzen eines Ereignisses der empfangende Task explizit mit angegeben werden. Das System muss daher keine Warteschlange für Ereignisse bereitstellen. Der zum Aufwecken bestimmte Task wird durch diese Semantik beim Aufruf bereits übergeben und kann direkt wieder für die Auftragsplanung in den Zustand **Bereit** übergehen.

Im Zusammenhang mit der Synchronisation ist der Ereignismechanimus kritisch. Da das Warten während der Ressourcenbelegung direkt das ICPP verletzen würde, ist das Warten während einer Ressourcenbelegung verboten. Es müssen daher laut Spezifikation alle Ressourcen vor dem Aufruf von **WaitEvent** wieder freigegeben werden.

Trotz dieser Einschränkung kann die Verwendung von Ereignissen weiterhin dazu führen, dass die Software nicht mehr verklemmungsfrei ist, ebenso ist eine Prioritätsumkehr wieder möglich. Die Möglichkeit zur Verklemmung und zur Prioritätsumkehr veranschaulicht das Programmbeispiel in Abbildung 3.5. Dargestellt ist eine mögliche Implementierung für eine einfache binäre Semaphore, die zur Synchronisation zwischen zwei Tasks verwendet wird.

Eine Anwendung, die auf diese Art implementierte Semaphoren verwenden würde, wäre weder verklemmungsfrei noch würde eine Prioritätsumkehr ausgeschlossen. Das Beispiel ist für KESO insofern interessant, als sich bei der Implementierung der JVM die Frage nach der Abbildung des in Java üblichen Synchronisationsmodelles stellt.

Ein weiterer Nachteil von Ereignissen ist der größere Speicherbedarf, da der Funktionsstack von einem Task, der auf ein Ereignis wartet, nicht für die Ausführung eines anderen Tasks genutzt werden kann. Ein Task, der hingegen nur auf eine Aktivierung wartet, hat einen geringeren Speicherbedarf, und für viele Anwendungen reicht die Task-Aktivierung als Ereignis aus.

3.3 Domäne: Schutzraum und JVM-Instanz

3.3.1 Überblick

KESO erweitert OSEK/VDX um einen konstruktiven Speicherschutz. Hierzu wird das System in Schutzräume unterteilt. Ein Schutzraum wird in KESO als Domäne bezeichnet.

```
#include "os.h"

typedef struct {
        int value;
        TaskType waiting_task;
} sem_t;

void P(sem_t* sem) {
        GetResource(SEM_LOCK);
        while (sem->value==0) {
                GetTaskID(&(sem->waiting_task));
                ReleaseResource(SEM_LOCK);
                WaitEvent(SEM_CHANGE);
                ClearEvent(SEM_CHANGE);
                GetResource(SEM_LOCK);
        }
        sem->value--;
        ReleaseResource(SEM_LOCK);
}

void V(sem_t* sem) {
        GetResource(SEM_LOCK);
        sem->value++;
        if (sem->waiting_task!=INVALID_TASK) {
                SetEvent(sem->waiting_task, SEM_CHANGE);
                sem->waiting_task = INVALID_TASK;
        }
        ReleaseResource(SEM_LOCK);
}

sem_t sem1 = { 1 , INVALID_TASK };
sem_t sem2 = { 1 , INVALID_TASK };

TASK(TaskB) {
        P(&sem1); P(&sem2);
        /* kritische Abschnitt */
        V(&sem2); V(&sem1);
}

TASK(TaskD) {
        P(&sem2); P(&sem1);
        /* kritische Abschnitt */
        V(&sem1); V(&sem2);
}
```

Abbildung 3.5: Beispiel für Semaphoren basierend auf Ressourcen und Ereignissen

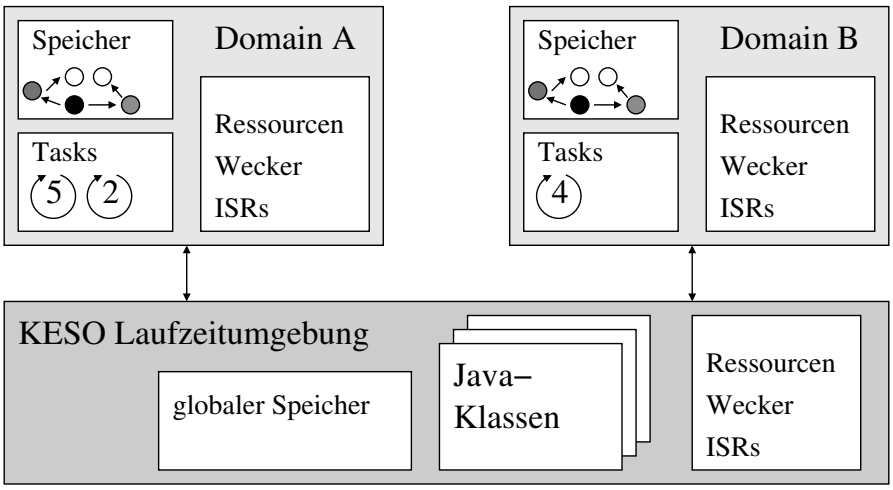

Abbildung 3.6: Überblick Domäne

Jede Domäne hat jeweils einen eigenen Speicherbereich und nutzt diesen exklusiv. Die Speicherbereiche werden fest einer Domäne zugeordnet und es ist nicht möglich, zur Laufzeit eine Domäne zu erzeugen bzw. zu zerstören. Aus der Sicht der Anwendung ist jede Domäne eine eigenständige, isolierte Laufzeitumgebung in Form einer eigenständigen JVM. Als solche besitzt jede Anwendung ihre eigenen globalen Variablen und eine eigenständige Speicherverwaltung für die Objekte. Es gibt keine Objektreferenz, die die Grenzen der Domäne überschreiten würde und über die es möglich wäre, den Speicher einer anderen Domäne zu lesen oder gar zu verändern.

Die komplette Funktionalität einer JVM, wie sie die *Java 2 Enterprise Edition* (J2EE) oder die *Java 2 Standard Edition* (J2SE) bieten, ist auf einem Gerät mit eingeschränkten Ressourcen nicht umsetzbar. In KESO kommt daher eine auf die Anforderungen abgestimmte JVM zum Einsatz. Dieser Ansatz ist vergleichbar mit der *Java 2 Micro Edition* (J2ME). Die J2ME kennt unterschiedliche Konfigurationen, die den jeweiligen Funktionsumfang der JVM so einschränken, dass die Anforderungen vom Anwendungsfeld erreicht werden können [60]. Eine existierende Konfiguration, die den Anforderungen von KESO am nächsten kommt, ist die *Connected, Limited Device Configuration* (CLDC). Diese Konfiguration ist für Geräte mit einem Arbeitsspeicher von 32–512 kByte ausgelegt.

Doch auch die CLDC passt noch nicht ideal zu den Anforderungen von KESO. So unterstützt die CLDC das dynamische Laden von Klassen, wenn auch nur sehr eingeschränkt, was in unserem Fall nicht benötigt wird. Andererseits werden Fließkommaoperationen nicht unterstützt, welche vielleicht benötigt würden. KESO hat daher eine eigene, auf die Anforderungen zugeschnittene, Konfiguration. Die Tabelle 3.2 zeigt die Einschränkungen für KESO-Anwendungen und stellt diese der CLDC gegenüber.

Funktionalität	CLDC	KESO
Fliesskommaarithmetik	nicht unterstützt	unterstützt, alternativ Festkommaarithmetik
Java Native Interface (JNI)	nicht unterstützt	KNI
Reflektion	nicht unterstützt	nicht unterstützt
Thread	eingeschränkt	Task
Fehlerbehandlung	Java Exceptions eingeschränkt	OSEK/VDX ErrorHook, Java Exception optinal
Laden von Klassen	eingeschränkt	nicht unterstützt
Weiche Objektreferenzen	nicht unterstützt	nicht unterstützt
Synchronisation	Java-Monitor	Ressourcen, Java-Monitor optinal

Tabelle 3.2: Gegenüberstellung CLDC und KESO-JVM

Die Verwendung einer eigene Konfiguration liegt in dem Bestreben, die Eigenschaften von OSEK/VDX weitestgehend zu erhalten. OSEK/VDX ist im Unterschied zu einer JVM ein statisches System, daher wird das Laden von Klassen und Reflektion nicht unterstützt. Die Auftragsplanung und die Synchronisation sind die zentralen Funktionen von OSEK/VDX. Diese zu verändern würde bedeuten, ein System mit einer anderen Zielsetzung zu haben. Die Java-Exception und das JNI besitzen für unsere Anforderungen einen zu großen Speicherbedarf und wurden daher durch schlankere Konzepte ersetzt. All diese Änderungen betreffen nicht die Typsicherheit von Java.

3.3.2 Aktivitätsträger: Task vs. Thread

KESO übernimmt das in Kapitel 3.2.2 beschriebene Taskmodell von OSEK/VDX. Die KESO-Programmierschnittstelle kennt daher, anstelle des aus der Standard Java Bibliothek bekannten Threads, den Task als Einheit zur Auftragsplanung. In diesem Punkt wurde bewusst dem schlankeren Konzept von OSEK/VDX der Vorzug gegeben, um Ressourcen zu sparen und den Anforderungen für die Entwicklung von echtzeitfähigen Anwendungen ein einfacher zu analysierendes Modell zur Auftragsplanung zu bieten. Der Unterschied in den Eigenschaften zwischen Java-Thread und OSEK/VDX-Task soll durch die geänderte Namensgebung klar zum Ausdruck gebracht werden.

Jeder Task wird in KESO durch ein Task-Objekt repräsentiert. Ein Task wird wie bei OSEK/VDX in der Konfiguration spezifiziert und kann nicht dynamisch zur Laufzeit erzeugt oder zerstört werden. Ein Task-Objekt ist einer Domäne fest zugeordnet. Alle vom Task erzeugten Objekte werden von der domänspezifischen Speicherverwaltung verwaltet.

Ein Task-Objekt wird im Regelfall direkt auf einen OSEK/VDX-Task abgebildet. Im Zusammenhang mit der in Kapitel 3.4.4 beschriebenen Inter-Domän-Kommunikation können weitere Task-Objekte in einer Domäne entstehen. Da in OSEK/VDX zur Laufzeit keine weiteren Tasks erzeugt werden können, wechselt in diesem Fall der OSEK/VDX-Task die Domäne, nicht aber das Task-Objekt.

3.3.3 Synchronisation: OSEK/VDX-Resource vs. Java-Monitor

Die Programmiersprache Java besitzt einen integrierten Sychronisationsmechanismus, welcher bis auf die Ebene der JVM durchgezogen wurde. Jedes Objekt der JVM kann sowohl als eine binäre Semaphore als auch zum Warten auf Ereignisse dienen. Beim Warten auf ein Ereignis wird implizit die zugehörige Semaphore freigegeben und beim Auslösen des Ereignisses wird diese wieder implizit angefordert.

Für die Programmiersprache Java ist diese Funktionalität gut geeignet, da für die Sychronisation von Methoden bereits ein Objekt zur Synchronisation vorhanden ist. Aus Sicht des Ressourcenbedarfs ist dies jedoch denkbar ungünstig, da so jedes Objekt zusätzlichen Speicherplatz für die von der Semaphore und der Warteschlange benötigten Datenstrukturen benötigt.

Es ist möglich, die Sychronisation mit Hilfe der von OSEK/VDX gebotenen Dienste nachzubilden. Eine Variante wurde bereits in Kapitel 3.2.3–3.2.4 mit dem Programmbeispiel in Abbildung 3.5 angedeutet. Eine vollständige Umsetzung würde bedeuten, dass mit jedem Objekt eine passende Datenstruktur für eine Semaphore und bis zu zwei Warteschlangen assoziiert werden können. Dies würde den Speicherbedarf von Objekten erhöhen.

Selbst wenn bei einer geschickten Implementierung und ungenutzter Funktionalität, wie sie Bacon et al. [19] vorschlägt, nur ein Bit pro Objekt extra anfallen würde, so würden durch diesen Ansatz die zwei wichtigen Eigenschaften, die das Synchronisationmodell von OSEK/VDX bietet – frei von einer Prioritätsumkehr und Verklemmungen zu sein – verloren gehen.

In KESO wird daher die von Java bekannte Sychronisation nur teilweise und auch nur optional unterstützt. Die Synchronistation wird dabei direkt auf die Belegung einer Ressource mit höchster Priorität im gesamten System abgebildet und das Warten auf Ereignisse wird nicht unterstützt. Damit können einfache Methoden bei Bedarf mit den Java-Sprachkonstrukten sychronisiert werden.

Für eine feingranulare Sychronisation werden die in Kapitel 3.2.3 beschriebenen OSEK/VDX-Dienste dem Programmierer über die Klassen-Bibliothek zur Verfügung gestellt. Diese sollten zur Synchronisation bevorzugt benutzt werden.

3.3.4 Unterbrechungsbehandlung (Interrupt processing)

Zur Behandlung von Unterbrechungen kennt die OSEK/VDX-Spezifikation zwei Klassen von *Interrupt Services Routinen* (ISR). Die Unterbrechungsbehandlung der Klasse 1 (ISR1) darf laut Spezifikation fast keine OSEK/VDX-Systemaufrufe nutzen. Ausgenommen sind lediglich die Funktionen zum Sperren und Freigeben von Unterbrechungen.

Die Unterbrechungsbehandlung der Klasse 2 (ISR2) ist dem Task fast gleichgestellt und nur wenige Systemdienste sind von der Verwendung ausgeschlossen. Die Einschränkungen können der Tabelle 3.1 entnommen werden.

In der KESO-Konfiguration können sowohl ISR1 als auch ISR2 spezifiziert werden. ISR1 werden jedoch nur eingeschränkt unterstützt, da diese außerhalb einer Domäne liegen und nur die Dienste der **InterruptService**-Klasse verwenden dürfen.

ISR2 werden ähnlich wie ein Task behandelt. Als eine zusätzliche Einschränkung darf eine Unterbrechungsroutine jedoch keine Objekte anlegen. Diese Einschränkung ermöglicht die Implementierung einer Speicherverwaltung ohne das Sperren von Unterbrechungen und so eine möglichst geringe Latenz.

3.3.5 Zähler, Alarmierung und Ereignisse

Neben einem Taskmodell und der passenden Sychronisation bietet die OSEK/VDX-API noch Zähler (**Counter**), eine Alarmierung (**Alarm**) und Ereignisse (**Events**). Diese Dienste werden wie bei Tasks und Ressourcen wieder statisch in der Konfiguration beschrieben und können nicht dynamisch zur Laufzeit erzeugt oder zerstört werden. Auch diese Dienste wurden für KESO übernommen. Die Abbildung 3.7 zeigt eine KESO-Konfiguration, die alle drei Dienste nutzt.

Ein Zähler kann sowohl Software- als auch Hardwareereignisse zählen. In dem Beispiel zählt der Zähler mit der Bezeichnung **counter1** die Anzahl von zeitgesteuerten Unterbrechungen, ausgelöst von einem Zeitgeber des TriCore-Mikrocontrollers. Nicht alle Parameter für die Konfiguration der Hardware sind von der OSEK/VDX-Spezifikation abgedeckt, so dass jedes OSEK/VDX-System und jeder Mikrocontroller eigene Parameter kennt. KESO reicht diese Parameter für die Hardwarekonfiguration einfach an die Konfiguration des OSEK/VDX-Systems durch.

Ein Alarm dient zum Überwachen von Zählern und kann beim Erreichen von entsprechenden Zählerständen unterschiedliche Aktionen auslösen. Im Beispiel wird der Zähler **counter1** vom Alarm **alarm1** überwacht und die ausgelöste Aktion ist das Setzen des Ereignisses **Wakeup**. Alternativ kann ein Alarm auch direkt einen Task aktivieren oder eine Funktion aufrufen. Im Beispiel nutzt der Task **task1** das **Wakeup**-Ereignis. Der

```
Counter (counter1) {
        MAXALLOWEDVALUE = "10";
        MINCYCLE = "1";
        TICKSPERBASE = "1";
        TRICORE_TIMER = "STM_T0";
        TRICORE_TIMER = "STM_T0";
        TIME_IN_NS = "100000000";
}

Event (Wakeup) {
        Mask = "Auto";
}

Domain (dom1) {
        Task (task1) {
                MainClass="test/HelloWorld";
                Autostart = true {
                        Appmode = "OSDEFAULTAPPMODE";
                }
                Priority = "5";
                UseEvent = "Wakeup";
        }

        Alarm (alarm1) {
                UseCounter = "counter1";
                UseEvent = "Wakeup";
                Autostart = true {
                        Appmode = "OSDEFAULTAPPMODE";
                        Alarmtime = "1";
                        Cycletime = "1";
                }
        }
}
```

Abbildung 3.7: Beispiel einer KESO-Konfiguration mit Zähler, Alarmierung und Ereignis

Task kann daher auf das Ereignis warten und wird so von der Hardware in regelmäßigen Abständen wieder aufgeweckt.

Die Domängrenze dient in KESO dazu, die Sichtbarkeit von Systemobjekten einzuschränken. Der Task **task1** und der Alarm **alarm1** sind derselben Domäne **dom1** zugeordnet, daher kann der Task den Alarm deaktivieren oder abfragen. Ein Task aus einer anderen Domäne kann auf den Alarm nicht direkt zugreifen. Das global definierte **Wakeup**-Ereignis kann jedoch auch in einer anderen Domäne genutzt werden.

3.3.6 Fehlerbehandlung

Fehler werden bei OSEK/VDX über den Rückgabewert von Funktionen oder über eine globale Fehlerbehandlung angezeigt. Die globale Fehlerbehandlung besteht aus einer registrierten Funktion mit dem Namen **ErrorHook**. Diese kann dazu dienen, Fehler für spätere Diagnosezwecke zu protokollieren oder das System in einen Notbetrieb schalten. Für eine aufwändige und spezialisierte Reaktion auf einen Fehler ist ein **ErrorHook** nicht geeignet. Auf einem System mit stark eingeschränkten Ressourcen sind einfache und robuste Reaktionen auf Fehler ausreichend und erwünscht. Eine robuste Reaktion wäre zum Beispiel das Anzeigen des Fehlers und das Umschalten in einen Notbetrieb, bis der Fehler in der Werkstatt analysiert und behoben wird.

Im Gegensatz dazu bietet Java in der Sprache eine integrierte Ausnahmebehandlung. Die Umsetzung dieser führt jedoch zu einem größeren Speicherbedarf und in vielen Fällen ist eine aufwändige Fehlerbehandlung in der Anwendungssoftware nicht erwünscht, da eine Fehlerbehandlung selbst wieder fehlerbehaftet sein kann. Es liegt sogar die Vermutung nahe, dass die meisten Fehler in Fehlerbehandlungen existieren, da diese selten durchlaufen werden und oft in den Testfällen nicht ausreichend berücksichtigt worden sind.

Aufgrund des erhöhten Speicherbedarfs für die Java-Exceptions wurde in KESO eine globale Fehlerbehandlung bevorzugt. Wird eine Java-Exception ausgelöst, so wird eine domänspezifische Fehlerbehandlung angesprungen. Das Vorgehen ist vergleichbar mit dem **ErrorHook** in OSEK/VDX, ermöglicht jedoch eine domänspezifische Fehlerbehandlung. Eine andere Domäne kann daher ihren Dienst weiterhin verrichten.

3.3.7 Gemeinsam genutzte Klassen

Die JVM der Domäne kann Programme im standardisierten Java-Klassendateiformat ausführen. Die Klassendateien können mit einem normalen Übersetzer für die Program-

miersprache Java erzeugt werden. Es ist auch möglich, eine andere Programmiersprache[1] zu verwenden, solange ein entsprechender Übersetzer nach Java-Bytecode existiert.

Im Unterschied zu einer JVM mit einem dynamischen Klassenlader müssen bei KESO alle verwendeten Klassen bereits zum Generierungszeitpunkt bekannt sein. Während der Generierung wird das Laden und Binden der Klassen simuliert. Hierdurch wird der Startvorgang auf dem Mikrocontroller stark verkürzt und die Routinen sowie benötigte Symbolinformationen zum Laden und Binden müssen nicht auf dem Mikrocontroller vorhanden sein.

Die geladenen Klassendateien werden von allen JVMs auf einem Mikrocontroller gemeinsam genutzt, dadurch wird der Speicherbedarf für die Programmdaten stark verringert und der Einsatz mehrerer JVMs mit einem akzeptablen Ressourcenbedarf ermöglicht.

3.3.8 Separate Speicherverwaltung

Die Spezifikation der JVM [66] sieht für die Speicherverwaltung eine automatische Speicherbereinigung vor. Diese entlastet den Programmentwickler von einer expliziten Verwaltung des Speichers und vermeidet Programmierfehler. Im Hinblick auf die geringen Ressourcen und die Echtzeitanforderung ist eine nicht spezialisierte automatische Speicherbereinigung für den Einsatz in dem angestrebten Einsatzgebiet jedoch ungeeignet.

Die strikte Trennung der Speicherbereiche ermöglicht es, dass jede Domäne ihre eigenständige automatische Speicherverwaltung besitzt und so für die unterschiedlichen Anforderungen der Anwendungen unterschiedliche Strategien zur Speicherbereinigung umgesetzt werden können.

In Rahmen dieser Arbeit wurden drei grundlegend verschiedene Strategien implementiert. Im Kapitel 3.5 werden diese vorgestellt. Jede Domäne kann dabei eine andere Strategie verfolgen. So ist es möglich, die Speicherverwaltung flexibel an die Anforderungen der Anwendung anzupassen.

Mit der aktuellen Implementierungen bestehen noch Abhängigkeiten zwischen den Domänen. So werden der Task zur Speichebereinigung und einige Datenstrukturen von unterschiedlichen Speicherverwaltungen gemeinsam genutzt. Diese Abhängigkeiten bestehen jedoch nur, um den Ressourcenbedarf möglichst gering zu halten. Die Architektur würde auch hier eine vollständige Entkopplung erlauben.

[1] Die Website http://www.robert-tolksdorf.de/vmlanguages.html listete im Dezember 2007 eine Vielzahl von weiteren Übersetzern für andere Programmiersprachen auf. Darunter waren die Sprachen C, C#, Fortran, Lisp, Scheme, Prolog und Ada.

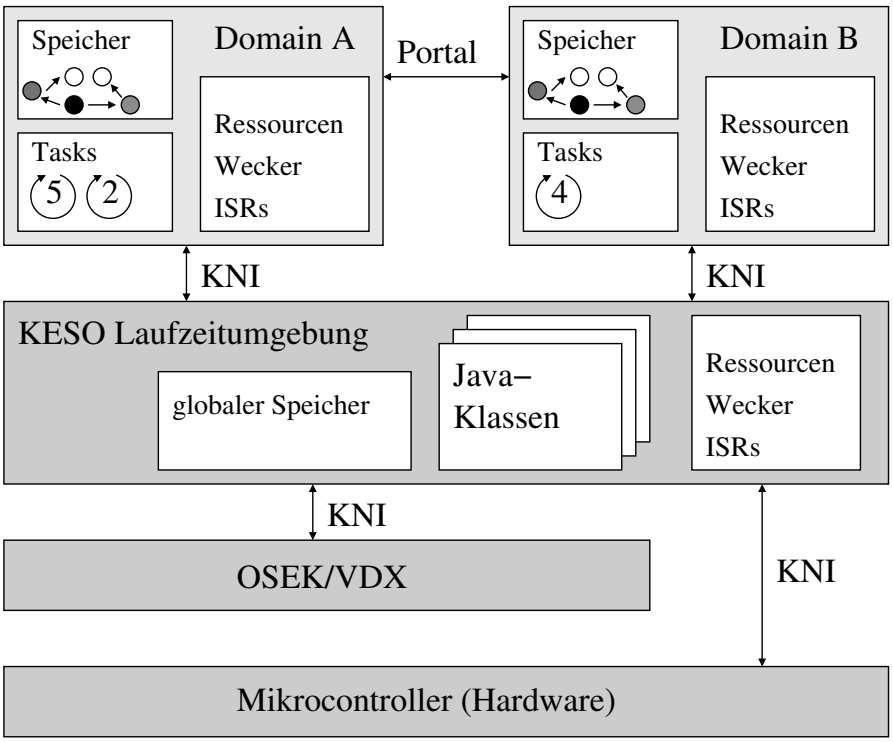

Abbildung 3.8: Interdomänenkommunikation

3.4 Interdomänenkommunikation

3.4.1 Einleitung

Die Abgrenzung von Anwendungen in Schutzräumen verhindert unkontrollierte Fehlerausbreitung auf einem Mikrocontroller, indem die Anwendungen gegeneinander isoliert werden. Jede sinnvolle Anwendung muss jedoch auch mit ihrer Umgebung kommunizieren können. In KESO gibt es neben den von der OSEK/VDX-Programmierschnittstelle angebotenen Diensten einige zusätzliche Dienste zur Kommunikation und Integration von nativen Maschinenprogrammen.

Um Software möglichst feingranular in Schutzräume einteilen zu können, muss der Mehraufwand für die Kommunikation zwischen den Schutzräumen möglichst gering sein. Im Zusammenhang mit der Interdomänenkommunikation von JX wurde bereits gezeigt, dass eine Kommunikation zwischen Schutzräumen in einem System mit softwarebasiertem Speicherschutz wesentlich effizienter möglich ist, als bei hardwarebasiertem Speicherschutz [102]. Für KESO wurden diese Verfahren angepasst und im Hinblick auf den Ressourcenbedarf und die Effizienz verbessert.

3.4.2 Native Schnittstelle (KNI)

Um die von OSEK/VDX angebotenen Dienste in der JVM nutzen zu können, wird eine Schnittstelle benötigt, mit der es möglich ist, die in C implementierten Funktionen und Datenstrukturen von KESO aus anzusprechen. Für das Binden von nativen Programm-Bibliotheken von einer JVM aus gibt es bereits unterschiedliche Lösungen, die Standard-Lösung hierfür ist das *Java Native Interface* (JNI). Diese Schnittstelle ist für eine möglichst große Kompatibilität zwischen unterschiedlichen JVMs und nativen Bibliotheken ausgelegt und nicht auf einen möglichst minimalen Zusatzaufwand aus. Für KESO wurde daher eine eigene Schnittstelle entwickelt, die es ermöglicht, an definierten Aufrufpunkten anstelle des Generators die Generierung des C-Programms zu übernehmen und so direkt die benötigten OSEK/VDX-Systemaufrufe in die Anwendung einzuweben.

Das Einweben der OSEK/VDX-Systemaufrufe in die Java-Anwendung sorgt dafür, dass ein Mehraufwand in vielen Fällen vermieden wird. Lediglich zusätzliche Funktionalität, wie die Verwendung von Task-Objekten anstelle von einfachen IDs, führt an der Schnittstelle zu einem zusätzlichen Aufwand. Ein genauerer Vergleich, der die Unterschiede zwischen JNI und KNI verdeutlicht, wird noch in Kapitel 5.5 angestellt.

Das KNI dient nicht nur zur Umsetzung der OSEK/VDX-Schnittstelle, sondern auch für zusätzliche Dienste, die für eine hardwarenahe Programmierung benötigt werden. Ein Beispiel dafür ist die Schnittstelle für einen nativen Speicherzugriff. Dieser Dienst ermöglicht es, typisierten und untypisierten Speicher von mehreren Schutzräumen aus gemeinsam zu nutzen oder auch in den Speicher eingeblendete Geräteregister direkt anzusprechen.

Auch architekturspezifische Funktionen und bereits bestehende Programmbibliotheken können über diese Schnittstelle genutzt werden. Die Schnittstelle schwächt jedoch das Sicherheitskonzept. Alle über die Schnittstelle eingewobenen Programmteile müssen als vertrauenswürdig gelten und die übergebenen Parameter müssen überprüft werden.

Ein Aufruf am KNI entspricht dem Wechsel in den privilegierten Modus bei einem Systemaufruf, wie es bei einem Betriebssystemkern mit hardwarebasiertem Speicherschutz geschieht. Nach dem Wechsel in den nativen Bereich ist kein Speicherschutz mehr gewährleistet. Dem durch das KNI angesprungenen Programm muss vertraut werden. Daher sollten möglichst wenige Programmabschnitte im nativen Teil der Anwendung implementiert werden. Deshalb ist es sinnvoll, möglichst einfache und primitive Funktionen über das KNI zur Verfügung zu stellen.

Dazu zählen spezielle Maschinenbefehle, die z.B. zum Auslesen und Schreiben von Geräteregistern dienen, oder Befehle für einen direkten Speicherzugriff, so dass auch Geräteregister angesprochen werden können. In dieser Arbeit wird noch gezeigt, dass der Mehraufwand durch das KNI so gering ist, dass ein Aufruf auch bei einfachen

Maschinenbefehlen nicht ins Gewicht fällt. Mit dem JNI wäre dies nicht so effizient möglich.

3.4.3 Nativer Speicherzugriff

Untypisierter Speicher

Der native Speicherzugriff ermöglicht den Zugriff auf untypisierte Speicherbereiche und ist ein gutes Beispiel für den effizienten Einsatz des KNI. Der Speicherdienst von KESO verwaltet den Zugriff auf untypisierten Speicher und Gerätespeicher und wird von der **MemoryService**-Klasse implementiert. Die Klasse ermöglicht das Anfordern von untypisiertem Speicher, wobei zwischen einfachem Speicher und Gerätespeicher sowie und zwischen einer dynamischen und einer statischen Speicheranforderung unterschieden wird.

Die Speicherbereiche liegen außerhalb der Domängrenzen und unterliegen nicht der domänspezifischen Speicherverwaltung. Der Zugriff erfolgt, wie in Abbildung 3.9 dargestellt, über Stellvertreterobjekte, die im Weiteren als Speicherobjekte bezeichnet werden. Speicherobjekte beschreiben einen klar definierten Speicherbereich, und beim Zugriff auf den Speicher stellen Bereichsüberprüfungen sicher, dass die Grenzen der Speicherbereiche nicht überschritten werden.

Mehrere Schutzräume können Speicherobjekte besitzen, die den gleichen Speicherbereich repräsentieren. Die Speicherobjekte können über die Domängrenzen hinweg ausgetauscht werden. Dabei werden statische Speicherobjekte und die repräsentierten Speicherbereiche nicht kopiert, sondern gemeinsam genutzt.

Abbildung 3.10 zeigt die Verwendung der Schnittstelle zusammen mit einer Anforderung für einen statischen Speicherbereich. Bei einem statischen Speicherbereich werden der Speicher und das Stellvertreterobjekt bereits zum Übersetzungszeitpunkt angelegt, und soweit möglich wird auch die Bereichsüberprüfung bereits zum Übersetzungszeitpunkt durchgeführt. Die Größe des Speicherbereiches muss dazu bereits zum Übersetzungszeitpunkt feststehen. Ein statisches Speicherobjekt bleibt für die gesamte Laufzeit des Systems bestehen.

Bei einem dynamischen Speicherobjekt, welches dazu dient, Speicherbereiche zu verwalten, die erst zur Laufzeit alloziert werden, wird das Speicherobjekt wie ein gewöhnliches Objekt im Speicher der Domäne angelegt. Der durch die Speicherobjekte repräsentierte Speicherbereich liegt wiederum außerhalb der Domäne und bleibt so lange erhalten, bis kein Speicherobjekt mehr diesen Speicherbereich referenziert. Zu diesem Zweck wird für den Speicherbereich ein Referenzzähler verwaltet. Der Referenzzähler wird beim Kopieren von Speicherobjekten erhöht und beim Entfernen von Speicherobjekten durch

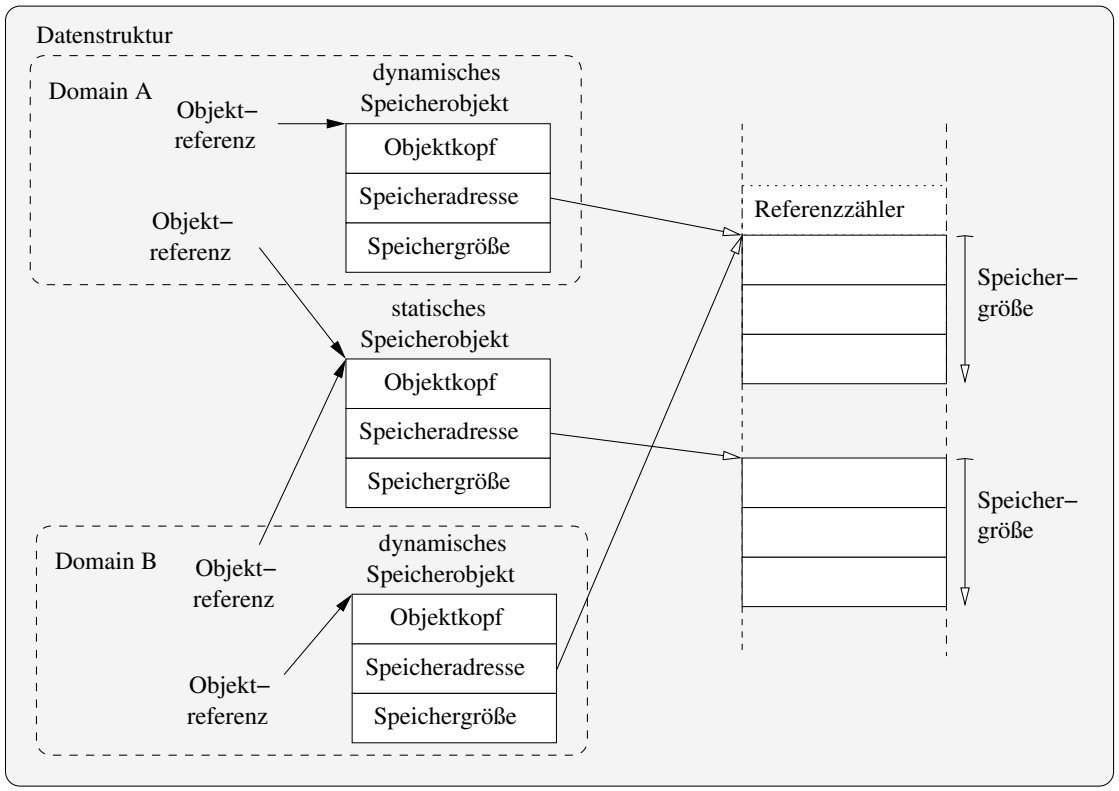

Abbildung 3.9: Die Speicherobjekte dienen als Stellvertreterobjekte für untypisierte Speicherbereiche. Statische Speicherobjekte können von mehreren Domains gemeinsam genutzt werden.

```
/*
 * Statische Belegung von einem Speicher-
 * bereich mit einer Speichergröße von 7 Byte.
 */
memobj = MemoryService.allocStaticMemory(7);

/* schreiben */
memobj.set8(6, 255);

/* lesen */
int value8bit = memobj.get8(6);

/*
 * Bei diesem Aufruf, der einen 32 Bit Wert liest,
 * tritt ein Fehler auf, da der Zugriff außerhalb
 * des Speicherbereiches liegt.
 */
int value32bit = memobj.get32(6);
```

Abbildung 3.10: Beispiel für einen direkten Speicherzugriff über Methoden an einem Speicherobjekt.

die automatische Speicherbereinigung der Domäne erniedrigt. Fällt der Zähler auf Null, wird der Speicherbereich wieder freigegeben.

Dynamische und statische Speicherobjekte unterscheiden sich aus der Sicht der Anwendung nicht. Sie können daher an System- und Bibliotheksschnittstellen gleichermaßen verwändet werden.

Gerätespeicher

Bei vielen Geräten werden Geräteregister einfach in den Speicherbereich des Rechners eingeblendet. So ist es möglich, die Geräte über die vorhandenen Schreib- und Lesebefehle der CPU zu steuern. Die Speicherobjekte können direkt auf diese gerätespezifischen Speicherbereiche gelegt werden, um so aus einer Domäne heraus die Geräteregister anzusprechen.

Im Zusammenspiel mit der Unterbrechungsbehandlung, die die OSEK/VDX-Schnittstelle bereitstellt, steht so die nötige Funktionalität zur Verfügung, um Gerätetreiber direkt in Java zu programmieren. Die Gerätetreiber müssen nicht zum vertrauenswürdigen Teil der Systemsoftware gezählt werden, sondern unterliegen dem Speicherschutz, den die JVM bereitstellt.

Abbilden von Klassen auf untypisierten Speicher

Außerdem ist es möglich, einzelne Felder von Klassen über die Speicherobjekte zu legen. So kann der Speicher durch eine Klasse nachträglich einen Typen erhalten. Der Gerätespeicher zum Ansteuern der Hardware besteht in der Regel aus unterschiedlichen Geräteregistern mit unterschiedlicher Bedeutung. Die Felder der Klasse können nun genau die Geräteregister repräsentieren, wodurch diese einen festen Bezeichner bekommen.

Bei der Abbildung der Klasse auf den Speicherbereich wird überprüft, ob die abgebildeten Felder in den Speicherbereich passen. Eine Bereichsüberprüfung beim jeweiligen Feldzugriff kann daher entfallen.

Das Vorgehen ist mit einem **struct** in C vergleichbar, mit dem eine feste Speicheradresse typisiert wird. Bei KESO werden jedoch nicht einfach die primitiven Datentypen auf den Speicher abgebildet, sondern es werden nur Felder mit speziellen Typen bei der Abbildung berücksichtigt. So wird verhindert, dass Felder, die echte Objektreferenzen enthalten, auf beliebige Speicherbereiche gelegt werden und die Typsicherheit der JVM ausgehebelt werden kann.

Die Abbildung 3.11 zeigt ein Beispiel für eine auf Gerätespeicher abgebildete Klasse. Nur die Felder mit den Typen **MT_U32** und **MT_SPACE32** werden auf den Gerätespeicher abgebildet. Die speziellen Typen werden in KESO als **MemoryTypes** oder Speichertypen bezeichnet. Es gibt eine ganze Reihe von unterschiedlichen Speichertypen, sowohl für unterschiedliche Bitweiten und Vorzeichen (z.B. einen Speichertyp für acht Bit ohne Vorzeichen **MT_U8**), als auch für unterschiedliche Verwendungszwecke (z.B. sechzehn Bit mit Vorzeichen und nur lesenden Zugriff **MT_S16RO**). Durch die Verwendung von Speichertypen für die abgebildeten Felder ist es möglich, dass die Klasse noch beliebige andere Felder enthalten kann. Diese Felder schaffen Platz für zusätzliche Informationen, wie z.B. Lese-/Schreibpuffer.

Bei der technischen Umsetzung wird wieder ein Stellvertreterobjekt verwendet. Das Stellvertreterobjekt enthält einen impliziten Zeiger auf den untypisierten Speicherbereich und die normalen Felder. Bei einem Feldzugriff auf einen Speichertyp findet ein Zugriff auf den Speicherbereich außerhalb der Domäne statt.

3.4.4 Synchrone Interdomänenkommunikation (Portale)

Die bis jetzt dargestellten Kommunikationdienste dienen in erster Linie dazu, mit der nativen Umgebung und der Hardware zu kommunizieren oder Speicherbereiche gemeinsam zu nutzen. Der Portal-Mechanismus, der nun beschrieben wird, bietet eine typsichere synchrone Kommunikation zwischen verschiedenen Schutzräumen.

```
package test;

import keso.core.*;

class TR_GPIO implements MemoryMappedObject {

        /* Ausgaberegister */
        MT_U32 gpio_out;

        /* Ausgabemodifikation */
        MT_U32 gpio_omr;

        MT_SPACE32 reserved_1;
        MT_SPACE32 reserved_2;

        /* E/A-Steuerregister 0,4,8 and 12 */
        MT_U32 gpio_iocr_0;
        MT_U32 gpio_iocr_4;
        MT_U32 gpio_iocr_8;
        MT_U32 gpio_iocr_12;

        MT_SPACE32 reserved_3;

        /* Input */
        MT_U32 gpio_in;

        /* andere Feldertypen und Methoden dürfen die
         * Klasse beliebig ergänzen. */
        int[] read_buffer new int[4];
        int pos = 0;

        static void test() {
                TR_GPIO gpio =
                        (TR_GPIO)MemoryService.mapStaticDeviceMemory(
                                        0xf0000c00,"test/TR_GPIO");
                gpio.gpio_out.set(5);
                read_buffer[pos] = gpio.gpio_in.get();
                pos = (pos + 1) % 4;
        }
}
```

Abbildung 3.11: Beispiel für die Abbildung einer Klasse auf den Gerätespeicher zur Steuerung der „General-Pupose-Input-Output“-Einheit (GPIO) des TriCore TC1796 Mikrocontrollers.

```
public void launch() {

        TickerService srv =
                (TickerService)PortalService.lookup("TickerService");

        for (int i=0;i<count;i++) { srv.roundtrip(); }
}
```

Abbildung 3.12: Namensdienstanfrage für ein Dienstobjekt

Eine Domäne, die einen angebotenen Dienst nutzen möchte, kann über einen Namensdienst eine Referenz auf ein Dienstobjekt bekommen. Die Abbildung 3.12 zeigt ein Beispiel für eine solche Anfrage. Wird der Dienst nicht innerhalb der gleichen Domäne angeboten, so wird ein Stellvertreterobjekt für den Dienst vom Namensdienst zurückgeliefert.

Das Stellvertreterobjekt stellt sicher, dass die Isolationseigenschaften der Domäne nicht verletzt werden. Bei der Kommunikation über Domängrenzen hinweg dürfen keine Speicheradressen die Domängrenzen passieren. Daher werden die Parameter bei einem Portalaufruf kopiert und nicht durch Referenzen übergeben.

Namensdienst

Der Namensdienst ermöglicht einen ortstransparenten Zugriff auf den angebotenen Dienst. Da die Dienstobjekte über den Namen angesprochen werden, ist es zum Zeitpunkt der Anwendungsentwicklung unerheblich, ob der Dienst von der gleichen Domäne, einer anderen Domäne auf dem gleichen Mikrocontroller oder einer Domäne auf einem anderen Mikrocontroller bereitgestellt wird.

In der Systemkonfiguration von KESO wird festgelegt, welcher Dienst von welcher Domäne exportiert bzw. importiert wird. Dienste können durch eine Änderung in der Konfiguration verlagert oder ausgetauscht werden, ohne dass eine Änderung im Programm erforderlich ist.

Die Abbildung 3.12 zeigt, wie ein Dienst in einer Java-Anwendung genutzt werden kann. Zuerst wird über den Namensdienst ein Stellvertreterobjekt für den Dienst angefordert. Anschließend kann das Dienstobjekt in der Anwendung verwendet werden.

Für eine ressourcensparende Implementierung des Namensdienstes muss der Name zum Übersetzungszeitpunkt bekannt sein. Der in Kapitel 5 beschriebene Übersetzer kann so die Stellvertreterobjekte statisch erzeugen und die Referenzen auf die Stellvertreterobjekte bereits zum Übersetzungszeitpunkt auflösen.

Parameterübergabe

Bei Java-Programmen werden die primitiven Parameter einer Methode immer „by-value“ übergeben und Objekte grundsätzlich als Referenzen weitergereicht. Objektreferenzen dürfen jedoch in KESO eine Domängrenze nicht überschreiten, daher werden beim Aufruf einer Portalmethode die Objekte kopiert und die Referenzen entsprechend angepasst. Dies gilt auch für Referenzen, die in den kopierten Objekten gespeichert sind. Es werden daher von den Parametern eines Portalaufrufs ausgehend alle erreichbaren Objekte kopiert.

Durch das Kopieren der Objekte wird zwar die Isolationseigenschaft der Domäne sichergestellt, jedoch kann dies zu einem erheblichen Ressourcenbedarf führen. Der Anwendungsentwickler muss sich daher über diese Eigenschaft von Portalaufrufen im Klaren sein, um eine schlanke Anwendung zu entwickeln. Eine einfache Lösung ist die ausschließliche Verwendung von primitiven Datentypen bei Portalaufrufen.

Da primitive Parameter jedoch nicht immer ausreichen, bietet KESO noch weitere Möglichkeiten. Ist ein Parameter selbst wieder als Portaltyp implementiert, so wird das Objekt als Dienst exportiert und in der Zieldomäne wieder ein Stellvertreterobjekt erzeugt. Die im Objekt gespeicherten Referenzen werden daher nicht kopiert.

Eine zweite Ausnahme bei Referenzen stellen die Speicherobjekte dar, die bereits in Kapitel 3.4.3 beschrieben wurden. Die von einem Speicherobjekt referenzierten Speicherbereiche liegen außerhalb der domäninternen Speicherverwaltung und können so als gemeinsam genutzter Speicher verwendet werden. Die statisch allozierten Stellvertreterobjekte existieren für die gesamte Betriebsdauer und können ebenfalls gemeinsam genutzt werden, nur bei der Verwendung von dynamischen Speicherobjekten wird in der Zieldomäne ein neues Speicherobjekt erzeugt.

Taskwechsel

Bei einem Portalaufruf werden nicht nur Nachrichten zwischen zwei Schutzräumen ausgetauscht, sondern es muss auch eine Aktivität in der Zieldomäne angestoßen werden. Ein Portalaufruf erfordert daher auch einen Taskwechsel oder einen vergleichbaren Mechanismus.

Ein echter OSEK/VDX-Taskwechsel erscheint jedoch nicht sinnvoll möglich. Ein OSEK/VDX-Taskwechsel bei jedem Portalaufruf würde bedeuten, dass entweder zur Laufzeit ein neuer Task erzeugt werden müsste, oder dass für jeden Dienst ein Task oder eine Gruppe von Tasks zur Bearbeitung der Anfrage bereitstehen müssen. In OSEK/VDX können keine Tasks zur Laufzeit erzeugt werden, und das Bereitstellen von einem Task pro Dienst weist einen zu hohen Bedarf an Speicherresourcen auf.

Ein weiteres Ausschlusskriterium ist eine Abhängigkeit zwischen allen Tasks, welche den gleichen Dienst nutzen. Da der Dienst-Task immer nur eine Anfrage bearbeiten kann, müssen die anderen Tasks warten. Damit das Warten auf den Dienst nicht gegen das Priority-Ceiling-Protocol verstößt, müsste der Dienst-Task folgerichtig immer mit einer Priorität größer oder gleich der höchsten Priorität aller Tasks laufen, die den gleichen Dienst nutzen. Die Abhängigkeit kann vermieden werden, indem für jeden Task, der eine Dienst-Methode aufruft, ein eigener Dienst-Task existiert. Dieser Ansatz würde jedoch eine Vielzahl an zusätzlichen OSEK/VDX-Tasks bedeuten und einen entsprechenden zusätzlichen Ressourcenbedarf erzeugen. Daher wurde in KESO ein anderer Ansatz umgesetzt.

Aufgrund des großen Ressourcenbedarfs findet bei einem Portalaufruf auf dem gleichen Mikrocontroller in KESO kein Taskwechsel statt. Für das Laufzeitsystem wird ein solcher Wechsel nur simuliert und der gleiche OSEK/VDX-Task wird in der Zieldomäne wiederverwendet. Dieses Vorgehen hat den Vorteil, dass kein extra Task für den jeweiligen Dienst benötigt wird und jeder Task den Dienst mit seiner eigenen Priorität durchläuft.

Das Überschreiten der Domängrenze durch einen Task kann die Isolationseigenschaft der Domäne zerstören. Daher ist es nötig, Maßnahmen zu ergreifen, welche die Isolation aufrecht erhalten. Anstelle eines echten Taskwechsels werden die Domäne und das Taskobjekt, welche einem Task zugeordnet sind, gewechselt. Alle Operationen finden nach dem Überschreiten der Domängrenze im Kontext der Dienstdomäne statt. Aus Sicht der Anwendung findet daher ein Taskwechsel statt, auch wenn es sich weiterhin um den gleichen OSEK/VDX-Task handelt.

Ein weiterer Punkt ist die Trennung der Methodenstacks. Die auf dem Stack gespeicherten Objektreferenzen sind immer nur in der jeweiligen Domäne gültig. Dies gefährdet die Isolation der Java-Anwendungen zunächst einmal nicht, da in einer Java-Methode immer nur auf den Stackbereich und die Parameter der eigenen Methode zugegriffen werden kann und die Parameter bereits in die Dienstdomäne kopiert wurden. Die automatische Speicherbereinigung einer Domäne würde jedoch alle Bereiche des Stacks untersuchen. Für die Speicherbereinigung muss daher erkennbar sein, zu welcher Domäne eine Referenz auf dem Stack gehört. Der Stack wird daher beim Überschreiten der Domängrenze durch das Einfügen von Domänbeschreibungen in Partitionen unterteilt. Jede Speicherbereinigung untersucht nur die Partitionen, die ihrer Domäne zugeordnet sind.

3.5 Verfahren zur automatischen Speicherverwaltung

Die Verwaltung von dynamischem Speicher ist in großen Softwareprojekten ein nicht zu unterschätzendes Problem. Vergisst ein Programmierer, den dynamisch angeforderten

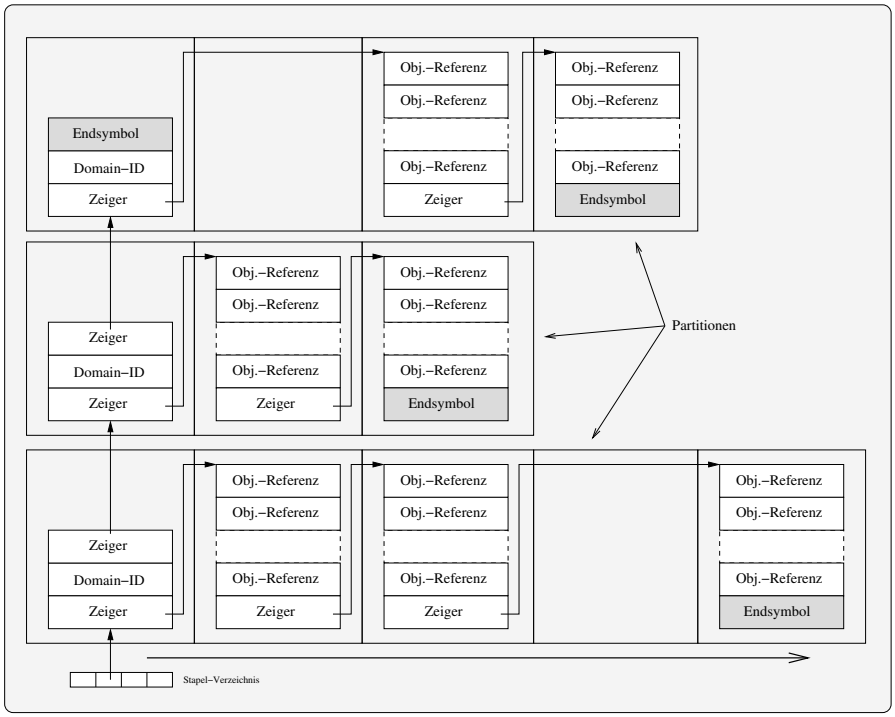

Abbildung 3.13: Leichtgewichtiger Taskwechsel

Speicher nach der Nutzung wieder freizugeben, so entstehen so genannte „Speicherlöcher“, das sind Speicherbereiche, die vom System nicht wiederverwendet werden. Wenn Speicherlöcher zur Laufzeit wiederholt entstehen, können diese nach und nach den gesamten Arbeitsspeicher belegen.

Speicherlöcher werden in einfachen Softwaretests oft nicht zuverlässig erkannt, da sie in selten durchlaufenen Programmabschnitten, wie zum Beispiel in Fehler- und Ausnahmebehandlungen, mit hoher Wahrscheinlichkeit erst nach Wochen oder Monaten zu einem Fehlverhalten der Software führen.

Weitere Programmierfehler sind die vorzeitige oder wiederholte Freigabe von Speicher, welcher jedoch weiterhin in Benutzung ist. Diese Fehler werden auch als „hängende Zeiger“ bezeichnet. Die fehlerhaft freigegebenen Speicherbereiche werden mit hoher Wahrscheinlichkeit zu einem späteren Zeitpunkt mehrfach genutzt. Die Folge sind Datenverlust und ein nicht vorhersagbares Verhalten der Anwendung. Auch diese Fehler führen in vielen Fällen nicht unmittelbar zu einem Fehlverhalten, so dass auch hier nicht alle diese Fehler bei einem Softwaretest auftreten.

Mit einer automatischen Speicherbereinigung gibt es Fehler dieser Art nicht. Jede JVM benötigt eine automatische Speicherbereinigung für alle dynamisch angeforderten Objekte, da keine explizite Freigabe von nicht mehr genutzten Objekten vorgesehen ist. Bei einer automatischen Speicherbereinigung wird der Speicher für die Objekte so lange als belegt betrachtet, bis das Objekt nicht mehr aus dem Programm heraus erreichbar ist. Es können daher keine hängenden Zeiger entstehen, und auch der nicht mehr erreichbare Speicher wird erkannt, was diese Art von Speicherlöchern verhindert.

Die automatische Speicherbereinigung hat jedoch nicht nur Vorteile. Besonders im angestrebten Einsatzgebiet von KESO wird die automatische Speicherbereinigung als Hinderungsgrund für den Einsatz von Java explizit genannt. Dafür gibt es gute Gründe: Zum einen besitzen die Mikrocontroller oft sehr wenig Speicher, weniger als ein Kilobyte, was eine dynamische Speicherverwaltung im Allgemeinen und erst recht eine automatische Speicherbereinigung als ungeeignet erscheinen lässt, da diese zusätzliche Datenstrukturen und damit Speicherplatz benötigen. Zum anderen erschwert eine dynamische Speicherverwaltung die Vorhersagbarkeit vom zeitlichen Ablauf der Anwendung, was ein Problem für die Entwicklung von echtzeitfähigen Anwendungen darstellt.

Bei wirklich kleinen Mikrocontrollern wie den ATMega 8535 mit einem Arbeitsspeicher von 520 Byte wird daher oft auf eine dynamische Speicherverwaltung und Zeigerarithmetik verzichtet. Ohne dynamischer Speicherverwaltung tritt das Problem von Speicherlöchern und verwaisten Zeigern nicht auf.

Der Funktionsumfang und die Größe der Mikrocontroller bei den Steuergeräten nehmen jedoch immer weiter zu. Der TriCore Mikrocontroller kann bereits einen Arbeitsspeicher von mehreren Megabytes verwalten. Der Einsatz einer dynamischen Speicherverwaltung

erscheint auf diesem Mikrocontroller problemlos machbar und je nach Anforderungen der Anwendung auch sinnvoll. Die in dieser Arbeit implementierten Verfahren für eine automatische Speicherverwaltung benötigen im Vergleich zu einer einfachen dynamischen Speicherverwaltung kaum zusätzliche Speicherressourcen, da sie zur Bestimmung der belegten Speicherbereiche die Typinformationen verwendet, welche bereits für die Typsicherheit benötigt werden.

Insgesamt wurden für KESO drei unterschiedliche Verfahren zur Speicherbereinigung entwickelt, die in den nachfolgenden Kapiteln genauer beschrieben werden. In Bezug auf eine echtzeitfähige Speicherbereinigung und Java soll jedoch zuerst die *Real-Time Java Spezification* (RTJS) diskutiertet werden, da diese sich auch mit einer echtzeitfähigen Speicherbereinigung für Java beschäftigt und eines der Verfahren für KESO mit dieser vergleichbar ist.

3.5.1 RTJS vs. Echtzeitfähiger Speicherbereinigung

Die zeitlichen Schranken, die ein Echtzeitsystem einhalten muss, können unter Umständen selbst von einer sehr feingranular unterbrechbaren Speicherbereinigung nicht eingehalten werden. Die Java-Erweiterung für Echtzeitsysteme (JSR-1 [56]) führt daher eine Reihe von zusätzlichen Konzepten ein, die dem Anwendungsentwickler mehr Gestaltungsmöglichkeiten beim Anfordern und Freigeben von Speicher bieten soll.

Eine dynamische Verwaltung von Speicher mit konstanter Laufzeit bietet zum Beispiel die in der Spezifikation als **ScopedMemory** bezeichnete Schnittstelle. Bei dem **ScopedMemory** ist die Lebensdauer von Objekten an das Eintreten und Verlassen eines Sichtbarkeitsbereiches durch einen Programmfaden geknüpft.

Dazu wird zuerst ein **ScopedMemory**-Objekt mit fester Speichergröße erzeugt. Über einen Methodenaufruf kann nun ein Programmfaden dieses Objekt betreten. Speicher für alle anschließend vom Programmfaden allozierten Objekte stammen anschließend aus dem für das **ScopedMemory**-Objekt vorab allozierten Speicherbereich. Der Speicherbereich wird erst wieder freigegeben, wenn die Eintrittsmethode wieder verlassen wird. Die Objekte im **ScopedMemory** werden nicht von der automatischen Speicherbereinigung freigegeben.

Das **ScopedMemory**-Konzept ist vergleichbar mit dem Anlegen von Objekten auf dem Methodenstack, wie es in anderen Sprachen, wie zum Beispiel in C++ möglich ist. Diese Objekte existieren nur für die Dauer, in der sich der Programmfaden in der Methode befindet. Im Vergleich dazu existieren die Objekte beim **ScopedMemory** so lange wie der Programmfaden im Geltungsbereich vom **ScopedMemory**-Objekt verweilt.

Die Spezifikation schränkt die Verwendung von Objekten, die im Geltungsbereich von einem **ScopedMemory**-Objekt angelegt wurden, stark ein. Objektreferenzen auf entsprechende Objekte dürfen nicht in Objekten abgelegt werden, die in einem anderen

Speichertyp angelegt wurden und sie dürfen ausschließlich im eigenen **ScopedMemory**-Bereich oder in einem in diesem **ScopedMemory**-Bereich angelegten anderen **ScopedMemory** abgelegt werden. Diese Einschränkung verhindert, dass eine Referenz den Geltungsbereich verlässt und die Freigabe des Speichers zu einem hängenden Zeiger führen würde.

Diese Einschränkungen werden von der Laufzeitumgebung überwacht. Eine JVM, die den **ScopedMemory** implementiert, überwacht hierzu das Lesen und Schreiben von Feldern durch sogenannte Lese- und Schreibbarrieren und verhindert so, dass Referenzen außerhalb des Geltungsbereiches genutzt bzw. geschrieben werden.

Ein großer Vorteil dieses Konzeptes ist, dass das Anlegen neuer Objekte und die Freigabe des gesamten Speicherbereichs jeweils sehr effizient und in konstanter Zeit durchführbar ist. Zur Verwaltung des freien Speichers im **ScopedMemory** ist nur ein Zeiger auf den noch freien Rest nötig, welcher bei jeder Speicherbelegung einfach erhöht wird. Beim Verlassen vom **ScopedMemory** wird der Speicherbereich wieder komplett zurückgesetzt.

Das **ScopedMemory**-Konzept hat jedoch auch wesentliche Nachteile. Wenn Daten außerhalb eines **ScopedMemory**-Bereiches weiter verwendet werden sollen, müssen diese den Geltungsbereich durch eine Kopieroperation explizit verlassen. Für diesen Vorgang sieht die RTJS eigene Methoden zum Übertragen der Daten vor.

Um die Daten möglichst selten aus dem Geltungsbereich heraus kopieren zu müssen, werden die **ScopedMemory**-Bereiche möglichst groß gewählt. Dieses führt jedoch zu einem erhöhten Bedarf an Arbeitsspeicher, da die Objekte erst beim Verlassen des Geltungsbereiches freigegeben werden und dies bei einem großen Geltungsbereich zu einem späteren Zeitpunkt geschieht. Pizlo et al. [80] verweist auf diese Problematik beim Vergleich einer echtzeitfähigen Speicherbereinigung mit dem **ScopedMemory**-Konzept.

Auch die Bibliotheks-Klassen, die in einem **ScopedMemory**-Bereich genutzt werden, müssen die durch die Spezifikation entstehenden Einschränkungen berücksichtigen und dürfen keine Objektreferenzen in statische Klassenvariablen speichern. Viele Klassen aus der Standard-Bibliothek von Java sind darauf nicht vorbereitet und können daher innerhalb eines **ScopedMemory**-Bereiches nicht verwendet werden.

Wenn es möglich ist, mit einer automatischen Speicherbereinigung die geforderten Zeitschranken einzuhalten, so ist diese dem **ScopedMemory**-Konzept gegenüber vorzuziehen. Abgesehen von den vielen Randbedingungen, die der Programmierer beachten muss, verschlechtern die zusätzlichen nötigen Überprüfungen der globalen Schreib- und Leseoperation das zeitliche Verhalten der Anwendungen und vergrößern das Maschinenprogramm. Auf eine echtzeitfähige automatische Speicherbereinigung kann in einer RTJS-spezifischen JVM auch trotz **ScopedMemory** nicht vollständig verzichtet werden.

```
Heap = RestrictedDomainScope {
        HeapSize=1024;
        Mode="Reset";
}
```

Abbildung 3.14: Konfigurationsbeispiel für die minimale Speicherverwaltung

3.5.2 Minimale Speicherbereinigung (RestrictedDomainScope)

Die minimale Speicherbereinigung greift die Idee vom **ScopedMemory** auf und verwendet als Geltungsbereich implizit den Geltungsbereich der Domäne. Alle Objekte, die innerhalb dieser Domäne erzeugt werden, belegen so lange Speicher, bis alle Programmfäden die Domäne verlassen. Zum Verlassen der Domäne muss der Programmfaden terminieren. Bei einer Sonderform dieser Strategie kann der Vorgang von einem der Programmfäden begünstigt werden, indem über einen Systemdienst weitere Task-Aktivierungen verzögert werden.

In der Konfiguration wird diese Speicherbereinigung als **RestrictedDomainScope** bezeichnet. Die Durchführung der Speicherbereinigung kann auch als ein Zurücksetzen der Domäne mit einem anschließenden Neustart angesehen werden. Nach dem Terminieren aller Programmfäden gibt es keine lebenden Objekte mehr, die von lokalen Variablen aus erreichbar sein könnten. Nur die globalen Variablen können noch Objektreferenzen enthalten. Um den Implementierungsaufwand für die Speicherverwaltung minimal zu halten, werden auch diese Referenzen gelöscht. Damit werden alle Objekte unerreichbar und der gesamte Speicher kann anschließend wiederverwendet werden. Die Initialisierungsroutinen der Klassen werden erneut aufgerufen, um die globalen Variablen in einen einheitlichen Zustand zu versetzen, und somit ist der Neustart der Domäne vollzogen.

Wie beim **ScopedMemory** muss der Anwendungsentwickler darauf achten, dass die Speicherbereinigung läuft, bevor der freie Speicher erschöpft ist. Da dazu alle Tasks innerhalb der Domäne terminieren müssen, ist es sinnvoll, immer nur einen Task einer Domäne zuzuordnen. Es ist auch möglich, die Speicherbereinigung komplett zu vermeiden. In diesem Fall darf die Anwendung nur eine begrenzte Anzahl von Objekten während einer Initialisierungsphase anlegen.

Wird ein periodischer Neustart der Domäne zum Aufräumen des Speichers genutzt, so sind zusätzliche Maßnahmen nötig, um Daten, die nach einem Neustart noch vorhanden sein sollen, zu sichern. Die Daten können zu diesem Zweck über die in Kapitel 3.4.4 beschriebene Interdomänenkommunikation in einem Speicherbereich außerhalb der Domäne abgelegt werden.

In Hinblick auf den Ressourcenbedarf ist die minimale Speicherverwaltung optimal für kleinste eingebettete Systeme geeignet. Der Aufwand zum Suchen von erreichbaren

```
Heap = CoffeeBreak {
        HeapSize=2048;
        SlotSize=8;
        Group="Default";
}
```

Abbildung 3.15: Konfiguration der durchsatzstarken Speicherbereinigung

Objekten entfällt, da keines der Objekte nach der Speicherbereinigung noch erreichbar ist. Es wird keine Freispeicherliste benötigt, da der freie Speicher immer aus einem zusammenhängenden Bereich besteht. Sowohl die Belegung von Speicher als auch die Bereinigung des Speichers sind in konstanter Zeit möglich. Daher ist dieses Verfahren auch für Echtzeitanwendungen ideal geeignet.

Außerdem sind im Unterschied zum **ScopedMemory** keine zusätzlichen Maßnahmen nötig, um die Gültigkeit von Objektreferenzen bei Zuweisungen zu überprüfen. Alle Programmfäden, die eine Objektreferenz erreichen können, laufen in der gleichen Domäne und damit im gleichen Geltungsbereich. Nur Daten, die an eine andere Domäne weitergereicht werden, verlassen den Geltungsbereich. Diese werden jedoch durch den Portalaufruf kopiert und nicht mehr referenziert.

Die Nachteile bestehen darin, dass die Anwendung entweder auf eine fortlaufende Anforderung von Speicher verzichtet oder später benötigte Zustandsdaten gesondert ablegen muss. Durch eine Aufgliederung der Anwendungen auf mehrere Domänen ist es möglich, die minimale Speicherbereinigung mit einer vollwertigen Speicherbereinigung zu kombinieren und so die Vorteile aus beiden Welten zu nutzen.

3.5.3 Durchsatzstarke Speicherbereinigung (CoffeeBreak)

Neben der minimalen Speicherbereinigung wurden noch zwei weitere Verfahren für KESO entwickelt, welche eine vollwertige automatische Speicherbereinigung durchführen. Beim nachfolgenden Verfahren werden für die Dauer der Speicherbereinigung alle Aktivitäten innerhalb der Domäne angehalten. Auf Grund dieser entstehenden Pause wurde das Verfahren in der KESO-Konfiguration als **CoffeeBreak** bezeichnet.

Auffinden von Objektreferenzen

Die lebenden Objekte werden durch ihre Erreichbarkeit ermittelt. Hierzu markiert der Algorithmus, in einer ersten Phase, von einer Wurzelmenge ausgehend alle Objekte, die erreichbar sind. In einer zweiten Phase werden alle nicht markierten Objekte freigegeben. Die Wurzelmenge für die Analyse bilden die in den lokalen Variablen der

Programmfäden und die in statischen Feldern gespeicherten Objektreferenzen. Die statischen Felder werden zu diesem Zweck in einer Tabelle gespeichert und können so schnell und zuverlässig von der Speicherbereinigung untersucht werden.

Bei lokalen Objektreferenzen ist dies nicht ganz so einfach. Die Objektreferenzen können sich zum Zeitpunkt der Speicherbereinigung auch auf dem Methodenstack oder in den CPU-Registern befinden. Da das Maschinenprogramm bei KESO vom C-Compiler erzeugt wird, besitzt die Speicherbereinigung kein gesichertes Wissen über den Aufbau des Methodenstacks und die Zuordnung der CPU-Register und kann so eine Objektreferenz nicht von einer Ganzzahl unterscheiden.

Automatische Speicherbereinigungen, die dieses Wissen nicht besitzen, umgehen für gewöhnlich dieses Problem, indem sie alle Werte, die in Registern und im Speicherbereich des Methodenstacks vorkommen und vom Wertebereich her eine Referenz auf ein Objekt darstellen könnten, als Objektreferenz interpretieren [24]. Durch dieses konservative Vorgehen bleiben Objekte möglicherweise länger bestehen als nötig, es kann jedoch nicht passieren, dass ein Objekt vorzeitig freigegeben wird.

Bei diesem Ansatz ist es nicht möglich, eine sichere Aussage über die Freigabe von ungenutzten Objekten treffen zu können, da auf Grund eines Zahlenwertes, der zufälligerweise für eine Objektreferenz gehalten wird, die Lebensdauer von Objekten beliebig verlängert werden kann. Für den Einsatz auf einem Gerät mit eingeschränkten Ressourcen hat ein spontaner Anstieg der Speicherbelegung jedoch schnell fatale Folgen für die Anwendung.

Für KESO wurde daher ein präzises Verfahren entwickelt, welches die Referenzen auf dem Stack von primitiven Daten unterscheiden kann. Dafür ist es nötig zur Laufzeit zusätzlich Informationen über den Stackinhalt zu besitzen. Bekannte Verfahren beziehen die zusätzlich benötigte Information meistens aus Bitfeldern, die im Weiteren als Stackmaps bezeichnet werden. In diesen Stackmaps wird für jede Stackposition im Methodenframe ein Bit reserviert, welches für den Fall, dass diese Position eine Referenz enthält, gesetzt ist. Die Stackmaps werden entweder dynamisch zur Laufzeit erstellt und bei jeder Stackoperation angepasst oder statisch zum Übersetzungszeitpunkt berechnet [76].

Bei einer dynamischen Stackmap kommt es zu einem nicht unerheblichen Mehraufwand zur Laufzeit, da beim Schreiben auf die Speicherstelle auch die Stackmap aktualisiert werden muss. Bei der statischen Variante werden keine zusätzlichen Operationen zum Aktualisieren der Stackmap zur Laufzeit benötigt, jedoch entsteht ein zusätzlicher Speicherbedarf, und die zur jeweiligen Position im Programm passende Stackmap muss durch die Speicherbereinigung bestimmt werden.

Um den Speicherbedarf für die statischen Stackmaps in einem vertretbaren Rahmen zu halten, werden diese immer nur für ausgewählte Positionen im Programmablauf

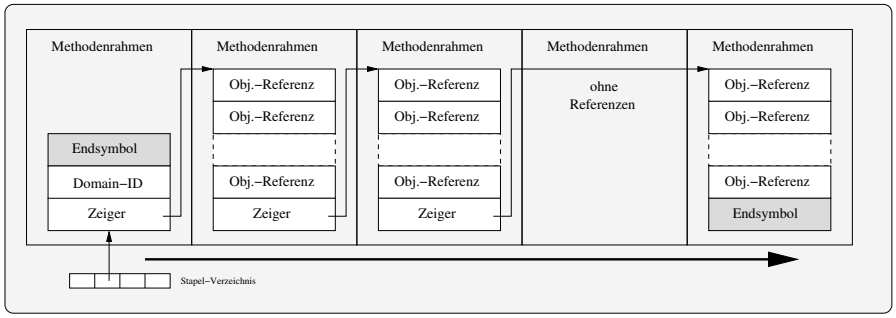

Abbildung 3.16: Auffinden von lokalen Objektreferenzen

berechnet. Ausreichend sind die Stellen im Programm, an denen Funktionsaufrufe oder rückwärtsgerichtete Sprünge, wie sie bei Programmschleifen vorkommen, stattfinden. Für den Fall, dass ein Faden zum Zeitpunkt der automatischen Speicherbereinigung nicht an einer Position mit Stackmap steht, wird dieser Faden bis zur nächsten gültigen Position vorgefahren. Dies setzt eine Unterstützung durch das Betriebssystem voraus, eine Funktionalität, die ein OSEK/VDX-System jedoch nicht bietet.

Beide Verfahren benötigen ein genaues Wissen über den tatsächlichen Stackaufbau und zusätzlichen Speicherplatz, um die Bitfelder zu speichern. Der tatsächliche Stackaufbau ist stark vom verwendeten Mikrocontroller und vom verwendeten C-Compiler abhängig. Für KESO wurde daher ein Verfahren entwickelt, welches keine Stackmap benötigt und mit jedem ANSI C-Compiler und auf beliebigen Mikrocontrollern funktionieren sollte.

Beim Übersetzungsvorgang wird sichergestellt, dass für jeden primitiven Datentyp unterschiedliche lokale Variablen eingeführt werden. Eine lokale Variable für eine Objektreferenz kann daher nie einen arithmetischen Zahlenwert enthalten. Der Abbildungsvorgang ist in Kapitel 5.4.1 genauer beschrieben. Zusätzlich werden die Variablen für die Objektreferenzen im Gegensatz zu den Variablen mit arithmetischen Werten in einer lokalen Tabelle gehalten und nicht in einzelnen Variablen. Beim Eintritt in die Methode wird diese Tabelle mit leeren Einträgen initialisiert, um den früheren Speicherinhalt zu löschen. Die Tabelle wird anschließend in der Tabelle vom Aufrufer registriert. So entsteht eine Liste von Referenztabellen, wie sie in Abbildung 3.16 dargestellt ist. Bei Methoden, die keine lokalen Objektreferenzen besitzen, wird keine Referenztabelle angelegt und die Methode wird beim Einhängen in die Liste übersprungen. Vor einem möglichen Lauf der Speicherbereinigung wird die Liste durch die Eintragung eines Endsymbols in der letzten Tabelle abgeschlossen.

Die automatische Speicherbereinigung kann nun über die Referenztabellen die Objektreferenzen präzise und schnell finden. Durch das Initialisieren und Verketten der Tabellen entsteht bei diesem Verfahren ein zusätzlicher Rechenaufwand. Im Gegensatz zu einer

vollständig dynamischen Stackmap entsteht dieser Aufwand jedoch nur beim Betreten einer Methode und nicht beim wesentlich häufiger auftretenden Verändern einer lokalen Variable.

Durch die verketteten Tabellen wird der C-Compiler gezwungen, die lokalen Variablen für Objektreferenzen zusammenhängend anzuordnen. Dies geschieht statisch zum Übersetzungszeitpunkt. Ein zusätzlicher Speicherbedarf entsteht durch die Zeiger zum Verketten der Tabellen und aufgrund schlechterer Optimierungsmöglichkeiten durch den C-Compiler, da dieser gezwungen ist, die lokalen Variablen auf den Methodenstack abzulegen, und sie nicht wie sonst üblich ausschließlich in Registern halten kann.

Wie bei einer statischen Stackmap steht die Information aus den Referenztabellen nicht zu jedem beliebigen Zeitpunkt zur Verfügung, da die Tabellen nicht zu jedem beliebigen Zeitpunkt mit den CPU-Registern in einem konsistenten Zustand sind. Die automatische Speicherbereinigung kann daher nur zu bestimmten Zeitpunkten laufen.

In KESO wird dieser Zeitpunkt durch die Priorität der Speicherbereinigung geregelt. Läuft die Speicherbereinigung mit der niedrigsten Priorität in der jeweiligen Domäne und damit nur dann, wenn sich alle anderen Tasks im Zustand „**Suspendiert**“ oder „**Wartend**“ befinden, so sind die Referenztabellen in einem konsistenten Zustand. Ist ein Task im Zustand „**Suspendiert**“, so ist die Referenztabelle leer. Ist der Task im Zustand „**Wartend**“, so war seine letzte Aktion der Aufruf der Methode **WaitEvent** aus der KESO-API. Da der C-Compiler davon ausgehen muss, dass die Tabelle durch die aufgerufene Methode verändert wird, werden alle Lese- und Schreibvorgänge vor dem Aufruf abgeschlossen und die Tabelle ist in einem konsistenten Zustand.

Dies hat zur Folge, dass nur **extended tasks** in KESO eine Referenztabelle benötigen und auch nur die Methoden, die zu einem Aufruf von **WaitEvent** führen können. Um den Speicherbedarf zu minimieren, werden die Referenztabellen nur für diese blockierenden Methoden aufgebaut. Blockierende Methoden sind in diesem Zusammenhang die Methoden, die entweder eine andere blockierende Methode oder die Methode **WaitEvent** aufrufen.

Das Verfahren hat den Vorteil, dass es von der verwendeten Mikrocontroller-Architektur und dem C-Compiler unabhängig ist. Ein weiterer Vorteil liegt in der zusammenhängenden Anordnung der Referenzen auf dem Methodenstack. Die Speicherbereinigung kann durch die zusammenhängende Lage die Referenzen schneller als bei einer statischen Stackmap finden. Der dritte Vorteil ergibt sich aus dem Zeitpunkt der Speicherbereinigung. Sie wird gestartet, wenn es keine anderen Aufgaben gibt und zu einem Zeitpunkt, an dem sich die meisten Methoden im Zustand „**Suspendiert**“ befinden und daher überhaupt keine lokalen Variablen besitzen und, im Vergleich zu einem anderen Moment, die meisten Objekte wahrscheinlich nicht erreichbar sind.

Es bleiben noch die Objektreferenzen, die in den Registern der CPU gehalten werden, und die Funktionsparameter. Beide müssen nicht untersucht werden, solange sichergestellt

ist, dass jede Objektreferenz, die dort vorkommen kann, auch über die Referenztabellen gefunden wird. Dies ist der Fall, da alle Referenzen, die von der Speicherverwaltung vergeben werden, vor ihrer Verwendung zuerst in der Referenztabelle gespeichert werden. Durch die Priorität der Speicherbereinigung wird sichergestellt, dass zum Aktivierungszeitpunkt der Speicherbereinigung kein anderer Task mit höherer Priorität lauffähig ist. Alle Tasks befinden sich zum Aktivierungszeitpunkt an einem Punkt, an dem sich die Referenztabellen in einem konsistenten Zustand befinden und die Register gesichert sind.

Aufbau der Freispeicherliste

Zum Markieren der lebenden Objekte wird ein Bitfeld verwendet. Hierfür ist der Speicher in gleich große Speicherslots unterteilt. Für jeden Slot gibt es ein Bit im Bitfeld. Das Bitfeld wird vor der Markierungsphase gelöscht und für die erreichbaren Objekte werden die belegten Slots im Bitfeld markiert. Nach der Markierung aller erreichbaren Objekte werden die nicht als belegt markierten Slots gelöscht. Die freien Bereiche werden im gleichen Durchlauf zu einer neuen Freispeicherliste verkettet.

Um Speicherplatz zu sparen, kann das Bitfeld von der Speicherbereinigung für andere Domänen wiederverwendet werden. Die Verwendung eines Bitfeldes hat den Vorteil, dass die Suche nach freien Bereichen im Bitfeld stattfindet und daher schneller abläuft. Darüber hinaus ist kein zusätzlicher Aufwand nötig um die freien Bereiche zu verschmelzen. Auch eine Verzeigerung oder zusätzliche Markierung in den lebenden Objekten kann so entfallen.

3.5.4 Echtzeitfähige Speicherbereinigung (IRR)

An eingebettete Systeme wird meist die Anforderung gestellt, feste Zeitschranken bei der Reaktion auf ein Ereignis einhalten zu müssen. Daher ist es erforderlich, dass die Speicherbereinigung die Reaktionszeit der Anwendung nicht über Gebühr verzögert. Die dritte automatische Speicherbereinigung, die für KESO im Rahmen dieser Arbeit implementiert wurde, ist auf diese Anforderung ausgerichtet.

Wie bei der durchsatzstarken Speicherverwaltung geschieht die Speicherbereinigung wieder in einem eigenen Task mit einer Priorität die kleiner ist als alle anderen Task-Prioritäten in der Domäne. Das bedeutet, dass die Speicherbereinigung nur arbeitet, wenn in der Domäne keine anderen Aufgaben zu erfüllen sind.

Aus Sicht der Echtzeitanforderungen ist dies der günstigste Zeitpunkt für eine Speicherbereinigung, da so kein Task der Anwendung durch die Speicherbereinigung verzögert wird. Die benötigte Rechenzeit für die Speicherfreigabe, die bei einer expliziten Freigabe von Speicher im Task der Anwendung während der Aktiven-Phase geschieht, wird so in die Inaktiven-Phase der Anwendung verschoben.

```
Heap = IdleRoundRobin {
        HeapSize=2048;
        SlotSize=16;
        Group="Default";
}
```

Abbildung 3.17: Konfiguration der durchsatzstarken Speicherbereinigung

Ist die Speicherbereinigung angelaufen, kann sie, im Gegensatz zu einer Konfiguration mit der **CoffeeBreak**-Speicherbereinigung, feingranular von einem höher-prioren Task wieder unterbrochen werden. Um dies zu ermöglichen, sind natürlich zusätzliche Maßnahmen nötig, damit die Speicherbereinigung eine konsistente Sicht auf die lebenden Objekte behält.

Bei der Implementierung wurde ein Algorithmus gewählt, bei dem alle Objekte, die zum Startzeitpunkt der Speicherbereinigung nicht erreichbar waren, freigegeben werden. Nach diesem Zeitpunkt frei gewordene oder allozierte Objekte werden erst beim nächsten Durchlauf berücksichtigt.

Zum Fixieren des Startzeitpunktes dient ein ein Bit großer Zähler in jedem Objekt. Dieser Zähler teilt die Objekte in zwei unterschiedliche Entstehungsepochen ein. Beim Start der Speicherbereinigung wird der Epochenzähler erhöht, und nur die Objekte aus der letzten Epoche werden untersucht. Die Menge der lebenden Objekte in der alten Epoche nimmt bei jedem Schritt der Speicherbereinigung ab. So ist ein Fortschritt der Speicherbereinigung garantiert. Die Zeit, die für die Untersuchung der Objekte in der Markierungsphase benötigt wird, hängt nur von der Anzahl der lebenden Objekte ab und nicht von der Größe des Speichers.

Das Auffinden von lebenden Objekten und die Markierung von belegten Speicherbereichen geschieht mit Hilfe eines Bitfeldes und ist vergleichbar zu dem in Kapitel 3.5.3 beschriebenen Vorgehen. Es wird zuerst die Wurzelmenge auf erreichbare Objekte hin untersucht; dabei werden für die lokalen Variablen die bereits in Kapitel 3.5.3 beschriebenen Referenztabellen benutzt. Die gefundenen Objektreferenzen werden auf einem Arbeitsstapel abgelegt und die Objekte durch die Erhöhung des Epochenzählers der neuen Epoche zugeordnet.

Nach dem Untersuchen der Wurzelmenge wird jeweils ein Objekt vom Arbeitsstapel genommen und auf Referenzen hin untersucht. Die Referenzen, die in die Vergangenheit zeigen, werden für die anschließende Untersuchung erneut auf dem Arbeitsstapel abgelegt und die Objekte durch das Anpassen des Epochenzählers ebenfalls der neuen Epoche zugeordnet.

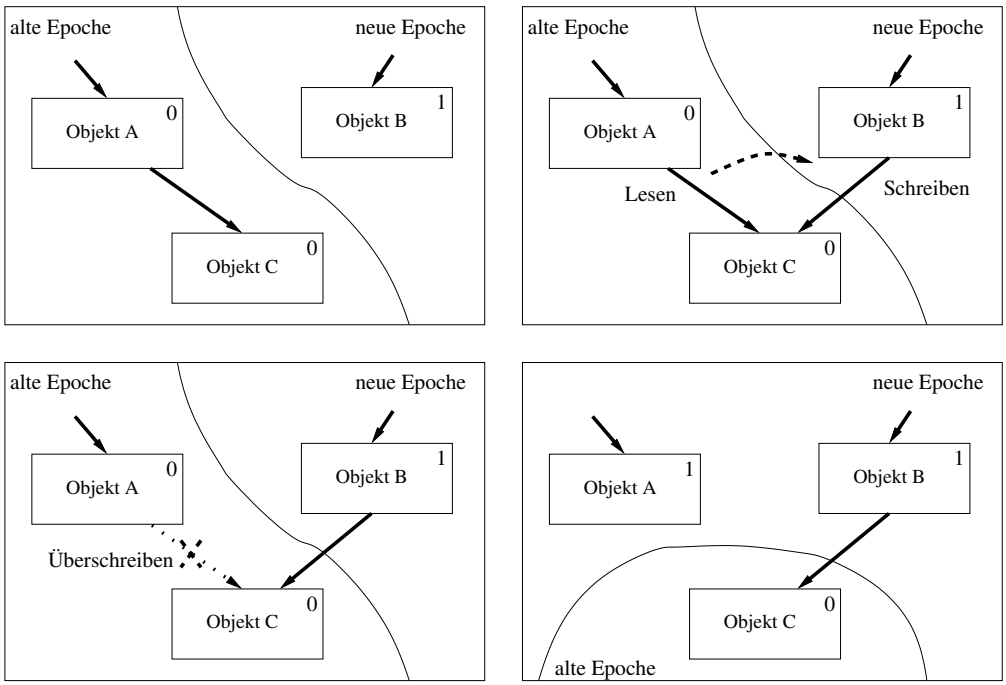

Abbildung 3.18: Wird während der Speicherbereinigung die Verzeigerung der Objekte verändert, so können Referenzen entstehen, die auf vermeintlich nicht mehr benötigte Objekte zeigen.

Schreibbarrieren zur Synchronisation

Durch die nebenläufig arbeitende Anwendung kann es passieren, dass der Objektgraph so verändert wird, dass lebende Objekte von der Speicherbereinigung übersehen werden. Kritisch ist eine Veränderung, wenn, wie in Abbildung 3.18 dargestellt, eine Objektreferenz zuerst von der alten in die neue Epoche kopiert wird und anschließend die letzte Referenz auf dieses Objekt in der alten Epoche überschrieben wird. Dieses Objekt wäre durch die neue Referenz nach dem Durchlauf weiterhin erreichbar, würde aber nicht untersucht, da die Objekte, die vollständig in der neuen Epoche sind, nicht mehr untersucht werden.

Es gibt nun verschiedene Ansätze, dieses Problem zu lösen. Die Speicherverwaltung kann beim „Lesen“, beim „Schreiben“ oder beim „Überschreiben“ [26, 104, 106] einer Objektreferenz informiert werden. Dies geschieht durch Lese- oder Schreibbarrieren. Die umgesetzte Speicherbereinigung verwendet Schreibbarrieren und überwacht damit das Überschreiben von Objektreferenzen. Wird eine Objektreferenz auf ein Objekt in der alten Epoche überschrieben, so legt die Anwendung diese Referenz auf den Arbeitsstapel der Speicherbereinigung und ordnet das Objekt der neuen Epoche zu. Dieses Vorgehen

entspricht einem weiteren Fortschritt in der Speicherbereinigung und gefährdet daher nicht die Fortschrittsgarantie.

Die Referenztabellen werden von der Speicherbereinigung am Anfang der Markierungsphase atomar auf den Arbeitsstapel kopiert. Durch dieses Vorgehen kann auf die Schreibbarrieren beim Schreiben von lokalen Variablen verzichtet werden.

Freispeicherliste

Bei der **IRR**-Variante wird für den Aufbau der Freispeicherliste wieder ein Bitfeld verwendet. Dieses Bitfeld wird vor dem Start der Speicherbereinigung gelöscht, und anschließend werden alle Bereiche in der noch existierenden Freispeicherliste im Bitfeld markiert. Die während der Speicherbereinigung allozierten Objekte sind daher vor der Löschung geschützt.

Während der Markierungsphase wird auch das Bitfeld aktualisiert. Die anschließend noch freien Bereiche werden gelöscht und in die aktuelle Freispeicherliste eingeordnet. Das Einhängen in die Freispeicherliste muss dabei mit dem Anfordern von neuen Objekten durch die Anwendung synchronisiert werden. Es kann passieren, dass Listenelemente erst beim Einordnen verschmelzen. Die **IRR**-Speicherverwaltung ist daher wesentlich umfangreicher und benötigt für die gleiche Anzahl von lebenden Objekten mehr Rechenzeit als die **CoffeeBreak**-Variante.

Es gibt drei Aktionen, die zwischen der Speicherbereinigung und der Anwendung synchronisiert werden müssen. Die Untersuchung der Referenztabellen muss für jeden Task atomar geschehen, da nur so Schreibbarrieren auf lokalen Variablen vermieden werden können. Weitere Aktionen, die atomar ablaufen müssen, sind das Ablegen von Objektreferenzen auf dem Arbeitsstapel und das Einhängen der freien Bereiche in die Freispeicherliste.

3.5.5 Interne und externe Fragmentierung

Im Gegensatz zur **RestrictedDomainScope**-Variante erzeugen die **IRR** und die **CoffeeBreak**-Variante der Speicherverwaltung eine interne und externe Fragmentierung des Speichers. Von externer Fragmentierung spricht man, wenn die freien Speicherblöcke durch belegte Speicherbereiche unterbrochen werden. Dies ist die Folge von ungeordneter Belegung und Freigabe von Speicher durch die Anwendung und kann dazu führen, dass eine Speicheranforderung nicht bedient werden kann, wenn nicht genügend zusammenhängender freier Speicher existiert, obwohl die Summe an freien Speicherblöcken noch ausreichen würde.

Die interne Fragmentierung entsteht, wenn die Speicherverwaltung nur Speicherbereiche mit einer festen Granularität oder einer Mindestgröße vergibt und so Bereiche belegt werden, die von der Anwendung jedoch nicht genutzt werden können. Auch von der Speicherverwaltung zusätzlich benötigte Informationen, die für den allozierten Speicherbereich benötigt werden, zählen zur internen Fragmentierung. Da abgesehen vom Objektkopf und einem darin enthaltenen Epochenzähler keine zusätzlichen Informationen von der Speicherbereinigung benötigt werden, und ein Objekt, welches nur aus dem Objektkopf besteht, bereits ein vollwertiges Objekt darstellt, vernachlässigen wir diesen Speicherbedarf für die Betrachtung der internen Fragmentierung.

Die Granularität wird bei **CoffeeBreak** und **IRR** durch die Slotgröße bestimmt, welche in der Konfigurationsdatei für jede Domäne unabhängig konfiguriert werden kann. Abhängig von der Mikrocontroller-Architektur gibt es eine Mindestslotgröße, welche mindestens die Datenstruktur zur Verwaltung der Freispeicherliste aufnehmen muss. Die minimale Slotgröße liegt beim TriCore bei 8 Byte. Es kann jedoch durchaus sinnvoll sein, die Slotgröße zu erhöhen, um ein kleineres Bitfeld für die Speicherbereinigung zu bekommen, den Durchsatz der Speicherbereinigung zu erhöhen oder um der externen Fragmentierung entgegenzuwirken.

Ein Nachteil der Fragmentierung ist eine schlechtere Speichernutzung und damit verbunden ein größerer Speicherbedarf der Anwendung. Die Slotgröße kann durch Analyse der durchschnittlichen Objektgröße an die Anwendung angepasst werden, so dass eine minimale interne Fragmentierung erreicht wird.

Die externe Fragmentierung hängt vom Verhalten der Anwendung ab und ist in vielen Fällen nicht vorhersagbar. Für Anwendungen, die harte Echtzeitanforderungen erfüllen sollen, ist dies kein zufriedenstellender Zustand. Daher besteht in diesem Punkt noch Entwicklungsbedarf. Eine externe Fragmentierung kann im Moment nur verhindert werden, wenn die Slotgröße gleich der maximalen angeforderten Objektgröße ist. Die maximale Objektgröße kann KESO automatisch ermitteln, solange keine Arrays verwendet werden, deren Größen erst zur Laufzeit bekannt sind. Diese Strategie verhindert zwar die externe Fragmentierung zuverlässig, sie führt jedoch in den meisten Fällen zu einer größeren internen Fragmentierung.

Die **RestrictedDomainScope**-Variante weist weder eine interne noch eine externe Fragmentierung des Speichers auf. Der kombinierte Einsatz von unterschiedlichen Speicherverwaltungen kann daher dazu dienen, Komponenten, die harte Echtzeitanforderungen erfüllen sollen, von Teilen der Software zu trennen, die von einer dynamischen Speicherverwaltung profitieren und so die positiven Eigenschaften beider Strategien auf einem Mikrocontroller zu vereinen.

Eine andere Möglichkeit wäre die Entwicklung einer Speicherverwaltung, die die Objekte im Speicher verschieben kann. Dies ist bei einer nebenläufigen Speicherbereinigung jedoch auch mit einem zusätzlichen Speicherbedarf verbunden, da das atomare Anpassen

der Objektreferenzen entweder eine Indirektionstufe oder eine längere synchronisierte Zeitspanne benötigt.

3.5.6 Speicherbedarf im Vergleich zu verwandten Arbeiten

Von der Grundidee her verwendet die echtzeitfähige Speicherbereinigung ein **Tricolor Marking**, wie es bei anderen unterbrechbaren Speicherbereinigungen auch verwendet wird [12, 17, 21, 27, 54, 84, 91]. Beim **Tricolor Marking** werden die Objekte in drei Mengen mit den Farben, Weiß, Grau und Schwarz eingeteilt. Die weißen Objekte entsprechen dabei der Menge der Objekte, die aus der alten Epoche stammen und nicht oder noch nicht gefunden wurden. Die grauen Objekte bilden die Menge der bereits gefunden Objekte, welche jedoch noch nicht untersucht wurden und daher noch auf dem Arbeitsstapel liegen. Die Menge der schwarzen Objekte bilden die Objekte, die der neuen Epoche zugehören und nicht mehr untersucht werden müssen und daher auch nicht mehr auf dem Arbeitsstapel liegen.

Der Algorithmus von Baker [21] verrichtet bei jeder Speicheranforderung auch einen kleinen Teil der Speicherbereinigung, indem er lebende Objekte in einen neuen Speicherbereich kopiert und den alten Speicherbereich dabei leert. Er braucht daher den doppelten Speicherplatz und ist für das Anwendunggebiet somit nicht geeignet. Die Speicherbereinigung findet zum Zeitpunkt der Speicheranforderung statt und verzögert dabei die Anwendung möglicherweise auch zu einem ungünstigen Zeitpunkt.

Günstiger sind zeitbasierte Verfahren, die einen festen Anteil an der Rechenzeit für die Speicherbereinigung vorsehen und diese zum Beispiel durch eine zeitgesteuerte Auftragsplanung garantieren [84]. Auch bei KESO muss bei der Auftragsplanung dem Task der Speicherbereinigung genügend Zeit eingeräumt werden, da ansonsten nicht genügend freier Speicher zum Zeitpunkt der Speicheranforderung zur Verfühgung steht.

Damit die Objekte ohne gleichzeitiger Anpassung der Objektereferenzen verschoben werden können, nutz Metronome [16] eine zusätzliche Referenz im Objektkopf. Diese verweist bei nicht verschobenen Objekten auf sich selbst und bei einem kopierten Objekt auf die Kopie. Alle Objektzugriffe erfolgen immer über die zusätzliche Dereferenzierung dieser Referenz und die Objektreferenzen werden schrittweise angepasst.

Die echtzeitfähige Speicherbereinigung von KESO benötigt nur ein Bit im Objektkopf für den Epochenzähler und zusätzlich Speicherplatz für den Arbeitsstapel und das Bitfeld. Die Größe des Arbeitsstapels lässt sich durch die Anzahl der maximal lebenden Objekte, nach oben hin begrenzen. Für den ungünstigen Fall, dass die Anzahl der maximal lebenden Objekte nicht bestimmt werden kann, ist die Obergrenze durch die Anzahl der maximal allozierbaren Objekte gegeben. In diesem Fall entspricht die Speicherbelegung für den Arbeitsstapel genau einer zusätzlichen Adresse in jedem Objekt. Der Speicherbe-

darf ist daher geringer als bei einer doppeltverketteten Liste, wie sie zur Verwaltung der Objektmengen bei einigen anderen Verfahren verwendet wird.

3.6 Zusammenfassung

Mit KESO wird eine Multi-JVM auf der Basis von OSEK/VDX umgesetzt, um einen plattformunabhängigen Speicherschutz und die Isolation von mehreren Anwendungen auf einem Mikrocontroller zu bieten. OSEK/VDX ist eine Systemplattform, die auf die Anforderungen von Steuergeräten im Automobilbereich ausgelegt ist. Durch eine statische Konfiguration kann OSEK/VDX auf die Anforderungen der Anwendung maßgeschneidert werden und ermöglicht so den Einsatz von preisgünstigerer Hardware.

Eine JVM bietet auf Grund des typsicheren Bytecodes die Möglichkeit, unterschiedliche Softwarekomponenten in separaten Schutzräumen zu isolieren, ohne dass die Hardware spezielle Funktionalität zur Verfügung stellen muss. Um eine vollständige Isolation zwischen den Komponenten zu erreichen, wurde für KESO das Konzept von mehreren virtuellen JVMs von JX übernommen.

Die Vorteile von beiden Welten werden im Bezug auf das Anwendungsgebiet in der KESO-Architektur vereint. Die von OSEK/VDX angebotenen Dienste und Funktionalität werden erhalten und um neue Funktionalität ergänzt. Von OSEK/VDX erbt KESO die statische Konfiguration, das schlanke Taskmodell mit festen Prioritäten und den ICPP zur Synchronisation. Von der JVM übernimmt KESO den typsicheren und plattformunabhängigen Bytecode, die automatische Speicherbereinigung und ein Objekt-orientiertes Programmiermodell.

Die Isolation der Anwendungen wird durch die Aufteilung in Schutzräume erreicht, wobei jeder Schutzraum eine eigenständige JVM repräsentiert. Für die Kommunikation zwischen Anwendungen werden unterschiedliche Dienste bereitgestellt, wie die Portale, das KNI und die Speicherobjekte.

Kapitel 4

Aufbau der internen Datenstrukturen

4.1 Einleitung

Als Argument gegen den Einstaz von Java auf eingebetteten Systemen wird der höhere Bedarf an Speicherplatz und Rechenleistung angeführt. Mit der vorgestellten Architektur von KESO soll nicht nur eine JVM auf einem eingebetteten System realisiert werden, sondern gleich mehrere JVM zur Verfügung gestellt werden. Ein besonderes Ziel bei der Enwicklung von KESO war daher die Entwicklung einer Java-Laufzeitumgebung mit einem möglichst geringen Speicherverbrauch.

In diesem und im nachfolgenden Kapitel wird JINO, der Systemgenerator von KESO, vorgestellt. Durch die automatische Generierung aller von KESO benötigten Datenstrukturen und dem Übersetzen von Java-Bytecode nach C und anschließend in ein statisches Maschinenprogramm soll der Mehrbedarf an Hardwareressourcen minimiert werden.

Es ist dabei wichtig, die JVM auf die Anforderungen der Anwendung abzustimmen und auf jede nicht benötigte Funktionalität zu verzichten. In der Architektur wurden bereits einige Einschränkungen der JVM beschrieben. Die wichtigste Änderung ist dabei der Verzicht auf das dynamische Laden von Klassen, da so die Möglichkeit geschaffen wird, zum Übersetzungszeitpunkt eine vollständige Sicht auf die verwendeten Klassen zu bekommen und moderne Techniken aus dem Übersetzerbau zu nutzen, um eine für die Anwendung angepasste Laufzeitumgebung zu generieren.

Im folgenden Abschnitt sollen nun zuerst die vom Übersetzer erzeugten Datenstrukturen beschrieben werden.

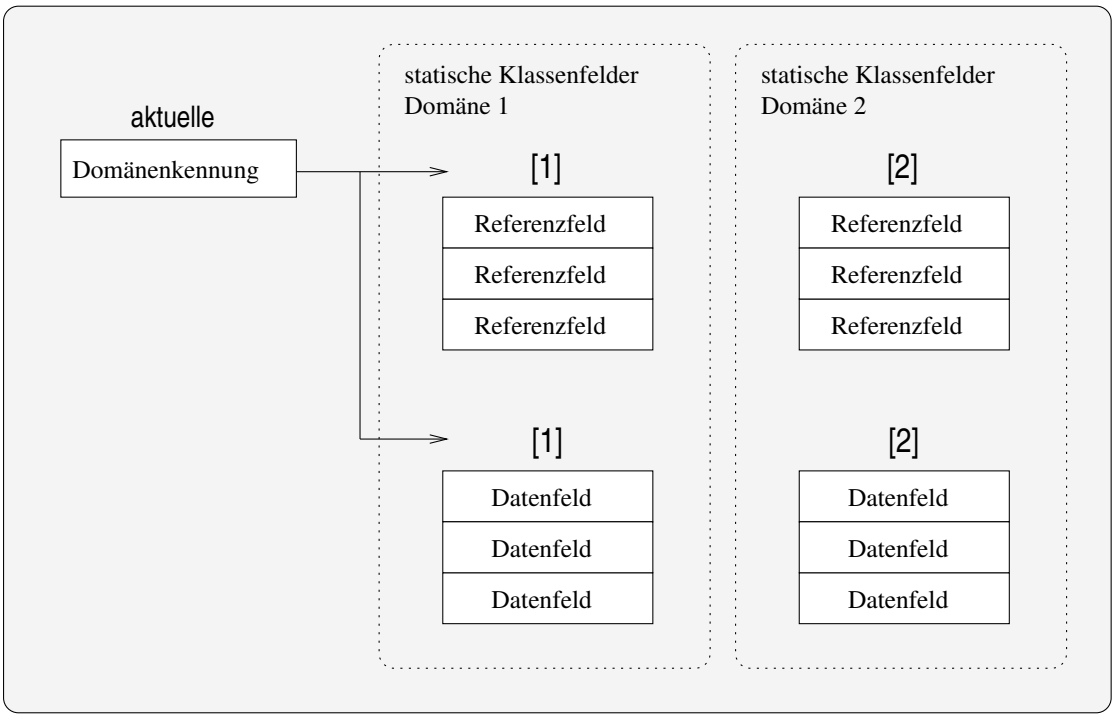

Abbildung 4.1: Statische Klassenfelder für jede Domäne

4.2 Globale Datenstrukturen

4.2.1 Domänenkennung

Besitzt die Konfiguration von KESO mehrere Domänen auf einem Mikrocontroller, so wird eine Domänenkennung auf dem Mikrocontroller verwaltet. Für die Kennung wird je nach Mikrocontrollerarchitektur ein Byte oder ein Register reserviert. Der TriCore TC1796 besitzt vier Register, welche für globale Aufgaben vorgesehen sind. Für die Domänenkennung kann eines dieser Register verwendet werden.

Jeder Task wird in der Konfiguration einer Domäne zugeordnet. Die Kennung wird im Taskobjekt gespeichert und nach einer Taskumschaltung wird im **PreTaskHook** die aktuelle Domänenkennung auf die für den Task gültige Domäne umgeschaltet.

4.2.2 Klassenfelder

Die statischen Klassenfelder werden für jede Domäne getrennt angelegt. Wird auf ein Klassenfeld zugegriffen, so werden über die aktuelle Domänenkennung die aktuell gülti-

gen Klassenfelder ausgewählt. Wie in Abbildung 4.1 dargestellt, werden die Klassenfelder, je nachdem, ob sie eine Objektreferenz enthalten oder einen primitiven Datentypen, getrennt verwaltet, damit die automatische Speicherverwaltung die Objektreferenzen von den Datenfeldern unterscheiden kann und diese beim Durchsuchen zusammenhängend im Speicher angeordnet sind.

Bei einer Konfiguration mit mehreren Domänen wird der Zugriff auf statische Felder deutlich aufwändiger, da jedes Feld zuerst über die Domänenkennung ausgewählt werden muss. Es ist daher zu erwarten, dass das Maschinenprogramm in einer solchen Konfiguration etwas umfangreicher ausfällt.

4.2.3 Klassenkennung und Typüberprüfung

Die Klassenkennung spielt in der Laufzeitumgebung von KESO eine zentrale Rolle, da sie den Typ eines Objektes kodiert. Die Kennung kann je nach Anzahl der Klassen unterschiedlich groß sein. Bei einer Konfiguration mit bis zu 255 Klassen werden nur 8 Bit benötigt, und bei einer Konfiguration mit bis zu 65535 Klassen werden 16 Bit genutzt. Der Wert für die Klassenkennung wird von JINO durch eine Nummerierung aller Klassen entlang einer Tiefensuche ermittelt. Hierbei wird die Klassenhierarchie von der Wurzelklasse ausgehend durchlaufend nummeriert, wobei immer zuerst abgestiegen wird und anschließend die Nachbarklassen durchnummeriert werden. In Abbildung 4.2 ist die Wurzelklasse die Klasse A und erhält damit die 1 als Kennung. Die 0 wird von JINO als Klassenkennung nicht vergeben. Objekte mit einer leeren Klassenkennung können so leichter als ein Fehler in den Datenstrukturen erkannt werden.

Die Nummerierung der Klassen entlang einer Tiefensuche ermöglicht nun eine sehr einfache Methode für die Typüberprüfung. Alle Objekte vom Typ B, das sind die Objekte, die entweder eine Instanz der Klasse B, C oder D darstellen, und damit genau die Objekte mit einer Kennung größer/gleich der Kennung der Klasse B und kleiner/gleich der Kennung der Klasse D besitzen. In dem gezeigten Fall sind dies die Klassen mit einer Kennung größer/gleich 2 und kleiner/gleich 4. Die Typüberprüfung reduziert sich daher auf eine einfache Bereichsüberprüfung. Der Startwert ist durch die Klassenkennung des zu prüfenden Typs gegeben, der Endwert durch die größte Kennung der abgeleiteten Klassen.

Anfangs- und Endwert für die Typüberprüfung stehen bereits zum Übersetzungszeitpunkt fest. Für unser Beispiel reicht daher das in Abbildung 4.3 gezeigte C-Programm für eine Überprüfung aus. Das C-Programm ist so gestaltet, dass nur eine Vergleichsoperation benötigt wird, dadurch generiert der C-Übersetzer im Maschinenprogramm nur einen bedingten Sprung. Die Typüberprüfung hat die ideale Eigenschaft, dass nur wenige Maschinenbefehle benötigt werden und der erfolgreiche Test eine konstante Laufzeit besitzt.

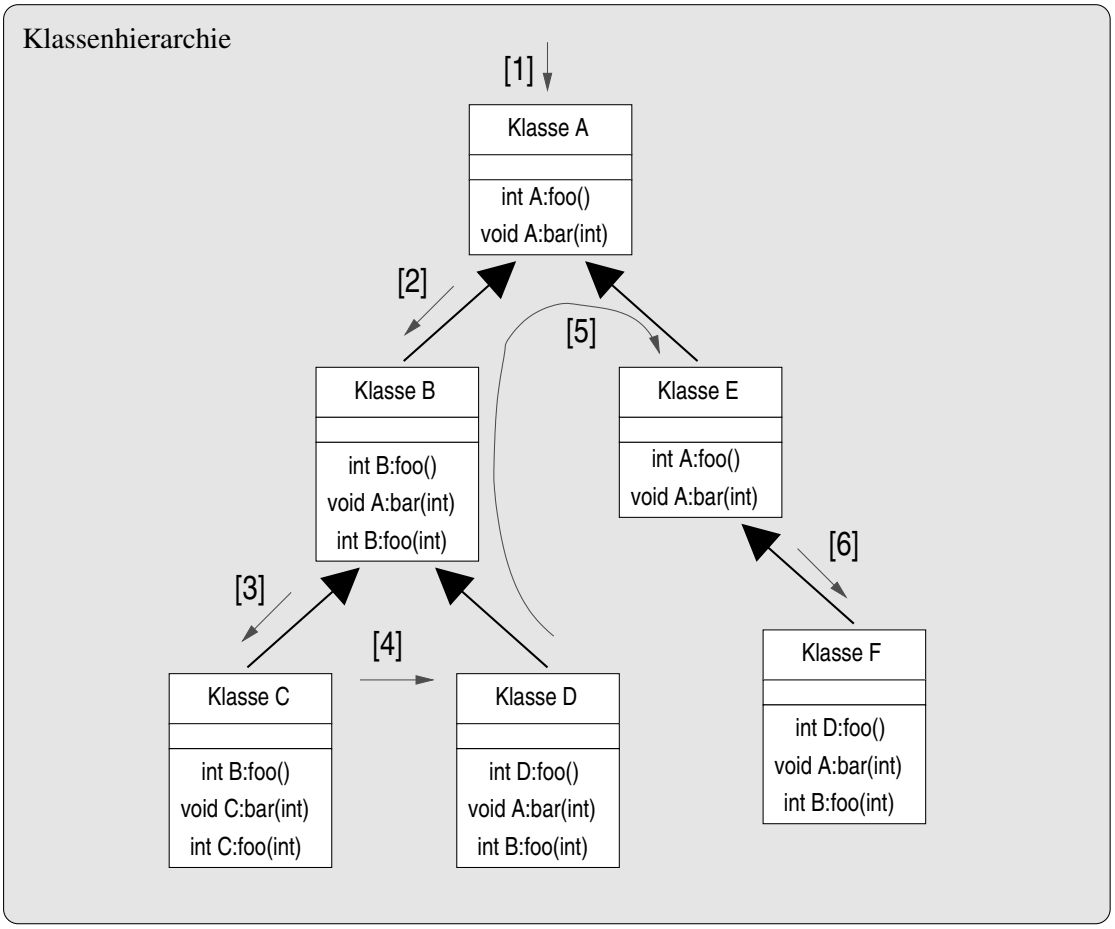

Abbildung 4.2: Ermitteln der Klassenkennung durch eine Tiefensuche in der Klassenhierarchie

```
/* Typüberprüfung bei Klassen */
if ((unsigned char)(obj->class_id-2)>=(5-2))
        KESO_THROW_ERROR();
```

Abbildung 4.3: Beispiel für die Typüberprüfung bei einem primären Objekttyp in C

```
/* Typüberprüfung bei Schnittstellen */
if (!(class_store[obj->class_id-1].ifaces & 0x01))
        KESO_THROW_ERROR();
```

Abbildung 4.4: Beispiel für die Typüberprüfung bei einem sekundären Objekttyp in C

Der Java-Bytecode besitzt zwei Befehle, die zur Laufzeit eine Typüberprüfung durchführen. Beide müssen zusätzlich zur Typüberprüfung noch den Fall behandeln, dass die Objektreferenz eine Null-Referenz darstellt. Daher wird in den meisten Fällen noch eine weitere Überprüfung benötigt. JINO erzeugt die dargestellte kurze Typüberprüfung nur, wenn die Datenflussanalyse ergeben hat, dass die Objektreferenz immer ungleich Null ist.

Das Verfahren wird in der englischsprachigen Literatur als „Relative Numbering“ bzw. „Schubert’s Numbering“ [87] bezeichnet und ist für einen Klassengraphen mit Einfachvererbung in Hinblick auf den benötigten Speicherbedarf optimal. Die Einfachvererbung bei den Klassen bestimmt die Implementierung der Methoden und den Aufbau der Objektdaten. Der Typ in der Klassenhierarchie wird daher im Weiteren als primärer Typ bezeichnet, der Typ, den eine Klasse über eine Schnittstelle zusätzlich bereitstellt, wird im weiteren als sekundärer Typ bezeichnet.

Java besitzt neben einer Einfachvererbung bei Klassen noch eine Mehrfachvererbung bei den Schnittstellen. Bei Mehrfachvererbung kann das “Relative Numbering“ nicht verwendet werden und muss auf eine andere Art realisiert werden.

In KESO wird zur Typüberprüfung bei den Schnittstellen ein einfacher Bitvektor benutzt. Jede Schnittstelle, für die in der Anwendung eine Typüberprüfung stattfindet, bekommt hierfür genau eine Position in einem Bitvektor. Für jede Klasse, die eine Schnittstelle implementiert, wird ein solcher Bitvektor mit entsprechend gesetzten Bits in einer globalen Tabelle hinterlegt. Die globale Tabelle kann z.B. der Klassenspeicher sein, wo für jede Klasse zusätzliche Informationen abgelegt werden. Eine Typüberprüfung auf einen sekundären Typ ist so zur Laufzeit, wie in Abbildung 4.4 dargestellt, mit einer einfachen Überprüfung, ob das entsprechende Bit gesetzt ist, möglich. Diese Operation hat wieder eine konstante Laufzeit.

Das Verfahren hat den Nachteil, dass der Speicherbedarf für die Bitvektoren sowohl mit der Anzahl der Klassen als auch mit der Anzahl der Schnittstellen wächst. In KESO ist der Bitvektor im globalen Klassenspeicher auf eine feste Länge von acht Bit beschränkt. Werden mehr als acht Bit benötigt, so wird eine weitere Indirektionsstufe eingeführt und anstelle der Bitvektoren werden Indizes auf einen eigenen Bitvektorspeicher hinterlegt. Auf diese Weise können Bitvektoren von unterschiedlichen Klassen gemeinsam genutzt werden.

Im Bitvektor werden nur Einträge für Schnittstellen benötigt, für die in der Anwendung eine Typüberprüfung stattfindet, daher kann die Anzahl der Bits einfach reduziert werden, indem nur für solche Schnittstellen ein Eintrag im Bitvektor vorgenommen wird. JINO erkennt dies beim Übersetzungsvorgang und reduziert so die Anzahl der benötigten Bits.

Es gibt eine Reihe von weiteren Verfahren, die eine möglichst optimale Kodierung der Hierarchie in einem Bitvektor für eine Typüberprüfung beschreiben [40, 58, 101, 105]. Am interessantesten für KESO erscheint dabei das *PQ-Encoding* (PQE) [105], welches für die meisten Fälle im Vergleich zu den anderen Verfahren den geringsten Speicherbedarf hat. Zur eigentlichen Typüberprüfung wird, wie beim „Relative Numbering“, nur eine Bereichsüberprüfung benötigt.

PQ-Encoding wurde für JINO jedoch noch nicht implementiert, da die Anzahl der benötigten Bits für das aktuelle Verfahren mit den aktuellen Testanwendungen noch keine aufwändigere Kodierung erforderlich gemacht hat.

Neben der normalen Typüberprüfung muss die Laufzeitumgebung ein reguläres Objekt von einem Arrayobjekt unterscheiden. Ein Arrayobjekt besitzt im Gegensatz zu allen anderen Objekten keine feste Größe; im Objektkopf von Arrays ist daher zusätzlich noch die Anzahl der Arrayelemente gespeichert. Die automatische Speicherverwaltung muss nun ein Arrayobjekt von einem normalen Objekt unterscheiden können, um den Objektkopf richtig zu interpretieren und die genaue Größe bestimmen zu können.

Um nun Arrayobjekte von regulären Objekten unterscheiden zu können, werden die Klassen in der 1. Ebene der Klassenhierarchie, wie in der Abbildung 4.5 dargestellt, vorsortiert. Hierdurch erhalten die Arrayklassen bei der Vergabe der Klassenkennung eine Kennung aus einem zusammenhängenden Bereich. In Abbildung 4.5 sind dies die Klassen mit einer Klassenkennung, die größer/gleich zwei und kleiner/gleich n ist.

Mit einer einfachen Bereichsüberprüfung kann nun ein Arrayobjekt von einem regulären Objekt unterschieden werden. Die Arrayklassen werden von JINO automatisch und nur im Bedarfsfall erzeugt, so dass es auch möglich sein kann, dass keine Arrayklassen im System existieren. In dieser Situation entfällt in der Speicherverwaltung auch die Sonderbehandlung für Arrayobjekte.

Die Klassen — oder deren Unterklassen —, die selbst eine Schnittstelle implementieren, bilden die zweite Gruppe von Klassen in der Klassenhierarchie. In der Abbildung 4.5 sind dies die Klassen von $(n+1)$ bis m. Daher sind auch diese Klassen in einem zusammenhängenden Bereich angeordnet.

Diese Anordnung ermöglicht es, die Bitvektoren zur Typüberprüfung aus dem Klassenspeicher auszulagern und diese in einer separaten Tabelle, dem Schnittstellenspeicher, abzulegen. Der Schnittstellenspeicher beinhaltet nur Einträge für Klassen, die eine Klassenkennung aus dem beschriebenen Bereich besitzen. Die Tabelle besitzt daher weniger Einträge als der Klassenspeicher und benötigt weniger Speicherplatz.

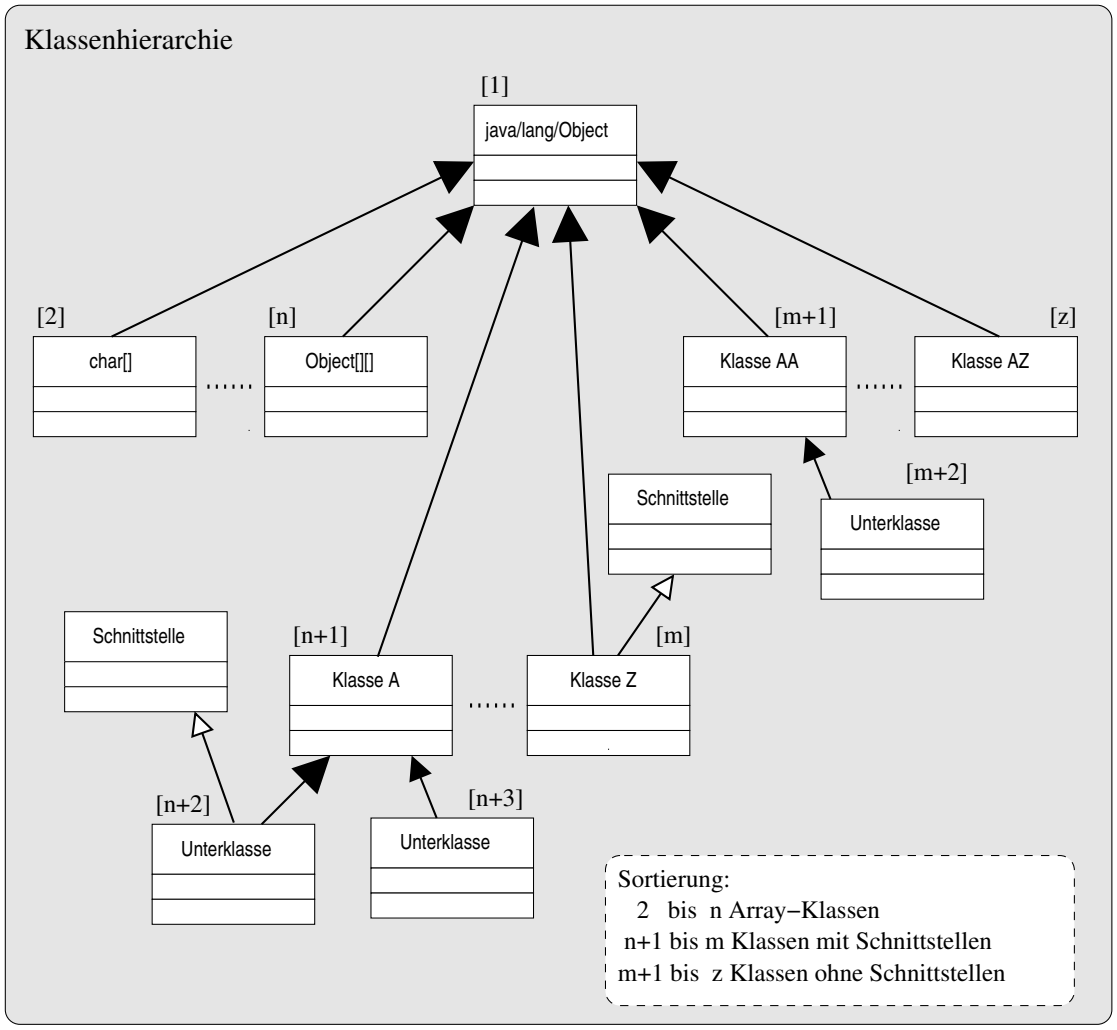

Abbildung 4.5: Sortierung der Klassenhierarchie kodiert zusätzliche Informationen

Bei einer Typüberprüfung wird durch eine Bereichsüberprüfung ermittelt, ob die Objektklasse einen Eintrag im Schnittstellenspeicher besitzt und, wenn dies zutrifft, die bereits beschriebene Typüberprüfung durchgeführt. Gibt es für die Klassen keinen Eintrag im Schnittstellenspeicher, so implementiert die Klasse auch keine Schnittstelle und folglich auch nicht die Schnittstelle, auf die getestet wird.

Durch die zusätzliche Bereichsüberprüfung wird die Typüberprüfung geringfügig aufwändiger. Über einen Kommandozeilenparameter von JINO kann daher ausgewählt werden, ob ein separater Schnittstellenspeicher verwendet werden soll oder ob die Schnittstelleninformation im Klassenspeicher hinterlegt wird.

Anschließend werden die restlichen Klassen behandelt. Schnittstellen, die in Java eigenständige Klassendateien ohne Programmcode darstellen, erhalten keinen Eintrag im Klassenspeicher. Da es nicht möglich ist, eine Schnittstelle zu instanziieren, wird für diese Klassen auch keine zusätzliche Information im Klassenspeicher benötigt.

4.2.4 Globaler Klassenspeicher

Die benötigten Typ- und Zusatzinformationen für Klassen werden in einem globalen Klassenspeicher abgelegt, welcher von allen Domänen gemeinsam genutzt wird. Zu den im Klassenspeicher abgelegten Informationen zählen die Objektgröße, die Anzahl der im Objekt enthaltenen Objektreferenzen und möglicherweise die sekundäre Typinformation. Die Einträge im globalen Klassenspeicher werden direkt durch die Klassenkennung indiziert, so dass die Informationen, wie bei der Typüberprüfung in Abbildung 4.4 gezeigt, durch einen einfachen Arrayzugriff zur Verfügung stehen.

Abbildung 4.6 zeigt eine mögliche Variante für den Aufbau des globalen Klassenspeichers. Welche Einträge jedoch tatsächlich im Klassenspeicher enthalten sind, hängt von der Anwendung und von der Entscheidung des Entwicklers ab, ob mehr Arbeitsspeicher (RAM) oder mehr Programmspeicher (ROM) verwendet werden soll. Eine Entscheidung zu Gunsten einer geringeren Belegung von Arbeitsspeicher führt dabei in der Regel zu einem schlechteren Laufzeitverhalten.

Die Objektgröße und die Anzahl der Objektreferenzen wird von der **CoffeeBreak**- und der **IRR**-Variante der automatischen Speicherverwaltung benötigt. Durch diese Information kann die Speicherverwaltung exakt nur die Referenzen lesen und den Speicherbereich von einzelnen Objekten freigeben. Die **RestrictedDomainScope**-Variante benötigt dieses Wissen nicht, daher können bei der ausschließlichen Verwendung dieser Variante die entsprechenden Einträge entfallen.

Für die Anzahl der Objektreferenzen kann der Generator anstelle der Einträge im Klassenspeicher auch eine Generatorfunktion erzeugen, welche die Anzahl abhängig von der Klassenkennung liefert. Dazu wird einfach eine C-Funktion generiert, die über eine

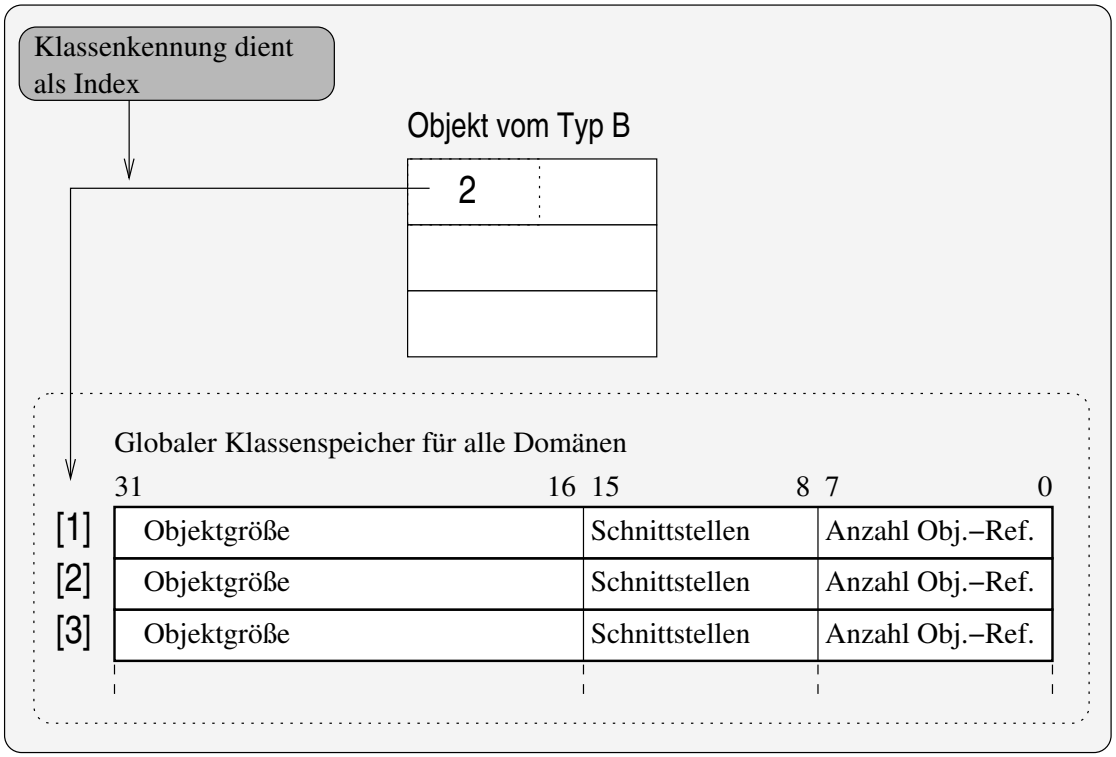

Abbildung 4.6: Aufbau des globalen Klassenspeichers

```
jboolean keso_instanceof_iface(object_t* obj, unsigned int id) {

        /* Laut Definition ist die NULL-Referenz eine
         * mögliche Instanz für jede Klasse.  */
        if (obj==NULL) return 1;

        /* Zustätzliche Überprüfung ob ein Eintrag
         * im Schnittstellenspeicher exisistiert. */
        if ((unsigned int)(obj->class_id-IFACE_START)>=IFACE_SIZE)
                return 0;

        /* Typüberprüfung über Eintrag im
         * separaten Schnittstellenspeicher */
        if (!(iface_store[obj->class_id-IFACE_START] & id))
                return 0;

        return 1;
}
```

Abbildung 4.7: Sekundäre Typüberprüfung mit Schnittstellenspeicher

`switch`-Anweisung die entsprechende Anzahl auswählt und als Wert zurückliefert. Mit Hilfe einer Generatorfunktion wird die Information komplett vom Datenspeicher in den Programmspeicher verlagert, als Nachteil steigt jedoch der benötigte Rechenaufwand. Da kleinste eingebettete Systeme oft über deutlich mehr Programmspeicher als Datenspeicher verfügen, die Anzahl der Objektreferenzen nur von der Speicherverwaltung in einem niederprioren Task benötigt wird, und es sich daher um keine zeitkritische Information handeln sollte, ist diese Maßnahme eine sinnvolle Alternative.

Die Typinformation für die Schnittstellen kann, wie bereits beschrieben, in einen eigenen Schnittstellenspeicher verlagert werden. Es ist jedoch eine zusätzliche Bereichsüberprüfung nötig. Die Typüberprüfung für den Sekundärtyp erfolgt in diesem Fall durch eine kleine C-Funktion, die in Abbildung 4.7 dargestellt ist. Die Einsparung beim Speicherbedarf wird daher auch hier durch einen höheren Rechenaufwand erreicht.

JINO erzeugt sehr unterschiedliche und flexible Datenstrukturen, um die benötigten Zusatzinformation zu speichern. Einige Entscheidungen trifft JINO dabei in Abhängigkeit vom verwendeten Mikrocontroller und aufgrund einer Analyse der Anwendung selbständig, andere können zum Übersetzungszeitpunkt über Kommandozeilenparameter ausgewählt werden.

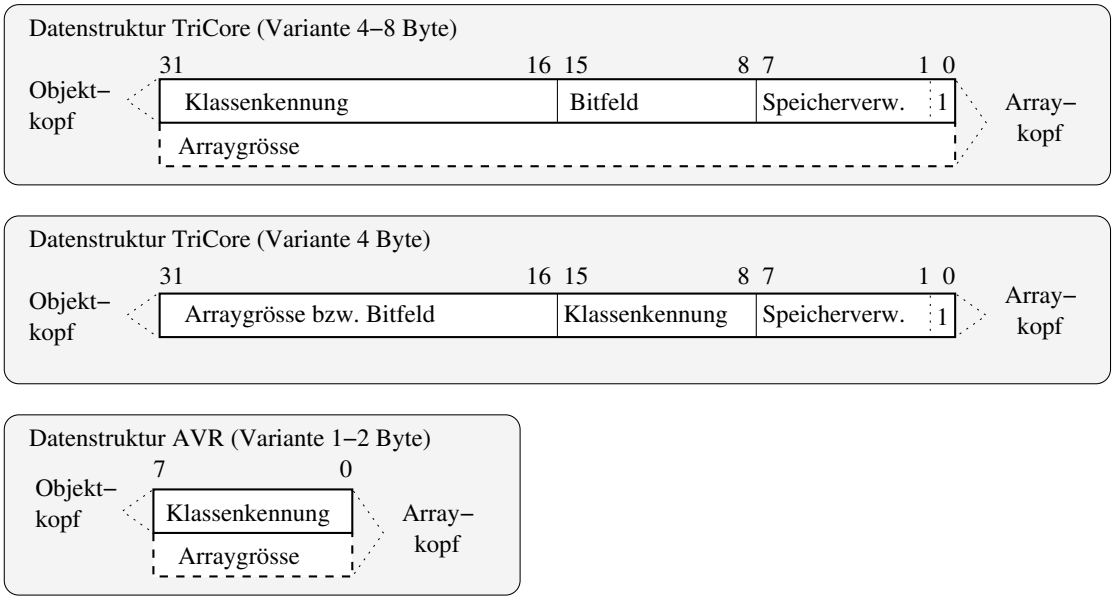

Abbildung 4.8: Aufbau des Objektkopfs von regulären Objekten und Arrayobjekten

4.3 Objektaufbau

4.3.1 Objektkopf

Mit dem Objektaufbau wird die interne Darstellung von Objekten in der virtuellen Maschine beschrieben. Ein Objekt besteht zur Laufzeit aus den eigentlichen Datenfeldern der Klasseninstanz und aus Zusatzinformationen, die von der Laufzeitumgebung benötigt werden, um zum Beispiel den Typ des Objektes zu bestimmen. Die Zusatzinformationen für das Laufzeitsystem werden im Objektkopf gespeichert, welcher für alle Objekte gleich ist und daher auch dann zuverlässig interpretiert werden kann, wenn der Objekttyp noch nicht bekannt ist.

Aufgrund der in der JVM-Spezifikation für Java-Objekte beschriebenen Grundfunktionalität benötigt eine JVM für alle Objekte zusätzliche Informationen. KESO implementiert diese Grundfunktionalität bewusst nicht vollständig konform zur Java-Spezifikation [66], da einige Funktionen, wie in Kapitel 3.3 beschrieben, für das Anwendungsfeld weniger geeignet erscheinen und unnötig den Ressourcenbedarf erhöhen. Im Folgenden sollen die typischen Objekteigenschaften und ihre Umsetzung beschrieben werden.

Objekttyp: Der Typ eines Objektes muss zur Laufzeit bestimmbar sein, dafür muss jedes Objekt einen eindeutigen Hinweis auf seine Klasse geben. Der Objekttyp

wird beim virtuellen Methodenaufruf und für die Typüberprüfung benötigt. In KESO ist dies durch die in Kapitel 4.2.3 beschriebene Klassenkennung realisiert. Die Größe der Klassenkennung wird zum Übersetzungszeitpunkt in Abhängigkeit von der Anzahl der Klassen und der Wortbreite der Mikrocontrollerarchitektur automatisch bestimmt und belegt entweder 8 oder 16 Bit im Objektkopf.

Hashwert: Jedem Objekt ist ein fester Hash-Schlüssel zugeordnet. In KESO dient hierfür die Speicheradresse des Objektes. Objekte werden in einigen JVMs bei der automatischen Speicherbereinigung verschoben, um so die Fragmentierung des Speichers zu bereinigen. Die in KESO bis jetzt eingesetzten Verfahren machen von dieser Möglichkeit keinen Gebrauch, daher ist die Speicheradresse eines Objektes für die gesamte Lebensdauer eindeutig und konstant.

Synchronisation: In Java kann jedes Objekt zur Synchronisation als Primitive für einen gegenseitigen Ausschluss und zum Signalisieren einer Zustandsänderung verwendet werden. Jedes Objekt muss daher Zusatzinformationen für die Synchronisation bereitstellen. Die Mehrzahl der Objekte in einer JVM wird jedoch nie zur Synchronisation verwendet. Um den unnötigen Speicherbedarf zu reduzieren, wurde mit den „thin locks“ von David F. Bacon [19] eine sehr speicherplatzeffiziente Methode entwickelt. Die „thin locks“ benötigen im Regelfall nur ein Bit im Objektkopf, und erst bei einem konkurrierenden Zugriff werden weitere Datenstrukturen aufgebaut. In KESO wird — wie in der *Real-Time Spezifikation für Java* (RTSJ) vorgeschlagen — die Emulation des „Priority Ceiling Protocol“ angeboten. Alternativ steht über die OSEK/VDX-Programmierschnittstelle das vollwertige ICPP zur Verfügung. Beide Verfahren benötigen keine zusätzlichen Informationen im Objektkopf.

Automatische Speicherbereinigung: In Java wird der Speicher für die Objekte dynamisch zur Laufzeit belegt, und nicht mehr genutzter Speicher wird automatisch wieder freigegeben. Es ist üblich, für den Vorgang der automatischen Speicherbereinigung benötigte Zustandsdaten im Objektkopf zu speichern. Die Anzahl der benötigten Bits hängt dabei stark von der verwendeten Speicherverwaltung ab. Für die in Kapitel 3.5.3 beschriebenen Verfahren wird z.B. ein Bit für einen Epochenzähler benötigt.

Wie beim Klassenspeicher erzeugt JINO auch für den Objektkopf eine bedarfsgerechte Datenstruktur. Abbildung 4.8 zeigt den Aufbau von unterschiedlichen Objektköpfen. Die ersten beiden Beispiele sind für den TriCore Mikrocontroller TC1796, welcher eine Ausrichtung der Daten auf 32 Bit Speichergrenzen benötigt. Der erste Objektkopf wurde für eine Konfiguration erzeugt, bei der die Anwendung mehr als 255 Klassen oder Arrays mit mehr als 65536 Einträgen benötigt. Der Objektkopf ist 32 Bit groß. Die Klassenkennung belegt die hochwertigen 16 Bit im Objektkopf. Die niederwertigsten acht Bit werden für die automatische Speicherverwaltung genutzt, wobei zur Zeit nur zwei Bit

wirklich benötigt werden: ein Bit für den in Kapitel 3.5.4 beschriebenen Epochenzähler und das niederwertigste Bit, welches dazu dient, den Objektkopf von einer Objektreferenz zu unterscheiden.

Die verbleibenden acht Bit im Objektkopf werden für Felder genutzt, die nur zwei Zustände aufweisen, wie es bei Wahrheitswerten der Fall ist, für die Java einen eigenen primitiven Datentyp kennt. Das Bitfeld gehört daher nicht mehr zum eigentlichen Objektkopf, da es bereits anwendungsspezifische Daten enthalten kann. Handelt es sich beim Objekt um ein Array, so wird der Objektkopf um ein weiteres 32-Bit-Wort erweitert, welches dazu dient, die Arraygröße zu speichern.

Die zweite in Abbildung 4.8 dargestellte Variante unterstützt maximal 255 Klassen und Arrays mit bis zu 65536 Elementen, was je nach Anwendungsgebiet bereits ausreichend ist. Für die Klassenkennung und die Speicherverwaltung werden hier nur die niederwertigen 16 Bit benötigt. Die höherwertigen 16 Bit fassen entweder die Arraygröße oder dienen wieder als Bitfeld.

Das letzte Beispiel ist ein Objektkopf für den 8-Bit-Mikrocontroller ATmega8535 mit einem Arbeitsspeicher von 512 Byte. Die Einschränkung auf 255 Klassen und Arrays mit maximal 256 Elementen stellt im Hinblick auf die begrenzte Arbeitsspeicherkapazität wohl kaum eine störende Einschränkung dar. Die Klassenkennung als Objektkopf belegt nur acht Bit. Die Arraygröße wird bei Bedarf in einem separaten 8-Bit-Wort gespeichert.

Im Zusammenhang mit der IBM *Research Virtual Machine* (RVM) wurde ein kompakter Aufbau von Objektköpfen auch für Server und Desktoprechner untersucht [18]. Der kleinste dabei verwendete Objektkopf hatte eine Größe von einem Maschinenwort, was auf der verwendeten PowerPC-Architektur 32 Bit entsprach. Die höherwertigen 30 Bit wurden dabei als Zeiger interpretiert, während von den verbleibenden zwei Bits je eines für die Synchronisation und eines für den Hashwert verwendet wurden. Als Hashwert für das Objekt diente dabei die Speicheradresse. Da die Speicherverwaltung der RVM jedoch lebende Objekte verschiebt, wird nach dem Verschieben die ursprüngliche Adresse in einem zusätzlichen 32-Bit-Wort am Ende vom Objekt gespeichert und das Bit im Objektkopf gesetzt, um der RVM zu signalisieren, dass bei zukünftigen Operationen der gespeicherte Hashwert verwendet werden soll. Objekte, die verschoben wurden, belegen daher zusätzliche 32 Bit für den Hashwert. Für die Synchronisation wurden die bereits erwähnten „thin locks“ [19] verwendet, auch hier entsteht im Bedarfsfall ein zusätzlicher Speicherbedarf.

Bei der KVM [60] und der Jelatine JVM [6] wird jeweils nur ein 32 Bit langes Maschinenwort für den Objektkopf verwendet. Die Arraygröße wird bei allen diesen Implementierungen immer separat im Objekt gespeichert.

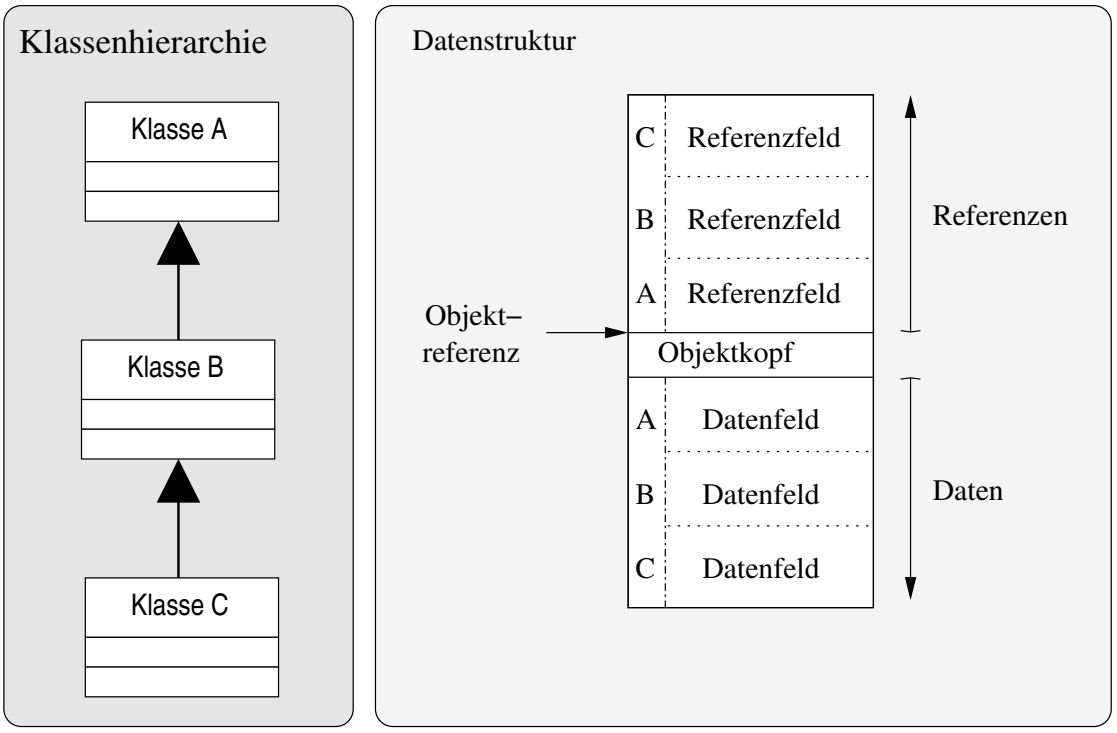

Abbildung 4.9: Bidirektionaler Objektaufbau

4.3.2 Bidirektionaler Objektaufbau

Abbildung 4.9 zeigt den Aufbau der von KESO verwendeten Objekte anhand einer Instanz der Klasse C. Objektreferenzen verweisen zur Laufzeit immer auf den Objektkopf. Datenfelder für primitive Datentypen schließen an den Objektkopf mit aufsteigenden Adressen an. Felder, die Referenzen enthalten, schließen hingegen in Richtung von absteigenden Adressen an den Objektkopf an. Die Position der geerbten Felder wird dabei von der Objektreferenz ausgehend immer beibehalten, so dass ein Objekt der Klasse C auch als ein Objekt der Klasse A oder Klasse B ohne Anpassung der Objektreferenz verwendet werden kann.

Die Objektstruktur kann so bei jeder Vererbung in zwei Richtungen wachsen. Dies ist nötig, damit die Objektreferenzen immer zusammenhängend im Speicher angeordnet werden. Die zusammenhängende Anordnung ermöglicht es, dass alle im Objekt gespeicherten Referenzen effizient und zuverlässig gefunden werden. Die einzige benötigte Zusatzinformation ist die Anzahl der Referenzen. Diese Information ist wie beschrieben im globalen Klassenspeicher (siehe Kapitel 4.2.4) hinterlegt. Die Idee zum bidirektionalen Objektaufbau stammt aus der Implementierung der SableVM [39].

Der Aufbau von Objekten spielt für den Speicherbedarf und die Implementierung der JVM eine wichtige Rolle. Während die Klassen die Objektstruktur und die möglichen Methoden zum Bearbeiten der Daten beschreiben, repräsentieren die Objekte als Laufzeitinstanzen einer Klasse den aktuellen Zustand. Es muss daher für jedes Objekt genau eine passende Klasse existieren, jedoch können mehrere Objekte von der gleichen Klasse instanziiert werden. Ein kompakter Objektaufbau ist daher noch wichtiger für den Speicherbedarf als z.B. ein möglichst kompakter Klassenspeicher.

In KESO ist daher der Objektkopf möglichst kompakt gehalten. Bei den Objektfeldern wird zusätzlich versucht die Anzahl der Felder zu reduzieren. Der Vorgang wird bei der Beschreibung der lokalen Optimierungen in Kapitel 5.4.3 noch genauer dargestellt.

4.4 Virtuelle Methodenaufrufe

Das objektorientierte Programmiermodell von Java führt zu einer großen Anzahl von virtuellen Methodenaufrufen. Es ist daher wichtig, hier ein Verfahren zu finden, welches sowohl ein effizientes Aufrufen von Methoden erlaubt, als auch mit einem möglichst geringen Speicherbedarf auskommt.

4.4.1 Definition Methodentyp

Bei einem virtuellen Methodenaufruf ist es nötig, zur Laufzeit aus einer Menge von möglichen Kandidaten abhängig vom Objekttyp die gültige Implementierung auszuwählen. Die Menge der möglichen Implementierungen soll im Weiteren durch den Begriff des Methodentyps beschrieben werden. Dabei implementieren Methoden den gleichen Methodentyp, wenn folgende Bedingungen zutreffen:

Identischer Bezeichner: Die Methoden besitzen alle den gleichen Methodennamen.

Identische Parameter: Die Methoden besitzen alle die gleiche Anzahl an Parametern und die Parameter sind alle vom gleichen Typ. Dies gilt auch für den Typ des Rückgabewertes.

Bestehende Vererbungsbeziehung: Zwischen den Klassen, die die Methoden definieren, besteht eine Vererbungsbeziehung.

Alle Methoden, die für einen virtuellen Methodenaufruf in Frage kommen, besitzen den gleichen Methodentyp. Betrachten wir nun zuerst einmal Verfahren zur Bestimmung der gültigen Methode, mit der Einschränkung, dass nur eine Einfachvererbung zwischen den Klassen besteht.

4.4.2 Methodentabellen

Ein einfaches und effizientes Verfahren zum Auffinden der gültigen Sprungadresse ist die Verwendung von Methodentabellen, wobei jede Methodentabelle für je eine Klasse die Adressen der für die Klasse gültigen Methoden enthält. Die Methodentabelle wird dabei ausgehend von der Wurzelklasse der Klassenhierarchie aufgebaut und jeder neu eingeführte Methodentyp bekommt eine neue feste Position in der Tabelle. Überschreibt eine Klasse eine Methode der Oberklasse, so wird auch die Adresse an der vom Methodentyp genutzten Position überschrieben. Abbildung 4.10 zeigt den Aufbau von entsprechenden Methodentabellen.

Da die Position für einen Methodentyp in der Tabelle fest ist, kann zur Laufzeit die gültige Methodenadresse über einen festen Index und einen Zeiger auf die passende Methodentabelle ermittelt werden. Der Zeiger auf die Methodentabelle kann hierfür direkt im Objektkopf gespeichert werden, wie es in Abbildung 4.10 dargestellt ist.

Der Aufwand zur Bestimmung der gültigen Adresse besteht aus einem Feldzugriff zum Auslesen des Zeigers auf die Methodentabelle und einem weiteren Feldzugriff, um die eigentliche Adresse aus der Methodentabelle zu lesen. Der Speicherbedarf beschränkt

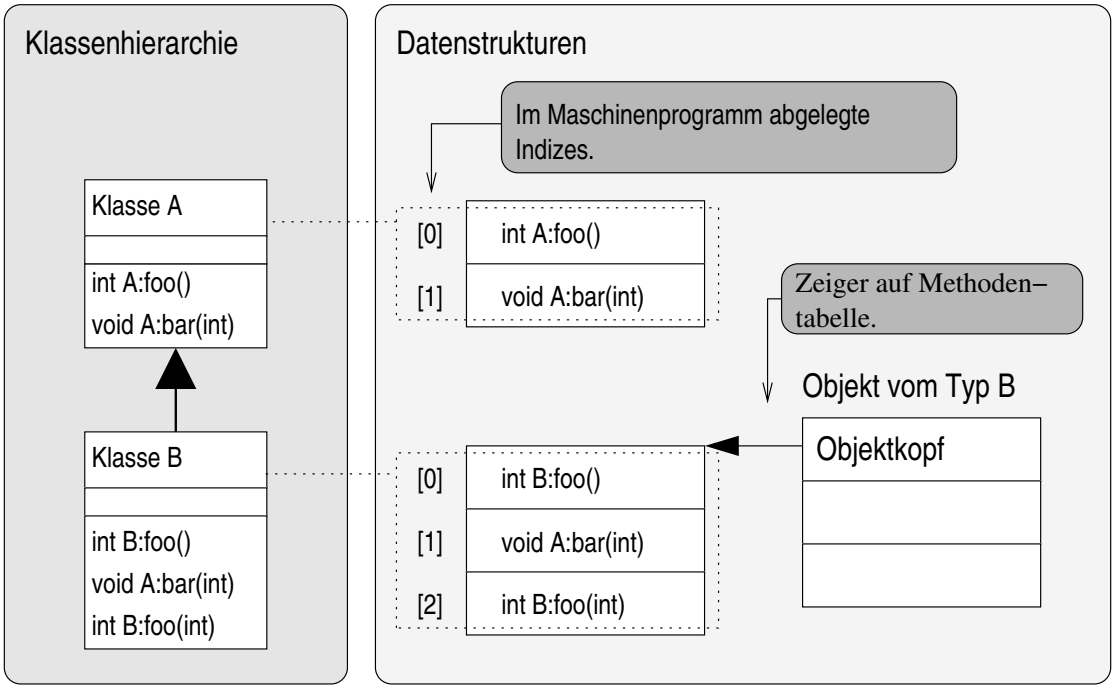

Abbildung 4.10: Vertikale Methodentabelle (VT)

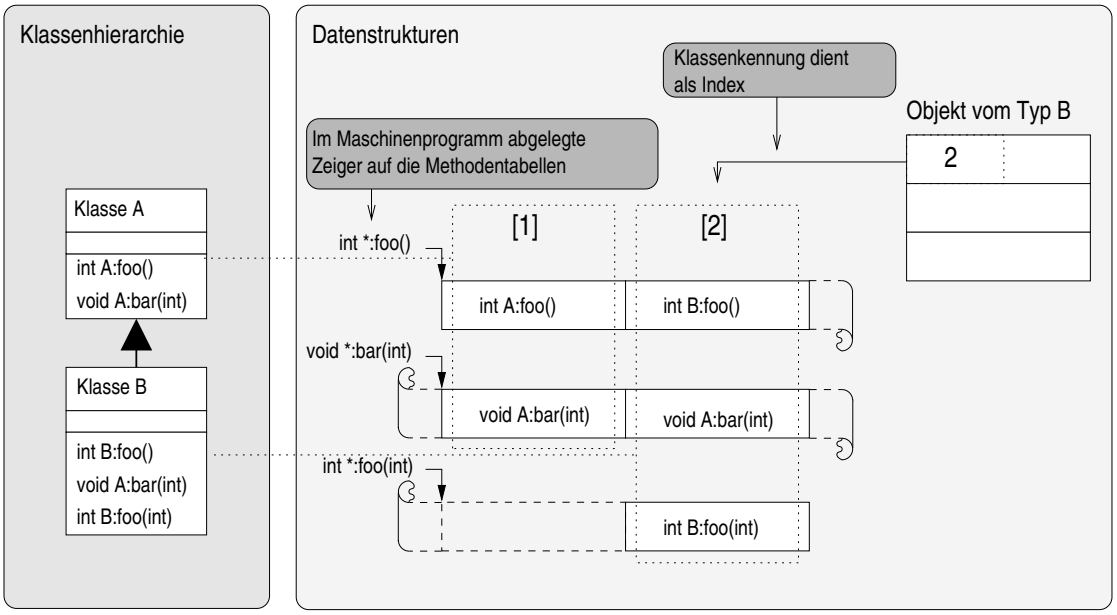

Abbildung 4.11: Horizontale Methodentabelle (HT)

sich auf eine zusätzliche Adresse für die Methodentabelle in jedem Objektkopf und einer Methodentabelle für jede Klasse.

Um nun keinen zusätzlichen Speicherplatz für den Zeiger auf die Methodentabelle im Objektkopf zu benötigen, wäre es möglich gewesen, den Zeiger auch im Klassenspeicher abzulegen. Dieses Verfahren würde jedoch einen weiteren Feldzugriff erfordern und einen zusätzlichen Eintrag im Klassenspeicher benötigen. Für KESO wurde daher ein anderer Aufbau für die Methodentabellen gewählt, der es ermöglicht, über die bereits im Objektkopf gespeicherte globale Klassenkennung die Methodenadresse direkt und ohne den Umweg über den Klassenspeicher zu bestimmen.

Um die beiden Arten von Methodentabellen im Folgenden besser unterscheiden zu können, wird die bereits beschriebene Tabelle als *vertikale Methodentabelle* (VT) bezeichnet und die in KESO verwendete Methodentabelle als *horizontale Methodentabelle* (HT).

Bei einer horizontalen Methodentabelle wird für jeden Methodentyp eine Tabelle aufgebaut und nicht, wie bei der VT, für jede Klasse. In der Tabelle bekommt jede Klasse, die den Methodentyp definiert oder erbt, einen Eintrag, wobei die Einträge nach der Klassenkennung sortiert werden.

Die Klassenkennung ist, wie in Kapitel 4.2.3 beschrieben, so gewählt, dass die Kennungen die Klassen fortlaufend und zusammenhängend aufzählen. Diese Eigenschaft weist die Klassenkennung auch für alle Teilbäume der Klassenhierarchie auf. Die Methoden eines Methodentyps besitzen eine Vererbungsbeziehung, daher ist die Klassenkennung auch für alle Methoden eines Methodentyps fortlaufend. Subtrahiert man nun von der Klassenkennung den Wert der Klassenkennung der ersten Klasse, die einen Methodentyp definiert, so erhält man einen für die Methodentabelle passenden fortlaufenden Index. Der aus der Klassenkennung berechnete Index adressiert dabei genau den Eintrag in der Tabelle, welcher die gültige Adresse für den Methodenaufruf enthält.

Die Berechnung vom Index entfällt zur Laufzeit, da der C-Übersetzer den konstanten Offset zum Übersetzungszeitpunkt bereits vom ebenfalls konstanten Zeiger auf den Tabellenanfang abziehen kann. Der Zeiger auf die Methodentabelle weist daher, wie in Abbildung 4.11 für die Methode `int foo(int)` dargestellt, auf eine Speicherstelle vor der eigentlichen Methodentabelle. Die Methodentabellen werden von JINO in einer einzigen großen Methodentabelle zusammengefasst und so zusammenhängend im Arbeitsspeicher abgelegt. Dieses Vorgehen ermöglicht es, die Adressen auf Methodentabellen von anderen Adressen zu unterscheiden, es ist jedoch ansonsten nicht zwingend notwendig.

Über die im Objektkopf gespeicherte Klassenkennung ist es nun möglich, mit einem zur vertikalen Methodentabelle vergleichbaren Aufwand einen virtuellen Methodenaufruf auszuführen. Anstelle des im Programm fest kodierten Index in die Methodentabelle und dem aus dem Objektkopf stammenden Zeiger auf den Anfang der Methodentabelle,

ist bei der HT nun der Index im Objektkopf gespeichert und der Zeiger auf die Tabelle fest im Programm kodiert. Beim Methodenaufruf wird durch einen Feldzugriff auf den Objektkopf der Index bestimmt und nicht der Zeiger. Alle anderen Operationen für den Methodenaufruf verlaufen identisch.

Der Speicherbedarf für die Tabellen ist in beiden Fällen gleich. Es wird für jeden Methodentyp und jede Klasse genau ein Eintrag in der Tabelle benötigt. Der Speicherbedarf im Objektkopf bzw. im Maschinenprogramm ist dagegen unterschiedlich. Während der Zeiger im Objektkopf abhängig von der Architektur — 32 Bit beim TriCore oder 16 Bit beim AVR — benötigt, kann die Klassenkennung in Abhängigkeit von der Anzahl der Klassen auch kleiner gewählt werden — 8 bis 16 Bit.

4.4.3 Virtuelle Methodenaufrufe über Schnittstellen

Die VT und HT wurden mit der Einschränkung eingeführt, dass die Klassenhierarchie nur eine Einfachvererbung aufweist. Java kennt bei Klassen, im Gegensatz zu anderen objektorientierten Programmiersprachen, keine Mehrfachvererbung, was die fortlaufende Klassenkennung bei der HT und die fortlaufende Anordnung der Methoden bei der VT erst ermöglicht. Java bietet jedoch eine Mehrfachvererbung bei den Schnittstellen. Die VT und die HT können beide auf die Mehrfachvererbung bei Schnittstellen erweitert werden.

In der SableVM [39] wurde gezeigt, wie dies für die VT effizient gelöst werden kann. Bei der Umsetzung wird eine wichtige Eigenschaft der reinen Schnittstellenvererbung ausgenutzt. Die Tatsache, dass Schnittstellen selbst keine Methoden implementieren, führt dazu, dass trotz Mehrfachvererbung immer nur genau eine Implementierung für eine Methode pro Klasse existieren kann. Es ist nun möglich, eine Tabelle für alle Methoden, die von Schnittstellen definiert werden, gemeinsam zu verwalten, in der es für jeden Methodentyp exakt einen Eintrag mit genau einem festen Index gibt. Es kann hier sogar die Methodentyp-Definition noch aufgeweicht werden, da keine Vererbungsbeziehungen betrachtet werden müssen.

Für jede Klasse werden nun zwei Tabellen erzeugt, zum einen die bereits beschriebene Methodentabelle für die Methoden der Klasse, zum anderen eine Schnittstellentabelle, die für alle in einer Schnittstelle definierten Methoden einen Eintrag mit einem für den Methodentyp eindeutigen Index enthält. Die Schnittstellenmethoden werden dabei zweimal gespeichert, einmal in der Methodentabelle der Klasse und einmal in der Schnittstellentabelle der Klasse. Implementiert eine Klasse eine Schnittstelle nicht, so werden die Positionen in der Schnittstellentabelle nicht genutzt und es entstehen daher auch ungenutzte Speicherbereiche.

Die zwei Tabellen können im Speicher so angeordnet werden, dass nur ein Zeiger im Objektkopf benötigt wird. Dafür baut man die Tabellen so auf, dass sie in unterschiedliche

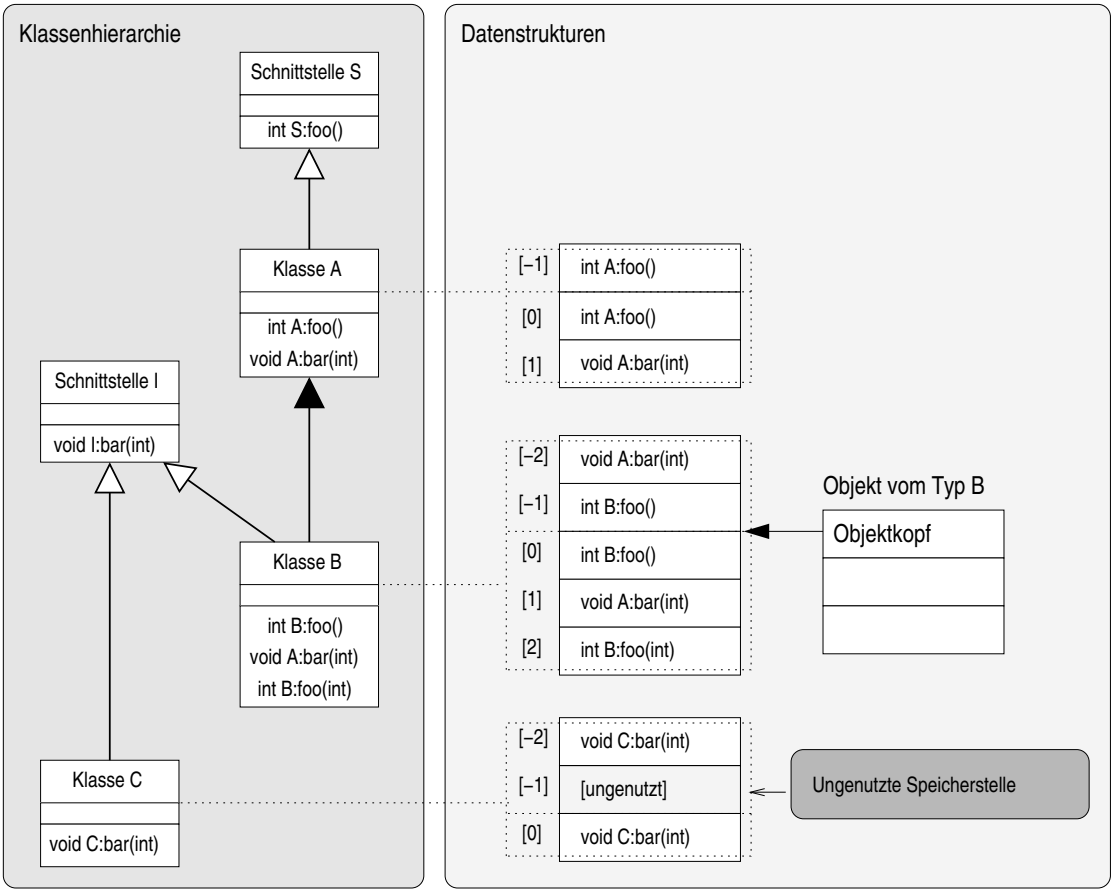

Abbildung 4.12: Bidirektionale vertikale Methodentabelle (BVT)

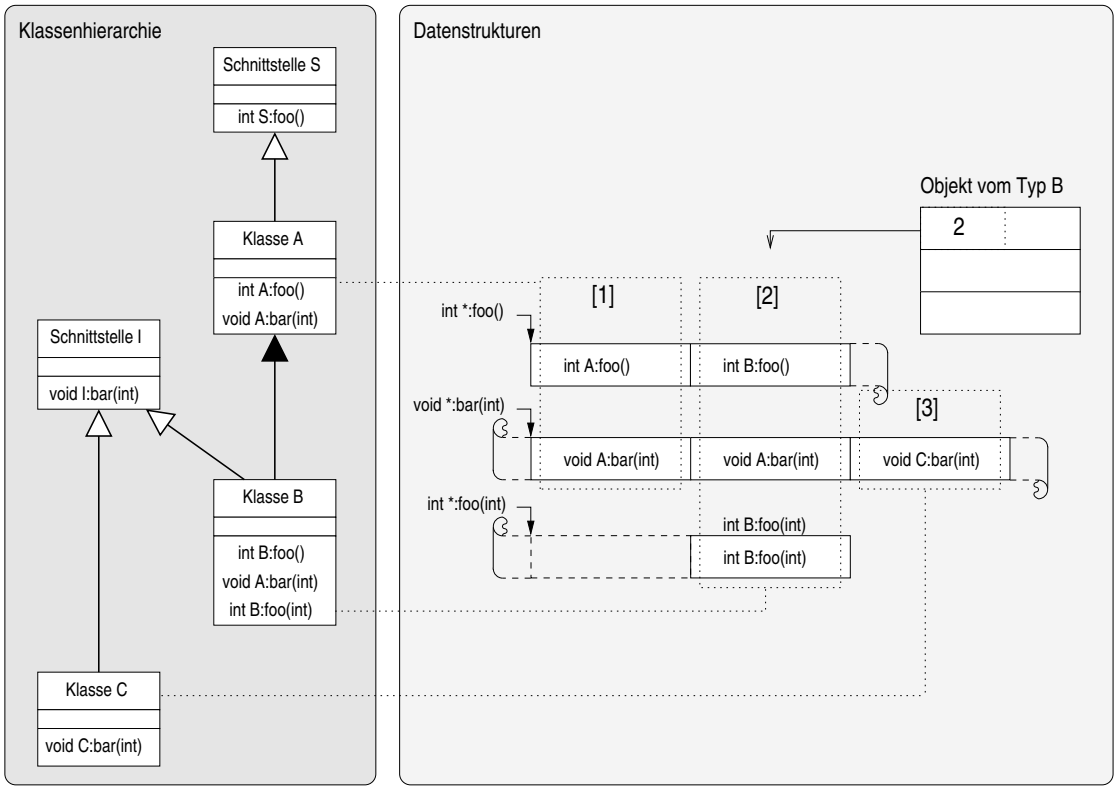

Abbildung 4.13: Erweiterte horizontale Methodentabelle (EHT)

Richtungen wachsen können. Eine Tabelle bekommt, ausgehend von der Startadresse, die negativen Indizes und die andere Tabelle die positiven Indizes. Die Abbildung 4.12 zeigt eine solche *bidirektionale vertikale Methodentabelle* (BVT), wie sie in der SableVM [39] und auch in anderen JVMs wie z.B. in JX zum Einsatz kommen. Der Speicherbedarf für die Methodentabelle steigt bei diesem Ansatz, da jede Schnittstellenmethode mindestens zwei Einträge benötigt, außerdem kommt es zu ungenutzten Einträgen in der Schnittstellentabelle, wie es in der Abbildung 4.12 dargestellt ist.

Der große Vorteil einer BVT ist jedoch, dass sie beim Laden von weiteren Klassen weiter wachsen kann, ohne dass bereits bestehende Tabellen angepasst werden müssen. Die HT besitzt diese Eigenschaft nicht. Sie kann jedoch noch kompakter auf die Schnittstellen erweitert werden.

Die Eigenschaft, dass Schnittstellen keine eigene Implementierung mitbringen, kommt der horizontalen Methodentabelle direkt zugute. Die Einträge in der Methodentabelle sind bereits nach dem Methodentyp sortiert. Es ist daher ausreichend, die Definition vom Methodentyp auf die Schnittstellen auszuweiten und eine gemeinsame Tabelle anzulegen

für alle Methoden, die einen gemeinsamen Methodentyp implementieren. Die Anordnung geschieht wieder nach der Reihenfolge der Klassenkennung, so dass die Kennung wieder als Index dienen kann. Die Tabelle kann nun sowohl für einen Methodenaufruf an einer Klasse als auch für einen Methodenaufruf an einer Schnittstelle verwendet werden. Die Abbildung 4.13 zeigt eine solche Tabelle.

Jede Methodenadresse benötigt in der *erweiterten horizontalen Methodentabelle* (EHT) nur genau einen Eintrag. Wie bei der BVT kann es jedoch auch bei der EHT zu leeren Einträgen in der Tabelle kommen, da die Eigenschaft der fortlaufenden und zusammenhängenden Klassenkennung über alle Klassen, die eine gemeinsame Schnittstelle implementieren, nicht gegeben ist. In Abbildung 4.14 wird eine Klassenhierarchie gezeigt, die zu einer Methodentabelle mit leeren Einträgen (A) führt und eine andere (B), die diese leeren Einträge nicht besitzt. Dabei hängt die Anzahl der leeren Einträge von der Anordnung der Klassen in der Klassenhierarchie ab. Durch die in Kapitel 4.2.3 beschriebene Sortierung der Klassenhierarchie werden Klassen, die eine Schnittstelle implementieren, möglichst zusammenhängend angeordnet und die Anzahl leerer Einträge in der Methodentabelle wird so verringert.

Der Speicherbedarf für die EHT fällt geringer aus als der für die BVT. Zum einen gibt es für jede Methode immer nur genau einen Eintrag und nicht wie bei der BVT im Falle von Schnittstellenmethoden zwei Einträge. Zum anderen ist der Speicherbedarf für den Index im Objektkopf geringer, welcher auch noch direkt zur Typüberprüfung verwendet werden kann. Virtuelle Methodenaufrufe verbrauchen jedoch weiterhin deutlich mehr Ressourcen als statische Methodenaufrufe. Daher wurden noch weitere Implementierungen entwickelt, um den Ressourcenbedarf noch weiter zu senken.

4.4.4 Auflösen von Sprungzielen zum Übersetzungszeitpunkt

Virtuelle Methodenaufrufe besitzen einen höheren Speicherbedarf und benötigen mehr Rechenzeit als einfache statische Methodenaufrufe. Darüber hinaus behindern virtuelle Methodenaufrufe die globale Analyse und erschweren so weitere Optimierungsschritte, da bei der Analyse zum Übersetzungszeitpunkt mehrere Sprungziele existieren.

Bei einigen virtuellen Methodenaufrufen besteht die Möglichkeit, das Sprungziel bereits zum Übersetzungszeitpunkt zu bestimmen. Bekannte und effiziente Techniken basieren auf der Analyse der Klassenhierarchie. Dies wird in der englischsprachigen Literatur als *Class Hierarchy Analysis* (CHA) bezeichnet [20, 34, 37]. Bei diesem Verfahren wird ausschließlich das Wissen über die Klassenhierarchie genutzt, um die Anzahl der Sprungziele einzuschränken, mit dem Vorteil, dass eine aufwändige Programmanalyse entfällt.

Durch eine vereinfachte Erreichbarkeitsanalyse kann die Klassenhierarchie zusätzlich noch eingeschränkt werden, wodurch das Resultat der CHA mit geringem Aufwand

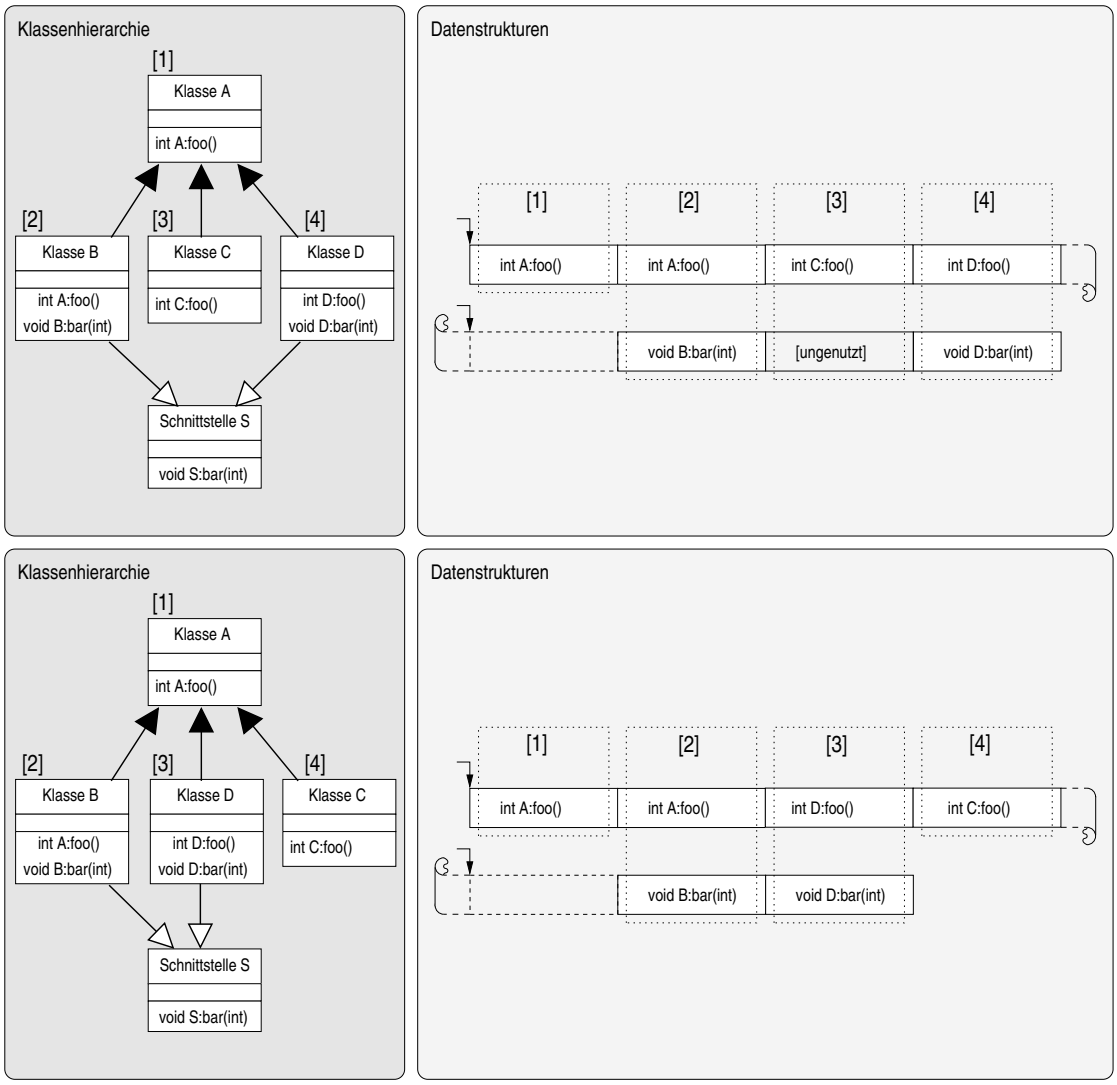

Abbildung 4.14: Vermeidung von leeren Einträgen in der EHT

verbessert wird [20]. Dabei werden zur Bestimmung möglicher Sprungziele bei der CHA nur noch die Klassen berücksichtigt, welche ausgehend vom Aufrufgraphen der Anwendung auch tatsächlich instanziiert werden. Das Auffinden instanziierter Klassen wird in der englischsprachigen Literatur als *Rapid Type Analysis* (RTA) bezeichnet.

Beim Aufbau der horizontalen Methodentabellen werden alle potenziellen Kandidaten für einen Methodentyp bereits gesucht. Daher liegt die Information, ob es mehr als einen Kandidaten gibt, bereits vor. Mit dieser Information werden entsprechende virtuelle Methoden durch direkte Aufrufe ersetzt und für diese Methoden werden auch keine Methodentabellen erzeugt. Auch wird die Klassenhierarchie durch das in Kapitel 5.3 beschriebene Verfahren auf die wirklich instanziierten Klassen eingeschränkt. Das in JINO verwendete Verfahren ist daher mit der RTA-Technik vergleichbar, mit dem Unterschied, dass die Information nicht nur für die Optimierung von virtuellen Methodenaufrufen verwendet, sondern auch bei der Reduktion des Klassenspeichers genutzt wird.

Darüber hinaus berücksichtigt der Übersetzer die Methoden anschließend auch als mögliche Kandidaten für die in Kapitel 5.4.2 beschriebene In-Line-Expansion.

4.4.5 Alternative Implementierung

Bei Mikrocontrollern kommt oft eine Harvard-Architektur zum Einsatz. Das bedeutet, dass es je einen separaten Speicher für das Maschinenprogramm und für die Daten gibt. Ein Grund für die Verwendung von Harvard-Architekturen liegt in den unterschiedlichen Anforderungen an den Speicher. Der Programmspeicher wird nur zum Zeitpunkt der Softwareinstallation beschrieben und sollte seinen Inhalt auch bei einem Verlust der Stromversorgung nicht verlieren. Für den Programmspeicher kommen daher Flash-Speicher bzw. EPROMs zum Einsatz. Als Datenspeicher dient hingegen meistens SRAM.

SRAM hat im Vergleich zu dem wesentlich platzsparenderen DRAM den Vorteil, dass dieser mit der gleichen Taktrate wie der Prozessorkern betrieben werden kann und keine aufwändige Puffer- und Steuerlogik benötigt. SRAM selbst benötigt jedoch mehr Platz auf dem Chip als DRAM, Flash oder EPROMs, was dazu führt, dass ein großer Teil der Fläche eines Mikrocontrollers für den Datenspeicher benötigt wird und dieser Speicher den Preis für den Mikrocontroller bestimmt.

Der Bedarf an Datenspeicher ist damit der entscheidende Kostenfaktor. Um Kosten zu sparen, sollten die Methodentabellen daher nicht im teuren SRAM liegen, sondern wenn möglich im billigeren Programmspeicher. Die Harvard-Architektur einiger Mikrocontroller verhindert jedoch, dass Daten aus dem Programmspeicher überhaupt oder hinreichend schnell gelesen werden können. Daher wurde eine alternative Umsetzung für virtuelle Methodenaufrufe entwickelt, die JINO bei Bedarf erzeugen kann, welche keine Methodentabelle benötigt.

Bei einem virtuellen Methodenaufruf wird eine Auswahl aus verschiedenen Implementierungen des gleichen Methodentyps in Abhängigkeit vom Objekttyp getroffen. Eine solche Auswahl kann natürlich auch mit einer bedingten Verzweigung im Maschinenprogramm als Alternative für eine Tabelle mit Startadressen erfolgen. Der Übersetzer erzeugt zu diesem Zweck nur eine Methode für alle alternativen Implementierungen des Methodentyps. Diese Methode verzweigt nach dem Aufruf über eine `switch`-Anweisung, in Abhängigkeit von der Klassenkennung in die unterschiedlichen Implementierungen.

Die virtuellen Methodenaufrufe werden durch einen direkten Aufruf zu dieser umfassenden Methode ersetzt. Für den Speicherbedarf im Programmspeicher ist es wichtig, dass die Auswahl der Implementierungen erst auf der Zielseite des Aufrufes erfolgt, da ansonsten die `switch`-Anweisung für jeden Aufruf erzeugt würde, was das Maschinenprogramm aufblähen würde.

Die `switch`-Anweisung wird je nach Architektur vom C-Übersetzer unterschiedlich umgesetzt. Einige Prozessorarchitekturen besitzen indirekte Sprunganweisungen, mit denen es möglich ist, die Auswahl über eine im Programmspeicher abgelegte Tabelle zu implementieren. Bei Architekturen ohne passenden Maschinenbefehl erzeugt der C-Übersetzer eine Kaskade von bedingten Sprüngen. Bei einer Kaskade von bedingten Sprüngen ist die Ausführungsdauer nicht mehr konstant und damit schlecht vorhersagbar, auch ist der Speicherbedarf im Programmspeicher in diesem Fall deutlich größer. Die virtuellen Methodenaufrufe auf eine `switch`-Anweisung abzubilden ist daher nicht für alle Anwendungsfälle ideal. Dem Anwender steht mit dieser Variante jedoch eine sehr Arbeitsspeicher sparende Alternative zur Verfügung, die im Bedarf genutzt werden kann.

4.4.6 Kompression von Methodentabellen

Wenn sich Methodentabellen nicht vermeiden lassen und diese eine inakzeptable Größe erreichen, bietet JINO die Möglichkeit, die Tabelle zu komprimieren.

Für die Komprimierung wird mit Hilfe einer Indextabelle eine weitere Indirektionsstufe, wie in Abbildung 4.15 dargestellt, eingeführt. Wenn die Anzahl der unterschiedlichen Methodenadressen kleiner ist als die Anzahl der Unterklassen und diese in einer separaten Adresstabelle gespeichert werden, so hat die Adresstabelle eine geringere Anzahl von Einträgen mit der vollen Adressbreite von 16 oder 32 Bit. Über eine Indextabelle mit einer kleineren Datenbreite, von 8-16 Bit, wird die Klassenkennung auf den Index für die Adresstabelle abgebildet. Der Speicherbedarf besteht nun aus dem Speicherbedarf für die Adresstabelle und aus dem Speicherbedarf für die Indextabelle.

Die Indextabelle verringert bereits den Speicherbedarf für die Methodentabelle, wenn es deutlich weniger Adressen gibt und der Speicherbedarf für die Indextabelle und die Adresstabelle zusammen aufgrund der unterschiedlichen Datenbreiten geringer ausfällt

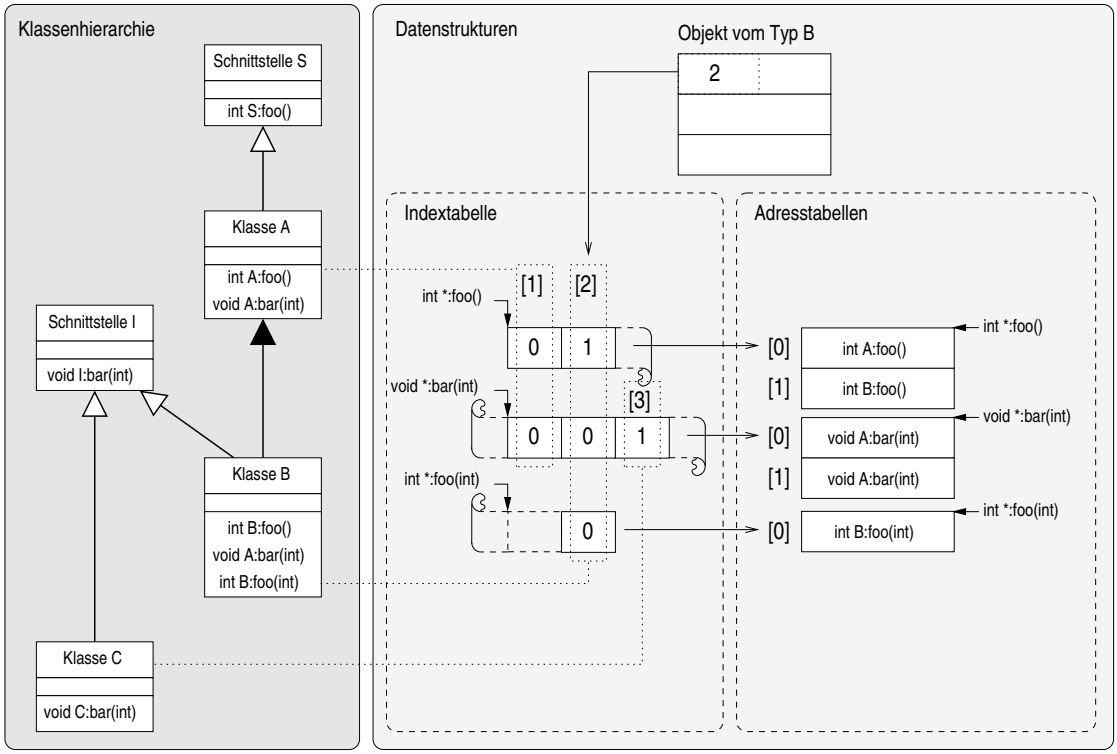

Abbildung 4.15: Indizierte horizontale Methodentabelle (IEHT)

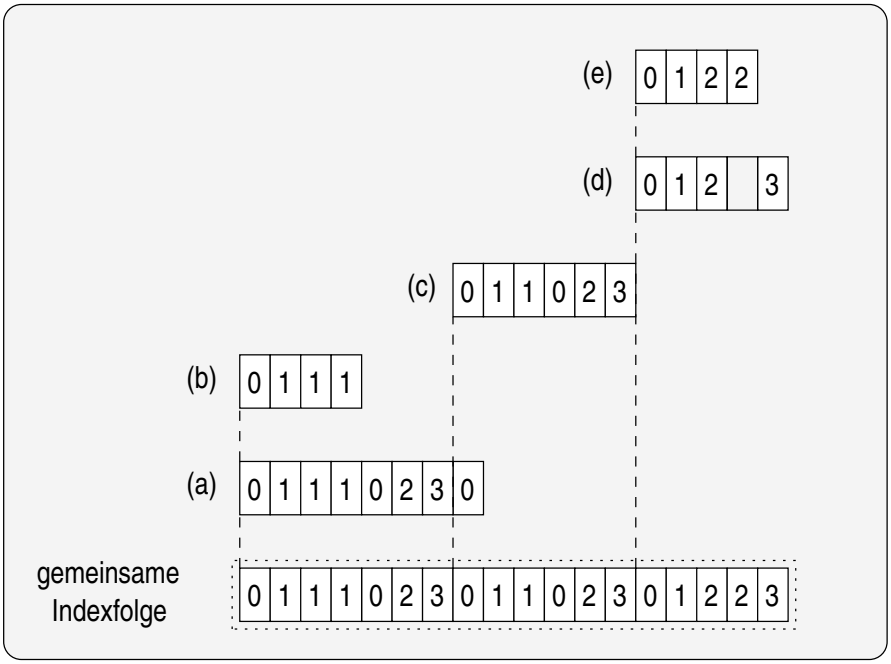

Abbildung 4.16: Gemeinsam genutzte Indextabelle

als für die EHT. Bei der Betrachtung der Indexfolgen ist zu erkennen, dass es noch weitere Möglichkeiten zur Verringerung des Speicherbedarfs gibt. In Abbildung 4.15 ist zu sehen, dass die Indexfolgen für die Methoden `int foo()` und `int foo(int)` auch in der Indexfolge von `void bar(int)` vorkommen. Daher ist es möglich, die Indexfolge von `void bar(int)` für die anderen Methodentabellen wiederzuverwenden.

Zu diesem Zweck erzeugt JINO eine gemeinsam genutzte Indexfolge für alle Methodentabellen. Abbildung 4.16 veranschaulicht die Erzeugung an Hand von aufwändigeren Indexfolgen. Für jede benötigte Indexfolge wird in der bereits existierenden Indexfolge nach einer Übereinstimmung gesucht. Gibt es keine Übereinstimmung, wie in den Fällen (a) und (d), so wird die Indexfolge einfach an die bestehende Indexfolge angehängt. Die Indexfolge (b) ist komplett in (a) enthalten und vergrößert daher die gemeinsam genutzte Indexfolge nicht. Die Indexfolge (c) kann zumindest noch das Ende der Indexfolge nutzen. Die leeren Einträge, die in der EHT durch die Verwendung von Schnittstellen entstehen, können in der Indexfolge beliebige Werte akzeptieren. Die Indexfolge (e) nutzt im Beispiel eine leere Stelle aus der Indexfolge (d). Die leeren Einträge werden so durch andere Indexfolgen aufgefüllt und benötigen keinen zusätzlichen Speicher mehr.

Die Verwendung von bestehenden Indexfolgen erhöht zwar den Kodierungsaufwand und somit die Übersetzungszeit, aber der zur Laufzeit der Anwendung anfallende Dekodieraufwand bleibt konstant. Die Abbildung 4.17 zeigt das Beispiel aus der Abbildung 4.15

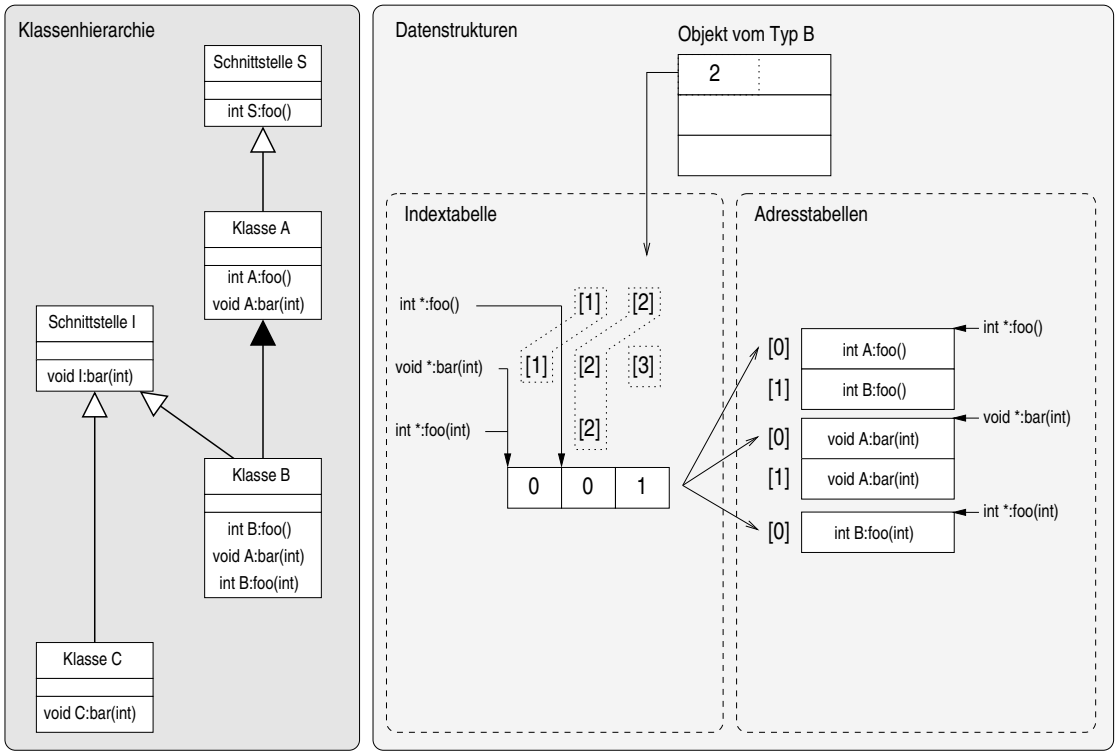

Abbildung 4.17: Komprimierte horizontale Methodentabelle (CEHT)

mit einer gemeinsam genutzten Indexfolge. Auf einem Mikrocontroller mit 32-Bit-Adressen und acht Bit großen Einträgen braucht die *Komprimierte EHT* (CEHT) nur noch 23 Byte im Vergleich zur BVT in der Abbildung 4.12 mit 44 Byte. Das ist eine Einsparung von 48%. Auf einem 16-Bit-Mikrocontroller liegt das Verhältnis bei 22 zu 13 Byte und daher noch bei einem 40% geringeren Speicherbedarf.

Eine aufwändige Kompression der EHT wird erst bei größeren Anwendungen interessant, da erst bei einer größeren Klassenhierarchie auch eine größere Anzahl von Indizes wiederverwendet wird. Bei den in Kapitel 6 beschrieben Anwendungen wurde die Kompression der EHT nicht genutzt, da die Methodentabellen hier komplett vermieden wurden oder nur wenige Byte im Speicher belegten.

4.5 Zusammenfassung

Die von JINO erzeugten Datenstrukturen sind auf einen möglichst geringen Speicherbedarf ausgerichtet und orientieren sich an dem Bedarf der Anwendung. Der Aufbau von

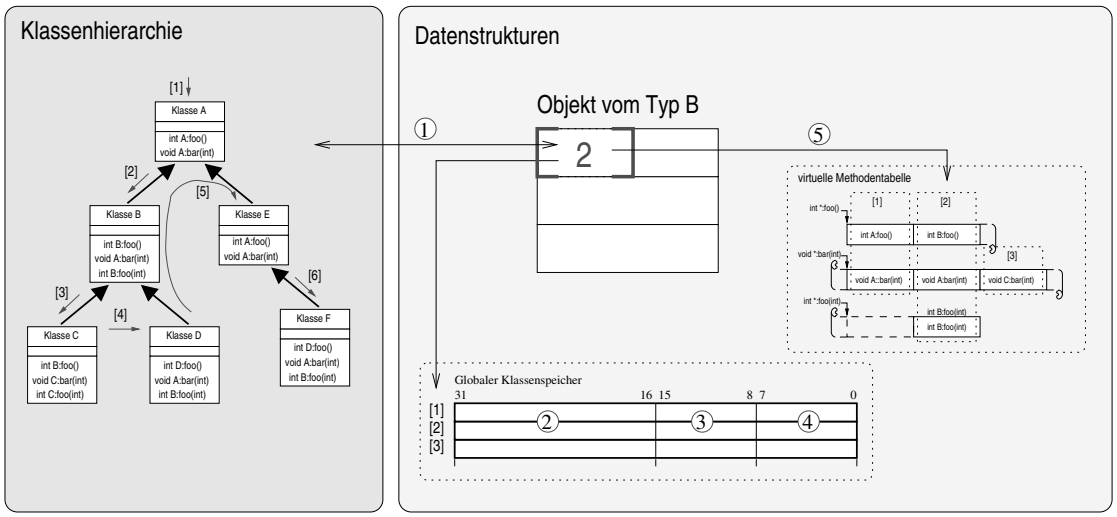

Abbildung 4.18: Die Bedeutung der Klassenkennung

Objekten als zentrales Element ist durch eine bidirektionale Datenstruktur gelöst, bei der alle Objektreferenzen immer fortlaufend angeordnet sind. Diese Anordnung ermöglicht es, dass nur die Anzahl der Objektreferenzen benötigt wird, um die Objektreferenzen bei einer Speicherbereinigung effizient finden zu können.

Der Objektkopf liegt von der Referenz aus gesehen immer an der gleichen Position im Objekt, so dass er immer korrekt interpretiert werden kann. Er enthält mindestens die Klassenkennung, daher ist ein minimales Objekt mindestens ein Byte groß.

Die Klassenkennung kodiert den Objekttyp und kann direkt für eine primäre Typüberprüfung genutzt werden. Die Kennung dient als direkter Index für den Klassenspeicher. Im Klassenspeicher sind weitere Information über das Objekt hinterlegt, wie die Objektgröße, die sekundäre Typinformation und die Anzahl der im Objekt gespeicherten Referenzen. Die Klassenkennung dient auch als Index für die virtuelle Methodentabelle, so dass eine Auflösung der Methodenadresse mit nur einem Speicherzugriff möglich ist. Auf diese Weise kodiert die Klassenkennung im Objektkopf, wie in Abbildung 4.18 noch einmal dargestellt, alle benötigten Objektattribute mit nur ein bis zwei Byte pro Objekt.

1. Der **primäre Typ** des Objektes wird direkt durch den Wert der Klassenkennung kodiert.

2. Die **Objektgröße** ist bei Bedarf in einem Klassenspeicher hinterlegt und die Klassenkennung dient als direkter Index zum Auffinden des passenden Eintrags.

3. Die Information über den **Sekundären Typ** kann auch im Klassenspeicher hinterlegt werden, oder alternativ in einer separaten Tabelle, wobei die Klassenkennung

erneut als Index dient.

4. Die **Anzahl an Objektreferenzen** ist im Bedarfsfall auch im Klassenspeicher gespeichert oder kann alternativ über eine Generatorfunktion

   aus der Klassenkennung erzeugt werden.

5. Bei virtuellen Methodenaufrufen dient die Klassenkennung als **Index für die Methodentabelle**.

Kapitel 5

Der Übersetzungsprozess

5.1 Überblick

JINO bekommt, wie in Abbildung 5.1 dargestellt, als Eingabe eine Konfigurationsdatei mit der Endung kcl, was für *keso configuration language* (kcl) steht. In der Konfiguration werden die benötigten Bibliotheken und Anwendungen als Module referenziert.

Ein Modul ist ein Verzeichnis oder eine Archivdatei im ZIP-Dateiformat. Die Module müssen eine META-Datei enthalten und mögliche Programmdaten in Form von Java-Quelldateien oder bereits übersetzten Java-Klassendateien. Die META-Datei enthält Zusatzinformationen, wie die Namen von benötigten Modulen. Aufgrund dieser Zusatzinformationen ist es ausreichend, das Modul, welches die Programmdaten der Anwendung enthält, in der Konfiguration anzugeben, und die von der Anwendung benötigten Module werden automatisch mit ausgewählt.

In der Konfigurationsdatei kann eine ganze Gruppe von Mikrocontrollern und deren Konfiguration beschrieben werden, wobei die Gruppe der Mikrocontroller als `Cluster` und jeder Mikrocontroller als `Node` bezeichnet wird. Abbildung 5.2 zeigt, wie eine Konfiguration mit zwei unterschiedlichen Mikrocontrollern aussehen kann. **KnotenA** ist die Beschreibung für einen TriCore TC1796 und **KnotenB** für einen AVR ATmega8535. Die Zielarchitektur der Knoten wird mit dem `Target`-Attribut festgelegt.

Aus der Konfigurationsdatei und den Modulen generiert JINO automatisch eine für das OSEK/VDX-System benötigte Konfigurationsdatei in Form einer OIL Beschreibung, mehrere C-Dateien und ein passendes Makefile für den abschließenden Übersetzungsvorgang durch den C-Übersetzer. Da für die verwendeten Zielarchitekturen unterschiedliche Übersetzer zum Einsatz kommen und die erzeugten C-Quellen aufgrund der Anpassung an die jeweilige Anwendung unterschiedlich ausfallen, wird für jeden Knoten ein eigenes Ausgabeverzeichnis angelegt.

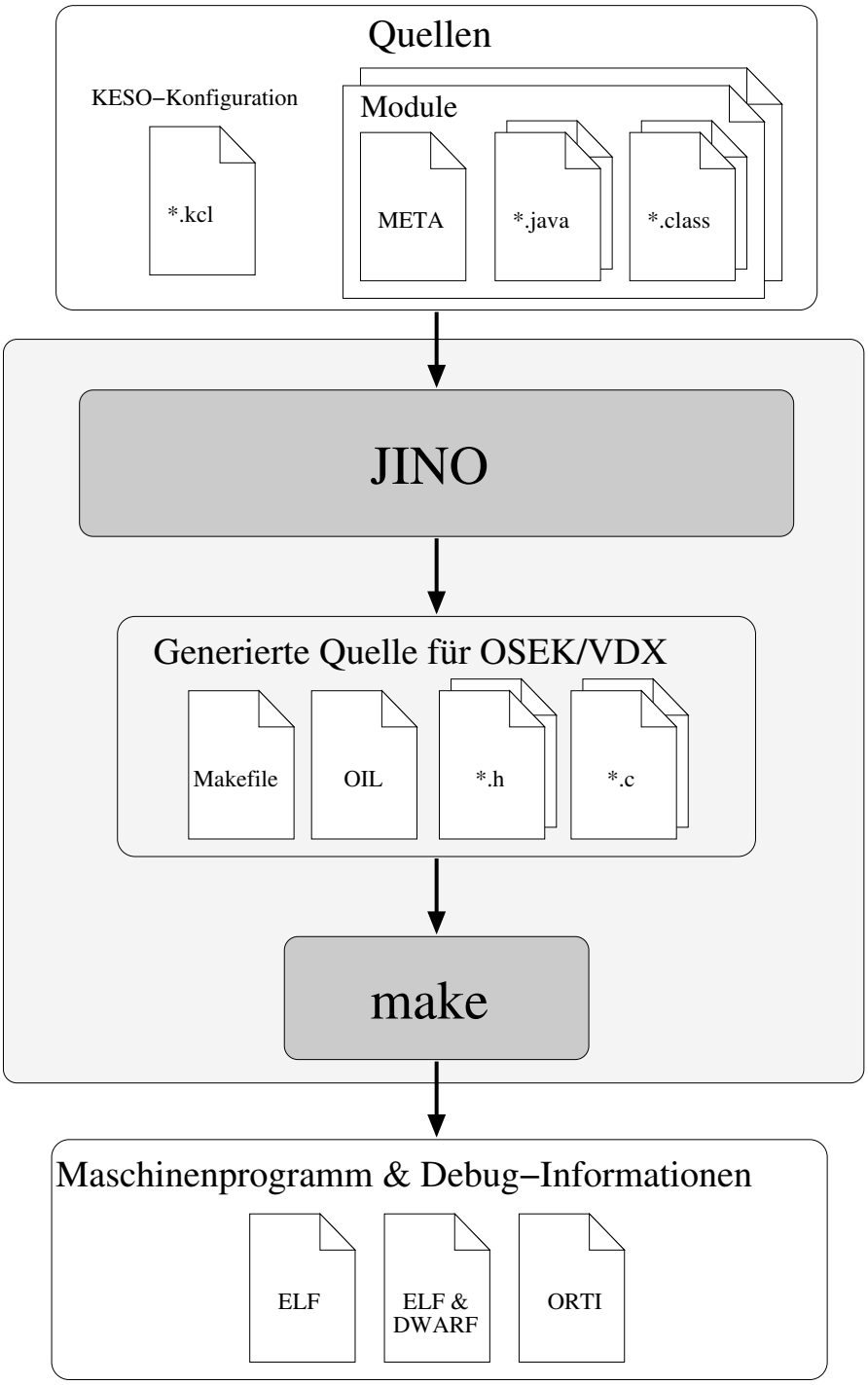

Abbildung 5.1: Überblick über den Übersetzungsprozess

```
Cluster (test) {

        Network (CanBus) {
                # ...
        }

        Node (KnotenA) {

                Target = "Tricore";
                ProcessorType = "tc1796";
                Modules = "test_com_tricore";

                Domain (domain0) {
                        # ...
                }
        }

        Node (KnotenB) {

                Target = "AVR";
                ProcessorType = "atmega8535";
                Modules = "test_com_avr";

                Domain (domain0) {
                        # ...
                }
        }
}
```

Abbildung 5.2: Beispiel für eine KESO Konfiguration mit mehreren Mikrocontrollerknoten

Im letzten Arbeitsschritt startet JINO für jedes Ausgabeverzeichnis einen make-Prozess, der sowohl die Generierung von OSEK/VDX als auch die Übersetzung in das Maschinenprogramm übernimmt. Als endgültiges Ergebnis entstehen ein Maschinenprogramm für jeden Mikrocontroller und — je nach Architektur — noch zusätzliche Dateien mit Debug-Informationen, wie zum Beispiel eine *OSEK run time interface* (ORTI), oder Grafiken zum Kontrollfluss und zur Klassenhierarchie der Anwendung.

5.2 Vorverarbeitungsphase

Die Aufgliederung der Quellen in Module kann bereits als erste Maßnahme betrachtet werden, um die Größe von Bibliotheken und damit den Ressourcenbedarf der Anwendung einzuschränken, da nur die Module verwendet werden, die in der Konfiguration auch ausgewählt wurden. JINO besitzt jedoch Analysetechniken, die den Programmumfang während des Generierungsprozesses auf die von der Anwendung verwendeten Klassen und Methoden einschränkt.

Die Module reduzieren jedoch den Aufwand für die Übersetzung bereits in der Vorverarbeitungsphase und ermöglichen es zusätzlich noch, mehrere unterschiedliche Versionen eines Moduls zu verwalten und diese durch die Konfiguration und ohne Änderung der Anwendung auszuwählen. In der Vorverarbeitungsphase werden Klassendateien aus allen verwendeten Modulen zusammengeführt. Klassen, die als Quelldateien vorliegen, werden mit einem voreingestellten Java-Übersetzer zu Klassendateien übersetzt.

5.3 Laden der Klassen

Nach der Vorverarbeitungsphase lädt JINO die Klassendateien und wandelt diese in eine interne Zwischendarstellung um. Beim Laden der Klassen wird gleichzeitig das Verhalten eines dynamischen Klassenladers simuliert, wie sie die JVM-Spezifikation vorsieht, und es wird eine Erreichbarkeitsanalyse durchgeführt, die die Anzahl der Klassen und der Methoden auf die durch die Anwendung erreichbaren Klassen und Methoden reduziert.

Die in der Konfiguration genannten Klassen und Methoden dienen für den Klassenlader als Ausgangspunkt für das Laden der Klassen und die Erreichbarkeitsanalyse. Ein Beispiel für einen solchen Ausgangspunkt ist die `launch`-Methode von einem Task. Alle in der Konfiguration angegebenen Ausgangspunkte werden als erreichbar angesehen, da diese durch das OSEK/VDX-System aufgerufen werden können.

Eine erreichbare Methode wird in die interne Sicht aufgenommen, indem zuerst die Klasse, welche die Methode definiert, und alle benötigten Oberklassen und Schnittstellen dieser Klasse in die interne Sicht aufgenommen werden. Alle von diesen Klassen

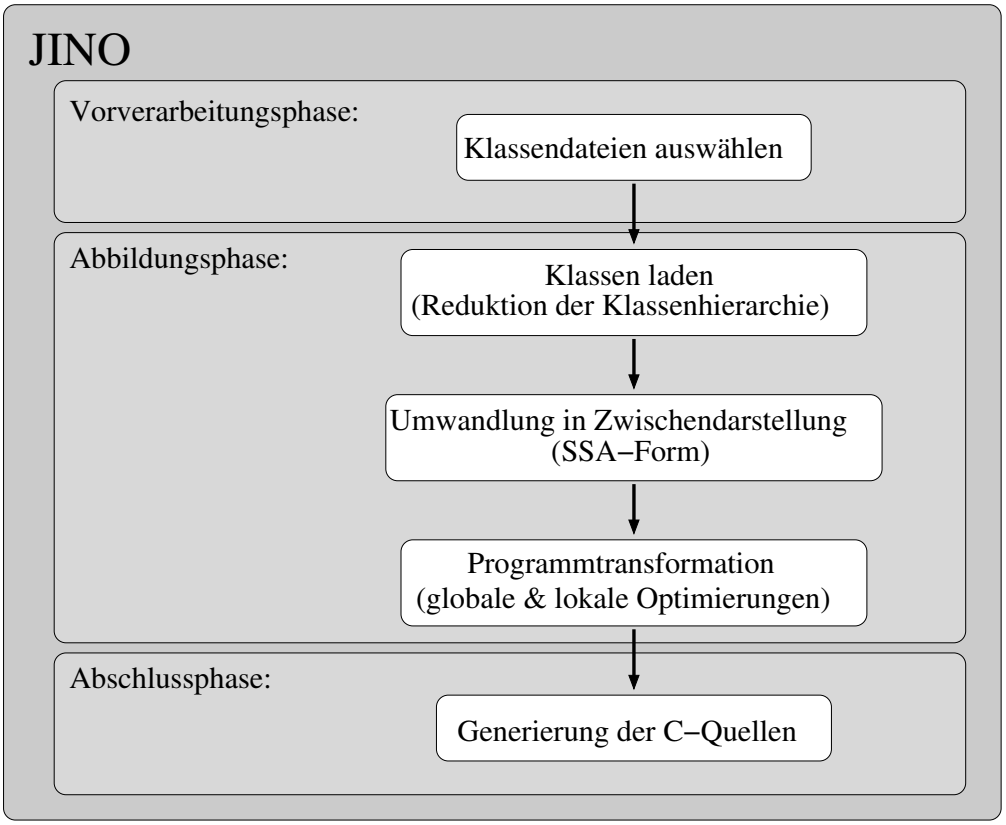

Abbildung 5.3: Überblick Aufbau von JINO

definierten Methoden werden nach dem Methodentyp getrennt registriert. Anschließend werden alle Methoden, die einen registrierten Methodentyp besitzen und deren Klasse bereits in der internen Sicht aufgenommen sind, auch in die interne Sicht aufgenommen. Anschließend wird der Bytecode der neu dazu gekommenen Methode gelesen, und beim Auftreten eines Methodenaufrufes wird der Vorgang wiederholt.

Werden zu einem späteren Zeitpunkt Klassen geladen, die Methoden vom gleichen Methodentyp implementieren, so werden diese ebenfalls als erreichbar angesehen und aufgenommen. Weitere Klassen werden geladen, wenn ein Feldzugriff, eine Typüberprüfung oder eine Objekterzeugung stattfindet, sofern eine noch unbekannte Klasse betroffen ist.

Der beschriebene Algorithmus berücksichtigt nicht den tatsächlich zur Laufzeit auftretenden Typ eines Objektes, sondern betrachtet nur den zum Übersetzungszeitpunkt bekannten Typ. Alle Klassen, die geladen werden und zum Typen des Objektes passen, werden als mögliche Kandidaten betrachtet. Daher werden möglicherweise mehr Methoden und Klassen als erreichbar betrachtet, als dies tatsächlich zur Laufzeit der Fall ist. Eine präzisere Typbestimmung würde jedoch zu diesem Zeitpunkt eine wesentlich aufwändigere Analyse erfordern.

Die Reduktion der Klassenhierarchie und der verwendeten Methoden führt zu einer drastischen Einsparung, sowohl beim Speicherbedarf für das Programm als auch beim Speicherbedarf für die Daten.

5.4 Die interne Zwischendarstellung

Nach dem Laden der Klassen wird der Bytecode in eine Zwischendarstellung überführt, auf der JINO anschließend weitere Analysen und Umformungen vornimmt und aus dem im Anschluss das C-Programm erzeugt wird. Um die Arbeitsweise von JINO besser zu veranschaulichen, soll das kleine Java Programm in Abbildung 5.4 dienen.

Der Java-Übersetzer erzeugt aus diesem Beispielprogramm zwei Klassendateien, eine für die Klasse `SimpleBCExample` und eine für die Klasse `SimpleClass`. Der in den Klassendateien enthaltene Bytecode kann mit Hilfe eines Disassemblers[1] in eine lesbare Form gebracht werden. Abbildung 5.5 zeigt den entsprechenden Bytecode in einer Bytecode Assemblersprache. Der Java Übersetzer hat für jede Klasse zwei Methoden erzeugt, und zwar `int increase(int,int)` und `void launch()` aus dem Beispiel und zusätzlich für jede Klasse noch eine Initialisierungsmethode. Die Methoden zum Initialisieren von Objekten werden automatisch erzeugt und bei einer Objekterzeugung auch implizit aufgerufen, solange keine andere Initialisierungsmethode spezifiziert wird. Die vier Methoden haben einen reinen Programmumfang von 52 Byte.

[1] Hierfür wurde der Java-Disassembler `javap` aus dem Java SDK verwendet.

```
package test;

import keso.core.Task;

class SimpleClass {
        int increase(int factor, int value) {
                if (factor>100) return value+100;
                return value+factor;
        }
}

public class SimpleBCExample extends Task {

        public void launch() {

                SimpleClass object = new SimpleClass();

                for (int a=0;a<10;a=object.increase(5,a)) ;
        }
}
```

Abbildung 5.4: Einfaches Programmbeispiel

Der Java-Bytecode unterscheidet sich vom Aufbau her von typischen Maschinenbefehlssätzen. Ein wesentlicher Unterschied besteht darin, dass die Bytecodebefehle ihre Operanden von einem Operandenstack beziehen und diese nicht direkt benennen, z.B. in Form von konstanten Werten, Registernamen oder über Speicheradressen.

Ermittelt man die zu einer Operation gehörenden Operanden beim Übersetzungsvorgang, so kann man auf den Operandenstack zur Laufzeit verzichten. Dies spart für sich genommen schon Speicherplatz und Rechenzeit, wesentlicher ist jedoch, dass der Operandenstack bei späteren Datenflussanalysen nur stören würde.

Die Zuordnung der Operanden geschieht bei der Umwandlung des Bytecodes in die interne Zwischendarstellung, auf der JINO anschließend weiterarbeitet. Für die Zwischendarstellung wird für jeden Befehl aus dem Bytecode ein Objekt erzeugt, welches den Befehl und seine Abhängigkeit repräsentiert. Diese Befehlsobjekte werden zuerst in eine Liste eingehängt.

Die Liste wird nach und nach in Basisblöcke unterteilt. Ein Basisblock repräsentiert dabei immer eine Befehlsfolge, die vom Kontrollfluss vollständig vom Anfang bis zum Ende durchlaufen wird. Sprungbefehle und Sprungziele bilden daher die Grenzen eines Basisblocks.

Die Basisblöcke werden in Abhängigkeit von den Sprüngen im Programm verkettet. Der so entstehende Graph wird als Kontrollflussgraph bezeichnet und repräsentiert alle

```
Compiled from "SimpleBCExample.java"
class test.SimpleClass extends java.lang.Object {
test.SimpleClass();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

int increase(int,int);
  Code:
   0:   iload_1
   1:   bipush  100
   3:   if_icmple       11
   6:   iload_2
   7:   bipush  100
   9:   iadd
   10:  ireturn
   11:  iload_2
   12:  iload_1
   13:  iadd
   14:  ireturn
}

Compiled from "SimpleBCExample.java"
public class test.SimpleBCExample extends keso.core.Task {
public test.SimpleBCExample();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method keso/core/Task."<init>":()V
   4:   return

public void launch();
  Code:
   0:   new     #2; //class SimpleClass
   3:   dup
   4:   invokespecial   #3; //Method test/SimpleClass."<init>":()V
   7:   astore_1
   8:   iconst_0
   9:   istore_2
   10:  iload_2
   11:  bipush  10
   13:  if_icmpge       26
   16:  aload_1
   17:  iconst_5
   18:  iload_2
   19:  invokevirtual   #4; //Method test/SimpleClass.increase:(II)I
   22:  istore_2
   23:  goto    10
   26:  return
}
```

Abbildung 5.5: Vom Java Übersetzer erzeugter Bytecode

möglichen Kontrollflüsse innerhalb einer Methode. Abbildung 5.6 und 5.7 zeigen für die beiden Methoden einen solchen Kontrollflussgraphen. Die Darstellungen wurde von JINO generiert. Den Inhalt der Basisblöcke stellt JINO dabei in einem C-ähnlichen Pseudo-Code dar, wie dieser nach der Zuordnung der Operanden aussieht.

5.4.1 Virtueller Operandenstack

Für die Zuordnung der Operanden wird ein virtueller Operandenstack verwendet. Dieser simuliert zum Übersetzungszeitpunkt die Arbeitsweise des in der JVM-Spezifikation beschriebenen Operandenstacks. Virtuelle Operandenstacks sind auch bei der Übersetzung von stackorientierten Zwischensprachen in Maschinenprogramme üblich [32, 98], um so eine Abbildung von einer stackorientierten Adressierung der Operanden auf die unterschiedlichen Adressierungsarten der Hardware zu finden. Im Gegensatz zu diesen bildet der virtuelle Operandenstack von JINO jedoch nicht auf Zwei- oder Dreiadressbefehle ab, sondern geht noch weiter und bildet stattdessen auf komplette Syntaxbäume ab. Abbildung 5.8 veranschaulicht an einem Beispiel die Funktionsweise des virtuellen Operandenstacks, wie er in JINO umgesetzt wurde.

Die Befehlsfolge, die im Beispiel dargestellt ist, besteht aus den Befehlen `iconst_1`, `iload_1`, `iadd` und `store_2`. Die Namen der Bytecodebefehle stammen aus der JVM-Spezifikation [66]. Laut Spezifikation legt `iconst_1` den konstanten Wert `1` auf den Operandenstack, der Befehl `iload_1` legt den Inhalt der zweiten lokalen Variable auf den Operandenstack, `iadd` nimmt die obersten zwei Werte vom Operandenstack, addiert diese und legt das Ergebnis wieder auf dem Operandenstack ab. `istore_2` nimmt den obersten Wert vom Operandenstack und speichert diesen in der dritten lokalen Variablen. Die Befehlsfolge ist daher eine einfache Addition der Form `i_2=i_1+1`.

Die Zuordnung der Operanden zu den Befehlsobjekten erfolgt nun, indem die Befehlsobjekte schrittweise abgearbeitet werden. Für Bytecodebefehle, die einen Wert auf dem Operandenstack ablegen, werden anstelle der Werte die Befehlsobjekte auf den virtuellen Operandenstack abgelegt. Dies wird für `iconst_1` und `iload_1` in der Abbildung 5.8 in den Bearbeitungsschritten **(a)** und **(b)** veranschaulicht.

Die Addition in Schritt **(c)** holt ihre Operanden vom Operandenstack, beim virtuellen Operandenstack werden nun die Befehlsobjekte vom Stack genommen und der Addition zugeordnet Anschließend wird das Befehlsobjekt für die Addition anstelle des Ergebnisses auf dem Operandenstack abgelegt.

Bei Befehlsobjekten, die nur vom Operandenstack lesen, wie es bei der Zuweisung in Schritt **(d)** der Fall ist, entsteht eine abgeschlossene Befehlsfolge. Diese wird dem Basisblock zugeordnet.

Bei dem beschriebenen Vorgang wird zum Übersetzungszeitpunkt die Ausführung des Java-Bytecodes teilweise simuliert. Im Unterschied zum realen Ablauf werden

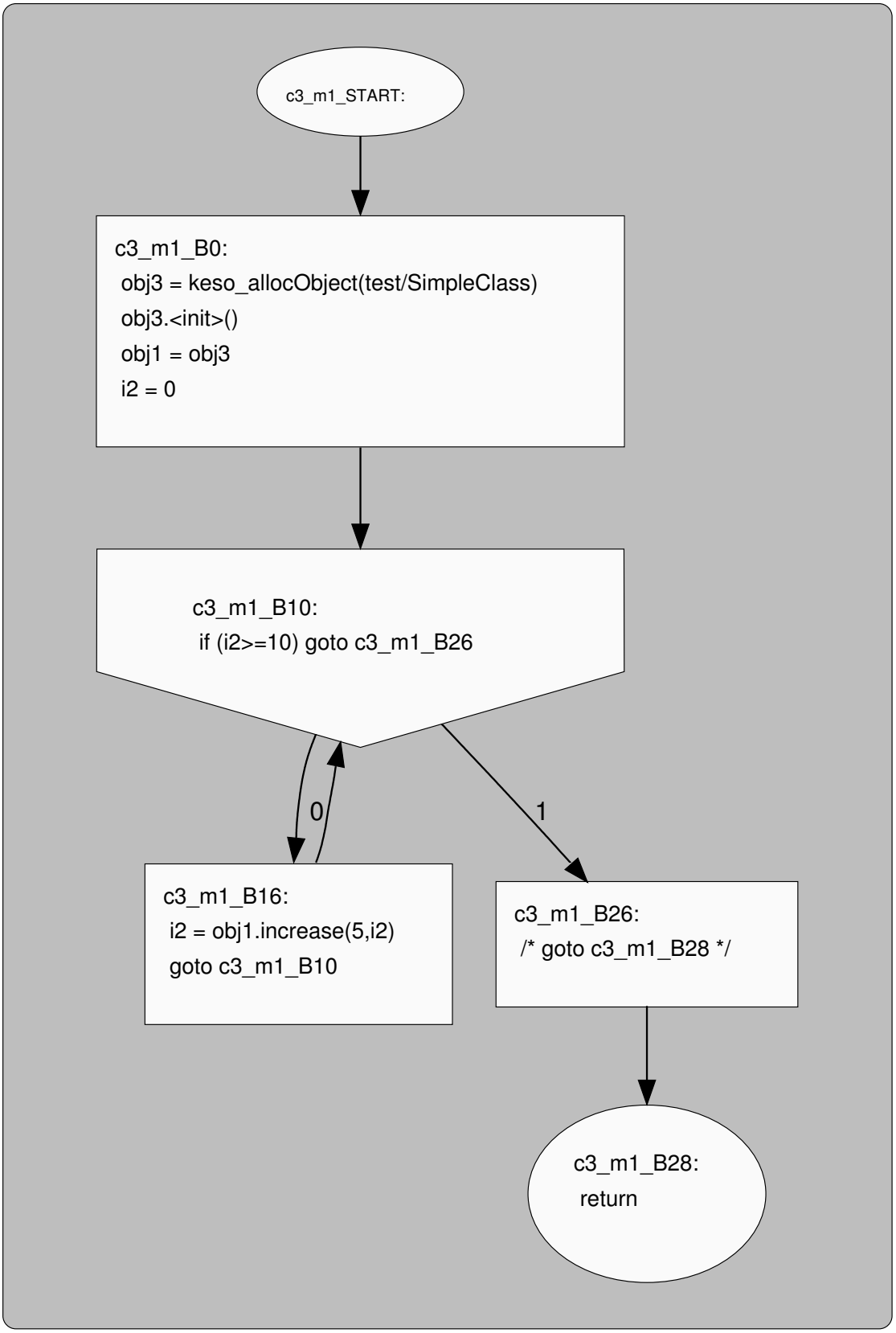

Abbildung 5.6: SimpleBCExample in der Zwischendarstellung von JINO

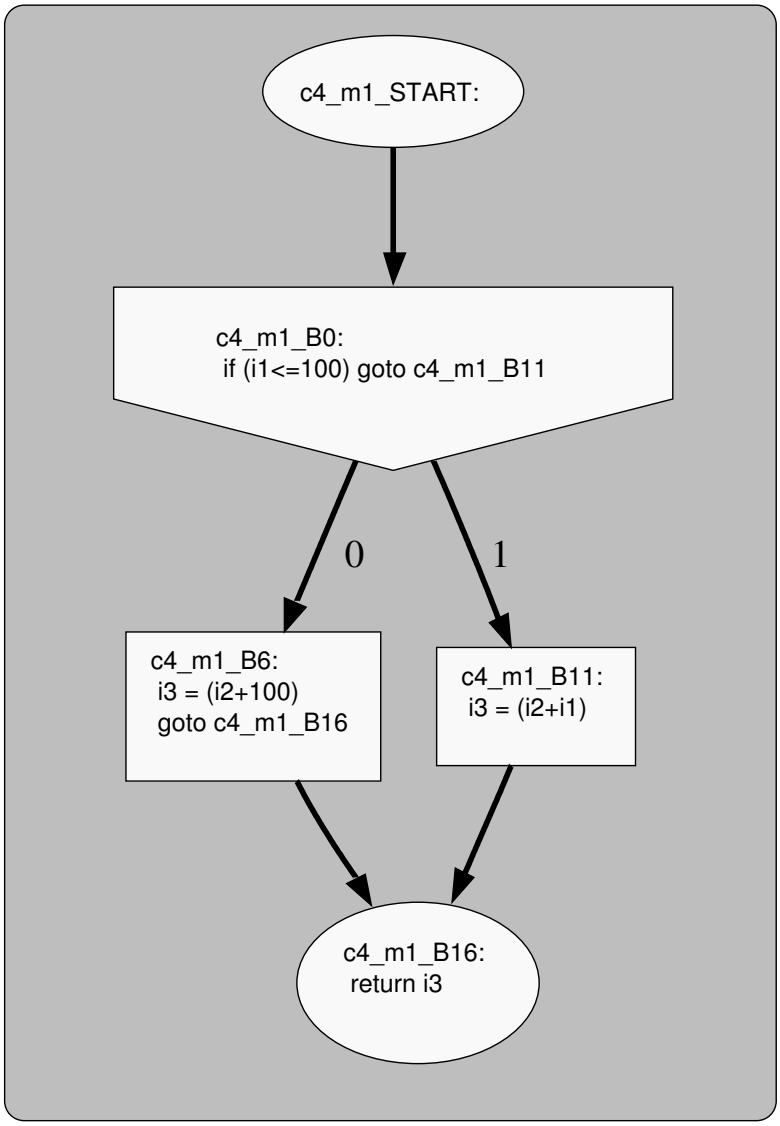

Abbildung 5.7: SimpleClass in der Zwischendarstellung von JINO

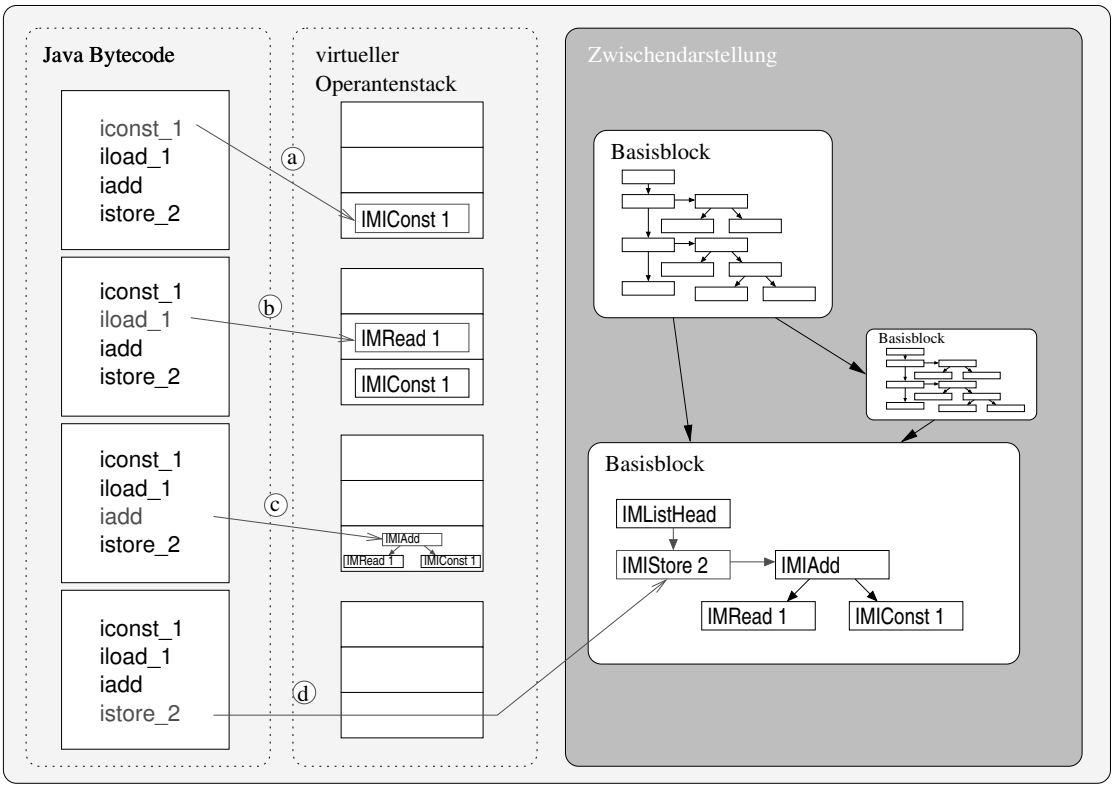

Abbildung 5.8: Funktionsweise des virtuellen Operandenstacks

die Befehlsobjekte als Stellvertreter für die realen Zustände auf dem Operandenstack abgelegt und die Anweisungen bekommen die Befehlsobjekte direkt als Operanden zugeordnet. Hierdurch entfällt der Operandenstack zur Laufzeit. Auch werden Bytecodebefehle, die reine Stackoperationen darstellen, wie die Bytecodebefehle zum Vertauschen oder Duplizieren von Operanden, bereits zum Übersetzungszeitpunkt ausgeführt.

Das Vorgehen ist jedoch nur zulässig, solange der Bytecodebefehl keine globalen Zustände verändert und der Operandenstack immer gleich und unabhängig vom Kontrollfluss verändert wird. So ist es zum Beispiel unkritisch, lokale Variablen und konstante Werte auf dem Operandenstack zu duplizieren. Würde man jedoch das Befehlsobjekt für einen Methodenaufruf duplizieren, so würde dieses die Semantik der Anwendung verändern. Diese Problematik wird gelöst, indem der Inhalt des virtuellen Operandenstacks am Ende eines Basisblocks neu erzeugten Variablen zugewiesen wird und diese auf dem Stack abgelegt werden. JINO achtet darauf, dass für alle Kontrollflüsse, die zu einem Basisblock führen, die gleichen Variablen auf dem Operandenstack liegen.

Die Größe des Operandenstacks und die auf dem Stack liegenden Datentypen müssen dazu beim Betreten des Basisblocks für jeden möglichen Kontrollflusspfad identisch sein. Ein laut Spezifikation [66] wohl geformter Bytecode besitzt genau diese Eigenschaft für fast alle Basisblöcke. Die Eigenschaft wurde von Gosling [44] bereits 1995 beschrieben. In der Literatur wird sie daher auch als Gosling-Eigenschaft bezeichnet.

Der Java Bytecode besitzt nun genau einen Befehl, der laut Spezifikation die Gosling-Eigenschaft verletzt. Es handelt sich dabei um den Bytecode *jump subroutine* (jsr), der zum Sprung in ein Unterprogramm dient, welches sich innerhalb einer Methode befindet. Laut Spezifikation müssen in diesem Fall nur die Einträge auf dem Operandenstack den gleichen Typ aufweisen, die auch im Unterprogramm verwendet werden.

Diese Ausnahme stellt nicht nur für die hier vorgestellte Transformation ein Problem dar, sondern auch für die genaue Typbestimmung, wie sie bei der Verifikation des Bytecodes und bei einer präzisen automatischen Speicherbereinigung benötigt wird [5]. Eine Lösung des Problems besteht darin, die Unterprogramme für jeden Aufrufpunkt zu vervielfältigen, so dass es für den Einstiegspunkt einer Unterroutine immer nur genau einen Vorgänger gibt. Da die Spezifikation rekursive Aufrufe von Unterprogrammen verbietet und in der Praxis die Unterprogramme häufig klein ausfallen, kommt es durch diese Umformung nur zu einem moderaten Zuwachs bei der Programmgröße [13]. Daher wurde auch für JINO dieser Ansatz gewählt und der Übersetzer vervielfältigt die Unterroutinen. Wird eine Unterroutine nicht spezifikationskonform verwendet, so bricht JINO den Übersetzungsvorgang ab.

Aktuelle Java-Übersetzer von Sun und Übersetzer für eine CLDC-konforme JVM erzeugen keine Unterroutinen mehr. Dies ermöglicht eine deutlich einfachere Umsetzung der JVM auf ressourcenbeschränkten Systemen.

5.4.2 Globale Optimierungen

Die Erreichbarkeitsanalyse und die Reduktion der Klassenhierarchie beim Laden der Klassen sind bereits Beispiele für globale Optimierungen. Für weitere globale Optimierungen werden zusätzliche Informationen benötigt, welche durch die Analyse der Zwischendarstellung gewonnen werden. Ändert sich die Zwischendarstellung, so veraltet die Information, und die Analyse muss wiederholt werden. Feldzugriffe und Methodenaufrufe können z.B. durch die Entfernung von nicht erreichbaren Programmteilen unter Umständen entfallen und so neue globale Optimierungen ermöglichen.

Die folgenden weiteren globalen Optimierungen werden von JINO durchgeführt:

In-Line-Expansion von Methoden

Das objektorientierte Programmiermodell von Java führt zu einer Vielzahl von kleinen Methoden. Insbesondere treten dabei oft Methoden auf, welche nur den Inhalt eines Feldes auslesen bzw. setzen oder lediglich einen weiteren Methodenaufruf tätigen.

Bei diesen Methoden ist der Zeitaufwand für den Methodenaufruf deutlich größer als die anschließend für die eigentliche Arbeit aufgewendete Zeit. Eine für diese Art von Methodenaufrufen vielversprechende Optimierungstechnik ist die sogenannte In-Line-Expansion.

Bei der In-Line-Expansion wird der Methodenaufruf durch den Inhalt der aufzurufenden Methode ersetzt. Werden häufig wiederverwendete große Programmabschnitte auf diese Weise in Methoden eingefügt, führt dies zu einem größeren Speicherbedarf für das Programm. JINO versucht deshalb nur bei Methoden, die den Speicherbedarf nicht unnötig steigern, die In-Line-Expansion anzuwenden.

Die In-Line-Expansion von virtuellen Methodenaufrufen ist nicht ohne Einschränkung möglich. Für die In-Line-Expansion muss zur Übersetzungszeit die aufzurufende Methode auch eindeutig feststehen. JINO nutzt hierfür die bei der Generierung der Methodentabellen gewonnenen Informationen und wendet die In-Line-Expansion an, wenn es nur einen Kandidaten für den Aufruf gibt.

Es stellt sich natürlich die Frage, ob die In-Line-Expansion in JINO sinnvoll ist, da moderne C-Übersetzer diese Technik auch beherrschen. Eine einfache Kennzeichnung von Funktionen, die für die In-Line-Expansion günstig erscheinen, könnte daher schon ausreichen. Es hat jedoch wesentliche Vorteile, die In-Line-Expansion in JINO selbst zu übernehmen und diese schon vor der der C-Programmerzeugung durchzuführen.

Die Entscheidung, ob und wo diese Optimierung zum Einsatz kommt, kann so unabhängig vom C-Übersetzer erfolgen, und zusätzlich ermöglicht die In-Line-Expansion spätere

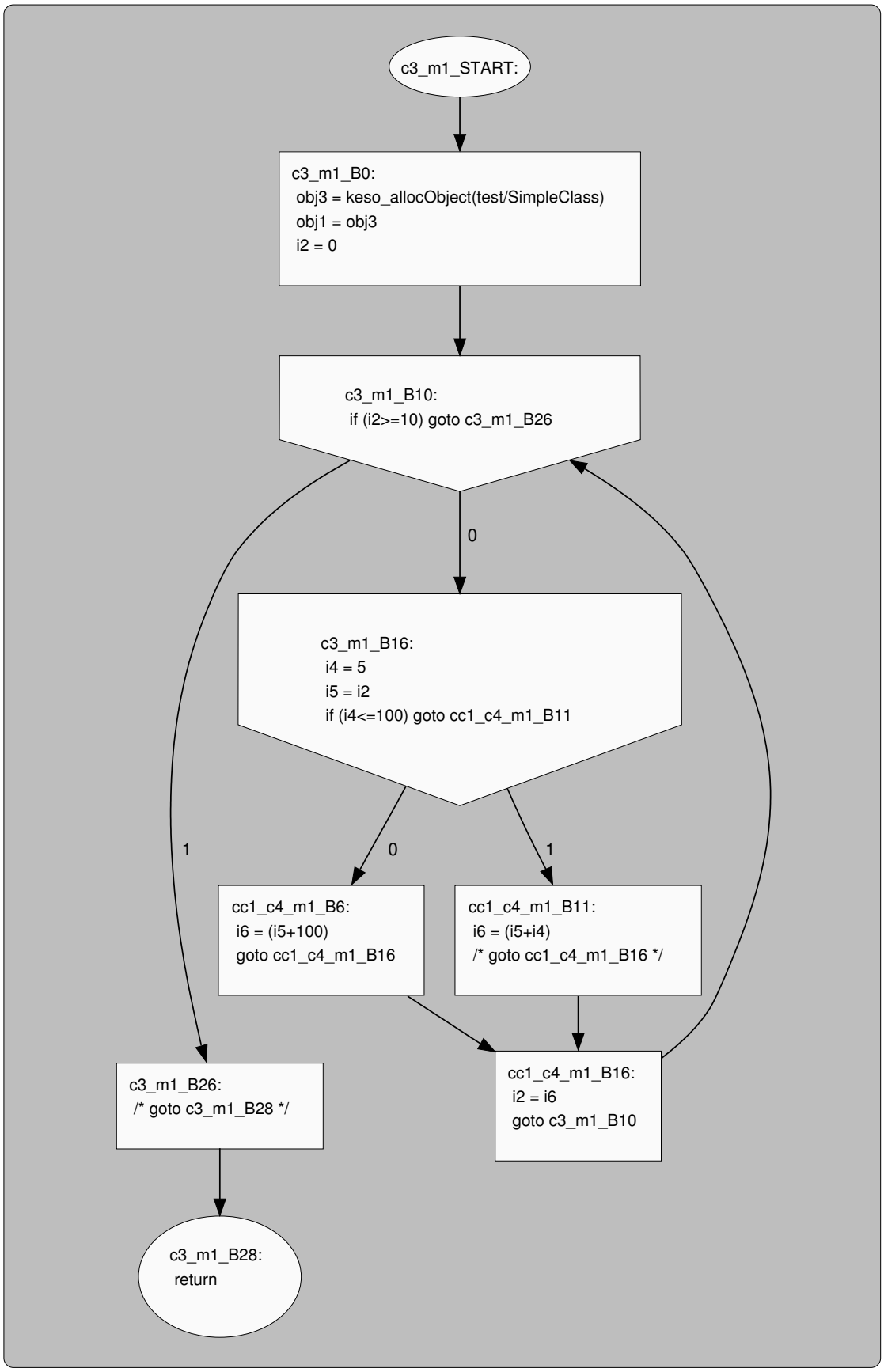

Abbildung 5.9: Beispiel: In-Line-Expansion

Optimierungsschritte, wie zum Beispiel das Eliminieren von Laufzeitüberprüfungen, wie es in Kapitel 5.4.3 beschrieben wird.

Abbildung 5.9 stellt den Kontrollflussgraphen der Methode `void launch()` nach der In-Inline Expansion dar. Durch die In-Line-Expansion entfällt der Aufruf der Initialisierungsmethode. Da die In-Line-Expansion aller Initialisierungsmethoden der Oberklassen zu leeren Methoden geführt hat, wurde auch hier nur noch eine leere Methode eingefügt.

Der Aufruf von `int increase(int,int)` wurde durch einen angepassten Kontrollflussgraphen ersetzt. Dabei werden neue lokale Variablen eingeführt, die die lokalen Variablen aus der eingefügten Methode ersetzen. Die Zuweisung der Argumente erfolgt im Anfang des ersten eingefügten Basisblocks, im Beispiel waren dies die Argumente `5` und `i2`, die nun den neuen lokalen Variablen `i4` und `i5` zugewiesen werden.

Durch die In-Line-Expansion ist eine Vielzahl neuer Möglichkeiten entstanden, Programmteile einzusparen. So wird zum Beispiel die lokale Variable `obj1` nicht mehr verwendet, und die lokale Variable `i4` ist immer kleiner/gleich `100` und als Folge ist immer die Bedingung im Basisblock `c3_m1_B16` wahr.

Auch hier könnte man die Optimierung dem nachfolgenden C-Übersetzer überlassen, der diese Möglichkeiten auch erkennen würde. Jedoch gibt es Fälle, die ein C-Übersetzer nicht erkennt. Daher beherrscht JINO einige lokale Optimierungen.

Weiterreichen von globalen Konstanten

Erkennt JINO ein globales Klassenfeld, welches nur bei der Klasseninitialisierung mit einem neu angelegten Objekt initialisiert wird und anschließend das Feld nur in der Ausführungsphase noch gelesen wird, so wird das Objekt statisch alloziert. Das statische Anlegen dieser Objekte ist sinnvoll, da diese über das Klassenfeld für die gesamte Laufzeit erreichbar ist und daher auch für die gesamte Zeit existiert. Eine dynamische Speicherverwaltung ist in diesem Fall nicht nötig.

Der Übersetzer muss jedoch sicherstellen, dass das globale Klassenfeld erst nach dem initialen Schreiben gelesen wird. Dies wird dadurch gewährleistet, dass das Schreiben nur in der Initialisierungsphase der Klassen stattfindet und das Lesen nur in der nachfolgenden Ausführungsphase.

Entfernen von nicht benötigten Objektfeldern

Um die Objektgröße zu reduzieren und damit Speicherplatz einzusparen, analysiert JINO die Feldzugriffe auf Objektfelder. Wird ein Objektfeld nur geschrieben, so können der Schreibzugriff auf das Feld und das Feld selbst entfernt werden, da die Daten

offensichtlich nicht weiter benötigt werden. Etwas Ähnliches gilt auch für Felder, die nur gelesen werden. Da die Spezifikation der JVM vorschreibt, dass alle Felder vor dem ersten Schreibzugriff mit Nullen initialisiert werden, liefert ein Lesezugriff auf diese Felder immer den konstanten Wert Null. Es ist daher möglich, die Feldzugriffe durch konstante Werte vom entsprechenden Typ zu ersetzen und das Feld aus dem Objekt zu entfernen.

Das Entfernen von Schreib- und Lesezugriffen wirkt sich auch positiv auf lokale Optimierungen des Übersetzungsvorgangs aus, wie zum Beispiel das Entfernen von unerreichbaren Programmteilen und das Auswerten und Weiterreichen von konstanten Ausdrücken.

Weiterreichen und Entfernen von konstanten Methodenparametern

Methoden und Klassen aus Bibliotheken sind in der Regel darauf ausgelegt, dass sie möglichst vielseitig wiederverwendet werden können. So besitzt z.B. die String-Klasse eine Methode, die Ganzzahlen zu einer Zeichenkette bezüglich einer frei wählbaren Basis umwandelt. Gleichzeitig gibt es auch für die am häufigsten verwendeten Basen 10 und 16 spezialisierte Methoden. Diese nutzen die allgemeine Methode und rufen diese mit einem festen Parameter auf.

So wird ein bestehender Programmteil wiederverwendet und bei der Verwendung von unterschiedlichen Basen wird hierdurch auch der Programmumfang kleiner. Wird jedoch nur eine Basis verwendet, was häufig der Fall ist, so wäre eine spezialisierte Implementierung der Methode in Bezug auf den Programmumfang besser.

Die bereits vorgestellte In-Line-Expansion von Methoden kann dieses Dilemma für kleine Methoden bereits lösen, da im Zusammenspiel mit den in Kapitel 5.4.3 beschriebenen lokalen Optimierungen nach der Expansion eine spezialisierte Version der Methode entsteht. Bei umfangreichen und häufig aufgerufenen Methoden führt die Anwendung der In-Line-Expansion jedoch zum Aufblähen des Programms und ist daher nicht praktikabel.

JINO erkennt, wenn Methodenparameter bei allen Aufrufen den gleichen konstanten Wert besitzen. In diesem Fall verschiebt JINO den Parameter durch das Einfügen einer neuen lokalen Variablen in den Methodenrumpf. Durch die lokalen Optimierungen entsteht, wie bei der In-Line-Expansion, eine spezialisierte Methode. Der ursprüngliche Parameter wird anschließend beim Aufruf entfernt.

Diese Optimierung ist auch bei virtuellen Methodenaufrufen möglich, falls bei allen Aufrufen ein konstanter Parameter verwendet wird. Das Weiterreichen von konstanten Methodenparametern ist daher auch noch anwendbar, wenn die In-Line-Expansion nicht möglich ist.

5.4.3 Lokale Optimierungen

Um den Datenfluss innerhalb einer Methode einfacher analysieren zu können, ist es sinnvoll, das Programm in der Zwischendarstellung zuerst in eine Form umzuwandeln, in der jede lokale Variable nur genau eine Definition besitzt. Im Übersetzerbau wird diese Form als *static single assignment* (SSA) Form [11, 33] bezeichnet. Da die SSA-Form die Optimierungen vereinfacht, wird sie in vielen modernen Übersetzern eingesetzt [9, 38, 38, 72, 96].

Bei der Umwandlung in die SSA-Form wird für jede Zuweisung, im Allgemeinen auch als Definition bezeichnet, eine neue Variable eingeführt. JINO ergänzt dabei die bestehenden Variablennamen um einen Zähler; dieser wird bei jeder neuen Definition erhöht. Interessant sind nun Stellen im Datenfluss, an dem die Definition einer Variablen mehrere Möglichkeiten besitzt und vom Kontrollfluss der Anwendung abhängt. In diesem Fall muss an der Stelle, an dem die unterschiedlichen Kontrollflüsse zusammenlaufen, an diesen Stellen muss eine neue Definition eingefügt werden, die in Abhängigkeit vom Kontrollfluss die neue Variable definiert. Für die Auswahl wird ein eigener Operator eingeführt, der als phi-Funktion bezeichnet wird. Die phi-Funktion bekommt für jeden eingehenden Kontrollfluss einen Parameter, mit dem für den Kontrollfluss jeweils gültigen Wert. Die phi-Funktion wählt anschließend in Abhängigkeit vom Kontrollfluss den entsprechenden Parameter aus.

Nach der Analyse- und Optimierungphase werden die phi-Funktionen wieder entfernt. Appel [11] beschreibt in seinem Buch zum modernen Übersetzerbau, wie die effiziente Umwandlung in die SSA-Form [33] möglich ist. Für das Verfahren werden die Dominanzeigenschaften zwischen den Knoten im Kontrollflussgraphen benötigt. JINO bestimmt diese Eigenschaft mit dem von Lengauer und Tarjan entwickelten Verfahren [63].

Abbildung 5.10 zeigt wieder das Beispiel aus Abbildung 5.9 nach der Umformung in der SSA-Form. Da in der SSA-Form jede Variable genau eine Definition besitzt, gestalten sich die lokalen Optimierungen wesentlich einfacher. Nachfolgend werden kurz die lokalen Optimierungen beschrieben, die JINO auf der Zwischendarstellung durchführt.

Weiterreichen von lokalen Konstanten

Wird eine lokale Variable durch einen konstanten Wert definiert, so ist es möglich, im jeweiligen Ausdruck anstelle der Variable den konstanten Wert zu verwenden. Das Weiterreichen von konstanten Werten führt unter Umständen dazu, dass es möglich wird, Ausdrücke bereits zum Übersetzungszeitpunkt auszurechnen. Ein Beispiel für diese Optimierung ist in Abbildung 5.10 zu sehen, bei der Zuweisung `i4_1=5` im Basisblock `c3_m1_B16`. Für sich genommen ist das Weiterreichen von Konstanten noch keine wirkliche Optimierung, sie ermöglicht jedoch die Auswertung von konstanten Ausdrücken.

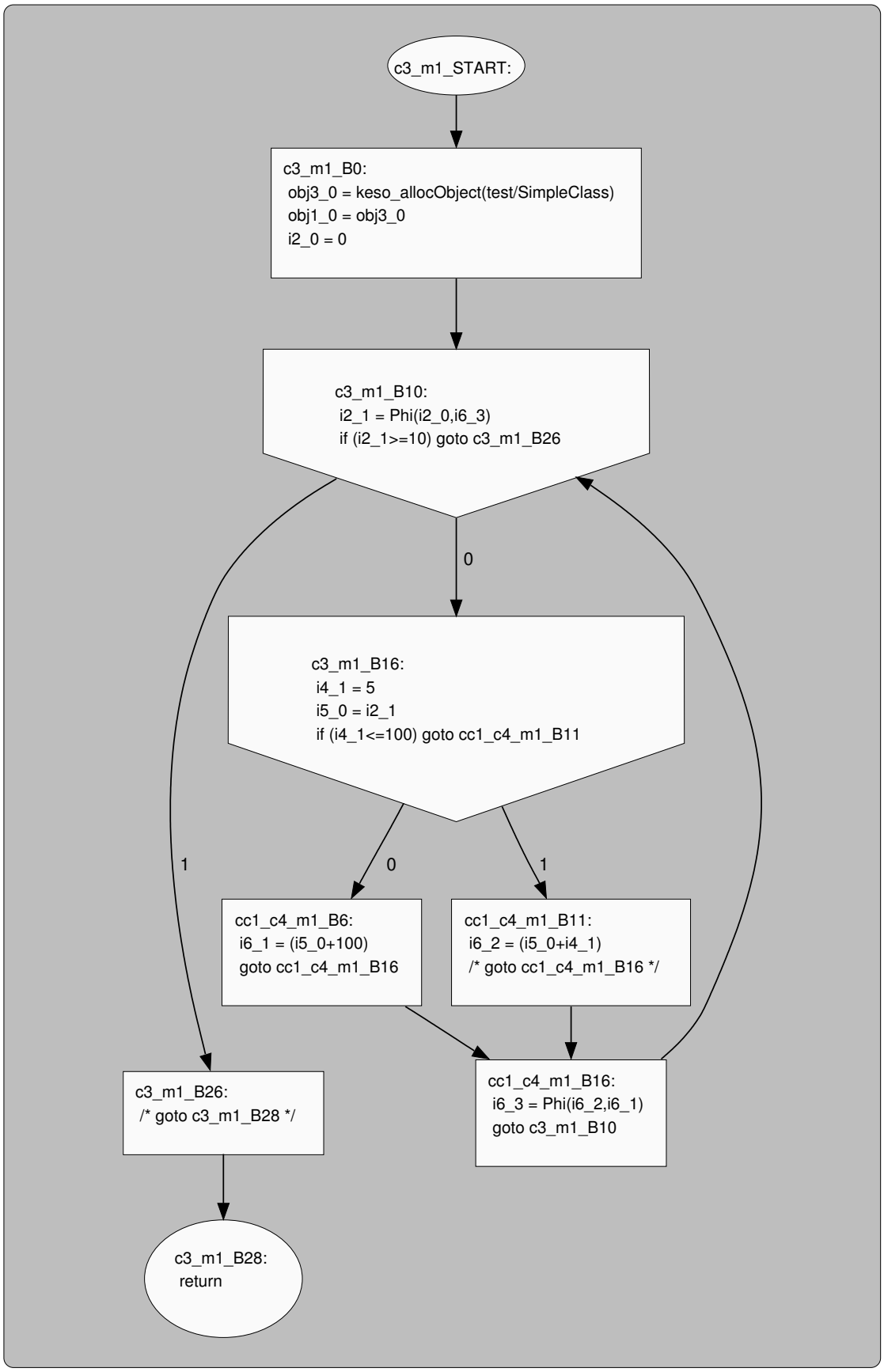

Abbildung 5.10: Beispiel: SSA-Form

Auswertung von konstanten Ausdrücken

Stehen alle Operanden eines Ausdrucks zum Übersetzungszeitpunkt fest, so kann er ausgewertet und durch sein Resultat ersetzt werden. Durch globale Optimierungen wie die In-Line-Expansion und durch das anschließende Weiterreichen von konstanten Werten entstehen vermehrt Ausdrücke, die bereits zum Übersetzungszeitpunkt ausgewertet werden können. Ein Beispiel dafür ist der Vergleich `i4_1<=100`. Da `i4_1` den konstanten Wert `5` besitzt, ist der Ausdruck immer wahr und der Sprung wird immer genommen. Es ist daher möglich, den bedingten Sprung durch einen unbedingten Sprung zu ersetzen und die Verbindung zwischen den Basisblöcken `c3_m1_B16` und `cc1_c4_m1_B6` zu löschen.

Weiterreichen von Kopien

Wird eine lokale Variable durch eine andere lokale Variable definiert, ist diese Variable also lediglich eine Kopie einer anderen Variablen, so ist es möglich, die ursprüngliche Variable anstelle der Kopie zu verwenden. Die Aussage ist so einfach zu treffen, da in der SSA-Form jede Variable nur einmal definiert und anschließend nur gelesen wird. Ein Beispiel hierfür ist die Zuweisung `i5_0=i2_1` im Basisblock `c3_m1_B16`. Durch das Weiterreichen von Variablen wird die Anzahl der Variablen reduziert und der Speicherbedarf für den Stack nimmt ab.

Diese Optimierung würde ein optimierender C-Übersetzer ebenfalls durchführen. Im Zusammenhang mit der in Kapitel 3.5.3 beschriebenen Referenztabelle zum Auffinden von lokalen Referenzen kann der C-Übersetzer diese Optimierung jedoch nicht mehr für lokale Objektreferenzen durchführen, da er davon ausgehen muss, dass die Kopie noch von einem anderen Kontrollfluss verwendet wird. In dem hier genannten Fall ist der zweite Kontrollfluss die automatische Speicherbereinigung. Der C-Übersetzer würde daher den Eintrag in der Referenztabelle belassen, was zu einem größeren Speicherbedarf auf dem Methodenstack führen würde.

Entfernen von ungenutzten Variablen

Variablen, die zwar definiert, aber nicht gelesen werden, sind ungenutzt und können entfernt werden, da offensichtlich der Inhalt der Variable nicht benötigt wird. Ein Beispiel hierfür ist die Zuweisung `obj1_0=obj3_0`. Wenn diese entfällt, ist auch die Zuweisung `obj3_0=keso_allocObject()` unnötig. Das Entfernen von ungenutzten Variablen senkt wieder den Speicherbedarf für den Stack. Auch für diese Optimierung gilt, dass ein C-Übersetzer diese Variablen nur entfernt, wenn keine Referenztabelle verwendet wird. Daher ist es sinnvoll, die Optimierung bereits in der Zwischendarstellung durchzuführen und sie nicht dem C-Übersetzer zu überlassen.

Entfernen von nicht benötigten Programmteilen

Durch das Entfernen von Zuweisungen wird unter Umständen auch der Ausdruck überflüssig, der die Variable definiert. Dies gilt für alle Ausdrücke, die abgesehen vom Rückgabewert keine globalen Zustände verändern. Im Beispiel wird die Speicheranforderung nicht mehr benötigt, denn da die zurückgelieferte Objektreferenz nicht gespeichert wird ist, das Objekt nicht mehr erreichbar. JINO entfernt daher den Aufruf von `keso_allocObject()`. Ein C-Übersetzer würde den Aufruf nicht entfernen, da aus der Sicht des C-Übersetzers global sichtbare Zustände verändert werden. Aus der Sicht von JINO ist dies jedoch nicht der Fall.

Entfernen von unerreichbaren Programmteilen

Basisblöcke, die keine eingehenden Kanten mehr besitzen, sind unerreichbar. Diese Basisblöcke werden nie angesprungen und daher auch nie ausgeführt und können entfernt werden, was direkt zu einem geringeren Speicherbedarf im Programmspeicher führt. Auch hier gilt wie für die anderen Optimierungen, dass sie auch vom C-Übersetzer geleistet werden. Das Entfernen von unerreichbaren Programmteilen beeinflusst jedoch auch die globalen Optimierungen von JINO, da hierbei möglicherweise auch Methodenaufrufe und Feldzugriffe entfallen und so bei einer globalen Analyse noch weitere Einsparungen möglich werden.

Entfernen redundanter Laufzeitüberprüfungen

Der Java-Bytecode erfordert an einigen Stellen Überprüfungen zur Laufzeit. Dazu zählen die Überprüfung, ob eine Objektreferenz vor der Verwendung gültig ist, ob ein Arrayzugriff innerhalb der Arraygrenzen stattfindet und ob das Objekt bei einer Einschränkung des Typs zur Laufzeit auch diesem Typ entspricht. Aus dem Datenfluss kann ein Teil der Überprüfungen bereits zum Übersetzungszeitpunkt beantwortet werden, da z.B. die Objektreferenz bereits zu einem früheren Zeitpunkt überprüft wurde. JINO versucht möglichst viele dieser Fälle zu erkennen und entfernt unnötige Überprüfungen.

5.4.4 Ergebnis

Die Abbildung 5.11 zeigt nun die umgewandelte Methode `void launch()`, nachdem die lokalen Optimierungen von JINO angewandt und die phi-Funktionen wieder entfernt wurden.

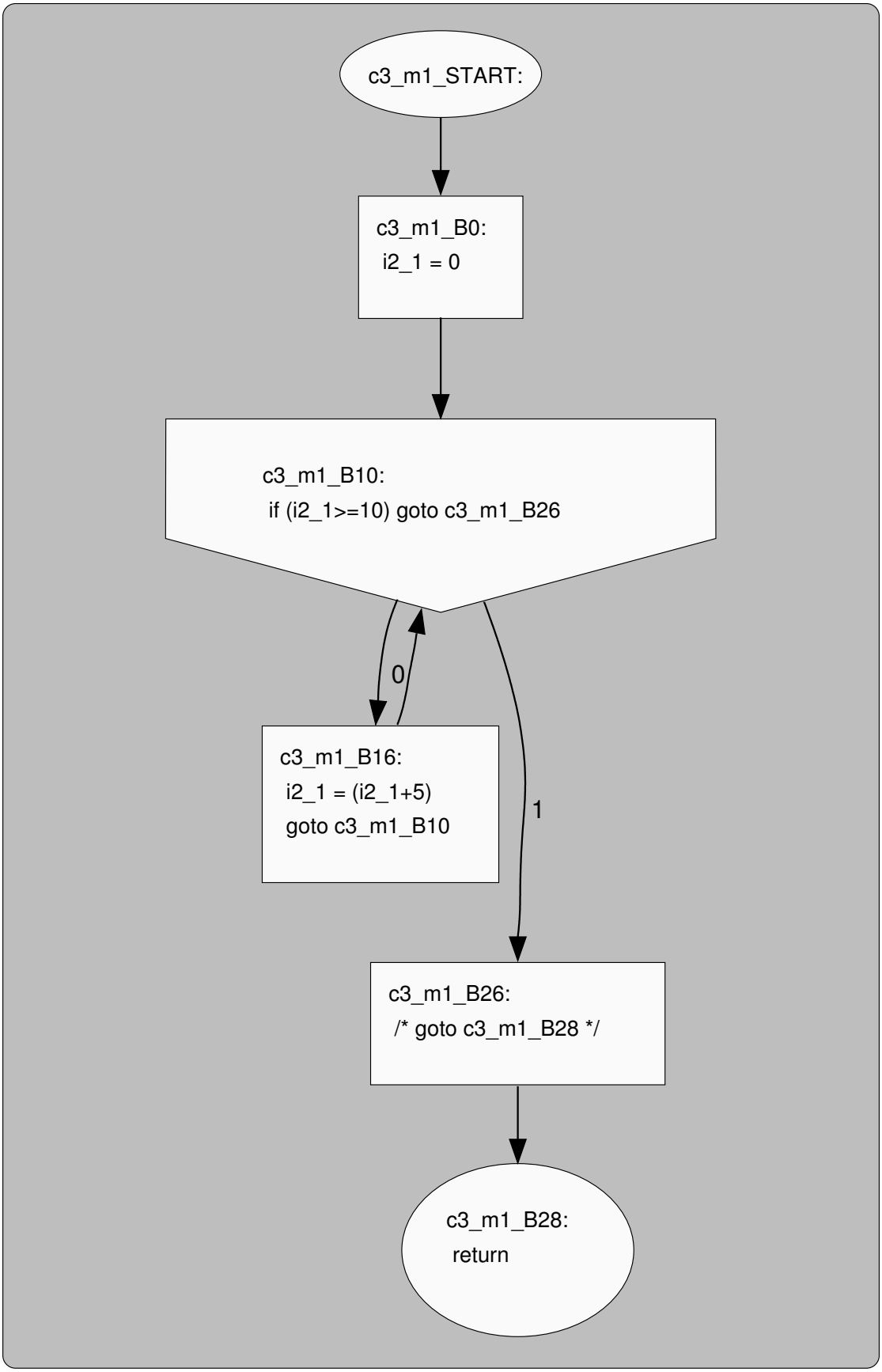

Abbildung 5.11: Beispiel: Von JINO erzeugtes C-Programm für die Methode `void launch()`

Wie schon erwähnt werden diese Optimierungen auch von heutigen C-Übersetzern vorgenommen. Es gibt jedoch gute Gründe die Optimierung bereits auf der Zwischendarstellung von JINO vorzunehmen. Zum einen besteht zu diesem Zeitpunkt noch ein größeres Wissen über die Bedeutung der Bytecode-Befehle und zum anderen beeinflussen die lokalen Optimierungen auch die globalen Optimierungen von JINO, die der C-Übersetzer mit seinem geringeren Wissen über die Bedeutung der Datenstrukturen nicht durchführen kann.

Der C-Übersetzer führt noch einige andere Optimierungen durch, die nicht in JINO umgesetzt wurden. Daher ist die Qualität des erzeugten Maschinenprogramms auch noch von der Qualität des nachfolgenden C-Übersetzers abhängig. Für den AVR und den TriCore wurden jeweils C-Übersetzer aus der *GNU Compiler Collection* (GCC) verwendet.

Diese Übersetzer verfügen über einige Erweiterungen, um das erzeugte C-Programm mit Zusatzinformationen zu versehen, die es dem Übersetzer ermöglichen, weitere Optimierungen vorzunehmen. JINO erzeugt mit den entsprechenden Aufrufparametern diese Zusatzinformationen automatisch. Die erzeugten Zusatzinformationen informieren den Übersetzer darüber, dass Sprünge zu einer Ausnahmebehandlung voraussichtlich nicht genommen werden, ob eine Funktion einen globalen Zustand verändert, und ob der Rückgabewert einer Funktion nur von deren Parametern abhängt. Besonders die letzteren Zusatzinformationen führen zu einem kompakteren Maschinenprogramm, da so der Übersetzer über Funktionsaufrufe hinweg globale Werte in Registern halten und Rückgabewerte mehrfach wiederverwenden kann.

Zu den weiteren Aufgaben, die dem C-Übersetzer überlassen werden, gehören die Abbildung der lokalen Variablen auf die Register und die Auswahl der Maschinenbefehle. Beides sind Operationen, die stark von der verwendeten Mikrocontrollerarchitektur abhängen.

Abbildung 5.12 zeigt nun abschließend das vom GCC erzeugte Maschinenprogramm für den TriCore TC1796. Das Maschinenprogramm ist ganze 20 Byte lang. Der ursprüngliche Programmumfang im Java-Bytecode betrug 52 Byte. Wie schon erwähnt sind diese Zahlen nicht repräsentativ, es wird jedoch in Kapitel 6 noch gezeigt, dass auch bei realen Anwendungen das erzeugte Maschinenprogramm kleiner wird als die ursprünglichen Klassendateien.

Die globalen Optimierungen ermöglichen oft neue lokale Optimierungen, und lokale Optimierungen wiederum können globale Optimierungen ermöglichen. Daher werden diese Optimierungen in mehreren Durchläufen so lange wiederholt, bis keine Änderungen mehr entstehen.

Es gibt noch eine Vielzahl von weiteren Optimierungen, die den Daten- und Programmumfang der Anwendung reduzieren würden. Eine für 8-Bit-Mikrocontroller besonders

```
tmp/keso_example/TriboardTC1796gnu/c3_m1.o:
file format elf32-tricore

Disassembly of section .text:

00000000 <c3_m1>:
   0:   82 00           mov %d0,0
   2:   da 09           mov %d15,9
   4:   02 21           mov %d1,%d2
   6:   8b 50 00 20     add %d2,%d0,5
   a:   3f 0f 04 00     jlt %d15,%d0,12 <c3_m1+0x12>
   e:   02 10           mov %d0,%d1
  10:   3c f9           j 2 <c3_m1+0x2>
  12:   00 90           ret
```

Abbildung 5.12: Das vom C-Übersetzer in der Abschlussphase für die Klasse `SimpleBCExample` erzeugte Maschinenprogramm

interessante Optimierung wäre die Reduktion der primitiven Datentypen aufgrund einer Datenflussanalyse. Der Java-Bytecode ist auf 32 Bit ausgelegt, 8- oder 16-Bit-Arithmetikbefehle gibt es nicht. Als Folge davon sind alle Arithmetikoperationen für 32-Bit-Werte ausgelegt und benötigen auf dem ATMega8535 vier Maschinenbefehle anstelle von nur eines Maschinenbefehls bei einer 8-Bit-Operation. Forschungsergebnisse [10] haben gezeigt, dass auch die Objektgröße sich um 40% verringern ließe, wenn die Datenfelder nur die wirklich benötigten Wortbreiten aufweisen würden.

5.5 Umsetzung der nativen Schnittstelle (KNI)

Der Java Bytecode kann nur den internen Zustand der virtuellen Maschine verändern. Für eine sinnvolle Anwendung ist es jedoch nötig, dass auch eine Kommunikation mit Anwendungen außerhalb der JVM und der Hardware stattfinden kann. Java sieht für diese Kommunikation den Aufruf von „nativen“ Methoden vor. Eine native Methode wird in Java durch das Schlüsselwort „native“ in einer Java Klasse definiert. Die eigentliche Implementierung der Methode erfolgt jedoch gesondert als Funktion in einer externen Programmbibliothek.

Die externen Programmbibliotheken werden in einer anderen Programmiersprache, wie zum Beispiel C, C++ oder Assembler geschrieben, und in Form von bereits übersetzten Maschinenprogrammbibliotheken bereitgestellt. Die nativen Bibliotheken profitieren nicht von der Portabilität des Java-Bytecodes. Die meisten werden bereits mit der JVM zusammen geliefert oder müssen von dieser dynamisch zur Laufzeit geladen werden.

Zur Interaktion zwischen den nativen Maschinenprogrammen und den Klassen und Methoden der virtuellen Maschine wurde eine Schnittstelle spezifiziert, das sogenannte *Java Native Interface* (JNI) [52]. Ein Entwicklungsziel dieser Schnittstelle war es, die nativen Bibliotheken möglichst unabhängig von der jeweiligen internen Implementierung der JVM zu machen. Dies ermöglicht, dass die Bibliotheken mit unterschiedlichen JVMs wiederverwendet werden können. Als Folge dieser Portabilität ist jedoch ein zusätzlicher Aufwand bei der Kommunikation zwischen virtueller Maschine und der nativen Bibliothek nötig, welcher sowohl den Speicherbedarf als auch die benötigte Rechenzeit erhöht.

Der zusätzliche Aufwand, der bei der Umsetzung des JNI nötig ist, macht diese Schnittstelle für den Einsatz in kleinsten eingebetteten Systemen mit stark eingeschränkten Ressourcen daher unattraktiv. In KESO gibt es zum Einbinden von nativen Programmteilen ein auf geringen Ressourcenbedarf hin optimiertes Konzept.

Abbildung 5.13 zeigt eine JNI-konforme Implementierung einer einfachen Methode für den direkten Speicherzugriff. Es handelt sich dabei um die Methode `get8(int)` aus der in Kapitel 3.4.3 beschriebenen Memory-Klasse. Die ersten drei Methodenaufrufe (`GetObjectClass`, `GetFieldID`) dienen dazu, von der JVM jeweils einen Identifier für die zwei Objektfelder der Memory-Klasse zu bekommen.

Anschließend wird die Größe des Speicherbereichs aus dem Objektfeld `addr` ausgelesen und es wird überprüft, ob der Zugriff innerhalb des Speicherbereichs erfolgt. Im Fehlerfall wird eine `java/lang/IndexOutOfBoundsException`-Exception geworfen. Im Erfolgsfall wird die eigentliche Speicheradresse ermittelt und der Speicher ausgelesen.

Nach der Rückkehr aus dem Methodenaufruf überprüft die JVM, ob ein Fehler aufgetreten ist und springt in diesem Fall in die Ausnahmebehandlung. Innerhalb der JNI-Implementierung muss diese Überprüfung vom Programmierer vorgenommen werden. Dies geschieht entweder durch Überprüfung der Rückgabewerte oder alternativ über den Aufruf der Methode `ExceptionCheck()`.

Es ist zu erkennen, dass neben der eigentlichen Funktionalität, die aus dem Auslesen des Speicherobjektes, der Bereichsüberprüfung und dem Speicherzugriff besteht, noch einiges an Mehraufwand geleistet werden muss und ein auf diese Weise implementierter Speicherzugriff nicht mit einem nativen Speicherzugriff vergleichbar ist.

5.5.1 Einweben im Methodenrumpf

Die von JINO angebotene Schnittstelle ermöglicht es, beliebige C-Anweisungen in das erzeugte C-Programm einzuweben. Hierzu werden, wie beim JNI, die Methoden einer Klasse nicht in Java implementiert, sondern die Klasse dient wieder nur als Gerüst und

```
#include <jni.h>
#include "libmem-jni.h"

/*
 * Class:     keso_core_Memory
 * Method:    get8
 * Signature: (I)B
 */
JNIEXPORT jbyte JNICALL Java_keso_core_Memory_get8(
                JNIEnv *env, jobject obj, jint offset)
{
        jclass   mem_class;
        jfieldID addr_id, size_id;
        jint     addr, size;

        mem_class = (*env)->GetObjectClass(env, obj);
        if (mem_class==NULL) return 0;

        addr_id = (*env)->GetFieldID(env, mem_class, "addr", "I");
        if (addr_id==NULL) return 0;

        size_id = (*env)->GetFieldID(env, mem_class, "size", "I");
        if (size_id==NULL) return 0;

        size = (*env)->GetIntField(env,obj,size_id);
        if ((*env)->ExceptionCheck(env)) return 0;

        if (offset<0 || offset>=size) {
                jclass ex_class = (*env)->FindClass(env,
                                "java/lang/IndexOutOfBoundsException");
                if (ex_class==NULL) return 0;

                (*env)->ThrowNew(env, ex_class, "out of memory range");
                return 0;
        }

        addr = (*env)->GetIntField(env,obj,addr_id);
        if ((*env)->ExceptionCheck(env)) return 0;

        return (jbyte)((unsigned char*)addr)[offset];
}
```

Abbildung 5.13: Implementierung der nativen Methode `Memory.get8(int)` mit JNI

```
package keso.compiler.kni.*;

import keso.compiler.*;
import keso.compiler.imcode.*;
import keso.compiler.backend.*;

public class MemoryGet8 extends Weavelet {

    /**
     * Im Konstruktor wird das Weavelet für die
     * gewünschte Methode registriert. Das Weavelet wird
     * daraufhin in den verschiedenen Phasen des Über-
     * setzungsvorganges zu der Methode konsultiert.
     */
    public MemoryGet8(BuilderOptions opts, ClassStore repository)
    {
        super(opts, repository);
        joinPoints = new String[] {
            "keso/core/Memory.get8(I)B"
        };
    }

    /**
     * Über die Methode "affectMethod" kann das Weavelet
     * die Implementierung der Methode ersetzen.
     */
    public boolean affectMethod(IMClass clazz, IMMethod method,
            Coder coder) throws CompileException
    {
        IMClass mem_class = repository.getClass("keso/core/Memory");

        IMNode obj = method.getArg(BCBasicDatatype.REFERENCE, 0);
        IMNode offset = method.getArg(BCBasicDatatype.INT, 1);

        coder.chk_range(obj, offset, 1, method, 0);

        coder.add("return␣((jbyte*)");
        coder.add_getField(mem_class, obj, "addr");
        coder.add(")[");
        offset.translate(coder);
        coder.addln("];");

        return true;
    }
}
```

Abbildung 5.14: Implementierung der nativen Methode `Memory.get8(int)` mit KNI, durch Einweben im Methodenrumpf

```
        /* ... */

        KESO_CHECK_NULLPOINTER(obj1, ERR267);
        b1 = c16_Memory_m3_get8(obj1, 1);
        b2 = c16_Memory_m3_get8(obj1, 5);

        /* ... */

jbyte c16_Memory_m3_get8(object_t* obj0,jint i1) {
        KESO_CHECK_NULLPOINTER(obj0, ERR342);
        KESO_CHK_BOUNDS(c16_Memory_t, obj0, i1, ERR343);
        return ((jbyte*)(C16_MEMORY_OBJ(obj0)->_addr))[i1];
}
```

Abbildung 5.15: Von JINO erzeugte C-Funktion beim Einweben im Methodenrumpf

Beschreibung der Schnittstelle für die eigentliche Implementierung. Die Methoden dienen dem Weber, der die C-Anweisungen direkt in die Anwendung einwebt, als Ansatzpunkte.

Der Weber wird dazu über Klassen, die zu JINO hinzugeladen werden, instrumentiert. Diese Klassen sind von der Klasse `keso.compiler.kni.Weavelet` abgeleitet. Die Abbildung 5.14 zeigt eine solche Klasse für die Methode `get8()`. Die Klasse wird im Konstruktor für die Methoden registriert, die von ihr beeinflusst werden sollen. Die Methode `affectMethod()` wird anschließend aufgerufen, bevor JINO das C-Programm für den Methodenrumpf erzeugt. Als Argumente werden die Zwischendarstellung der Klasse und der Methode des aktuellen Ansatzpunktes übergeben, so wie ein Coder-Objekt, welches zur Programmerzeugung dient.

Die Implementierung von `affectMethod()` ist nun vom Ablauf her mit der JNI-Implementierung vergleichbar. Von der Laufzeitumgebung werden zuerst die Klasse und die Parameter der Methode ermittelt, und anschließend wird mit dem `Coder` der benötigte C-Code erzeugt. Im Unterschied zu JNI geschieht dies jedoch bereits zum Übersetzungszeitpunkt und nicht erst zur Laufzeit.

Abbildung 5.15 zeigt den von JINO erzeugten C-Code. Die Methode `add_getField()` vom Coder-Objekt geht davon aus, dass der Zugriff über die Objektreferenz auf das `addr`-Feld noch eine Null-Zeiger-Überprüfung benötigt und diese einfügt.

Im Vergleich zur JNI Schnittstelle zeigt dieses Beispiel bereits eine beachtliche Einsparung. Ein nativer Speicherzugriff in einem C-Programm ist jedoch noch wesentlich schlanker. Durch die Funktionalität der KNI-Schnittstelle kann auch noch der verbleibende Mehraufwand verringert werden.

5.5.2 Einweben am Aufrufpunkt

Der Weber kann nicht nur die Programmerzeugung im Methodenrumpf einer Methode verändern, sondern auch die Programmerzeugung am Aufrufpunkt. Der eigentliche Aufruf der Methode kann somit entfallen. Bei kleinen Methoden, wie es bei unserem Beispiel der Fall, ist diese Variante die bessere Lösung. Abbildung 5.16 zeigt das entsprechende `Weavelet` zum Verändern des Methodenaufrufes. Wie beim Einweben im Methodenrumpf bestimmt der Weber wieder die Programmerzeugung.

Das Einweben am Aufrufpunkt hat den Vorteil, dass weniger oft eine Null-Zeiger-Überprüfung durchgeführt wird. Abbildung 5.17 zeigt das aus Abbildung 5.16 bekannte Beispiel nach dem Einweben am Aufrufpunkt.

Ein direkt in C programmierter Speicherzugriff würde jedoch im Vergleich immer noch mit einem geringeren Speicher und Zeit Aufwand auskommen, da bei JINO trotz konstanter Indizes immer noch die Überprüfung der Speichergrenzen zur Laufzeit stattfindet. Würde JINO zum Übersetzungszeitpunkt die im Speicherobjekt gespeicherte Adresse und Speichergröße kennen, so könnte diese Überprüfungen bereits zum Übersetzungszeitpunkt stattfinden und es würden weniger Maschinenbefehle benötigt.

5.5.3 Einwirken auf die Zwischendarstellung

Bei den in Kapitel 3.4.3 beschriebenen statischen Speicherobjekten stehen die benötigten Informationen für eine Bereichsüberprüfung bereits zum Übersetzungszeitpunkt zur Verfügung. Die Informationen müssen jedoch bis zum Aufrufpunkt von `get8()` weitergereicht werden. Die in Kapitel 5.4.2 beschriebenen Verfahren können die Objekte in der Zwischendarstellung weiterreichen, wenn bereits zu diesem Zeitpunkt ersichtlich ist, dass die Speicherobjekte konstant sind.

KNI bietet zu diesem Zweck die Möglichkeit, bereits auf die Zwischendarstellung einzuwirken. Der in Kapitel 3.4.3 beschriebene Speicherdienst wurde in KESO mit KNI umgesetzt. Werden über diesen Speicherdienst statische Speicherobjekte alloziert, so werden dafür in der Zwischendarstellung konstante Befehlsobjekte erzeugt. Die konstanten Objekte werden anschließend durch die lokalen und globalen Optimierungen bis zum Aufrufpunkt der Methode weitergereicht.

Beim Methodenaufruf an einem konstanten Objekt stehen so zum Übersetzungzeitpunkt die Speicheradresse und Spreichergröße bereit und können ohne Stellvertreterobjekt direkt verwendet werden. Die Überprüfung des Speicherbereiches kann bei einem konstanten Index nun auch zum Übersetzungszeitpunkt erfolgen.

Abbildung 5.18 zeigt das von JINO in diesem Fall erzeugte C-Programm. Der Aufwand eines Speicherzugriffes ist im Vergleich zu dem Aufwand eines handgeschriebenen C-Programms identisch. Im Gegensatz zum handgeschriebenen C-Programm wurden jedoch

```
package keso.compiler.kni;

import keso.compiler.*;
import keso.compiler.imcode.*;
import keso.compiler.backend.*;

public class MemoryGet8 extends Weavelet {

    public MemoryGet8(BuilderOptions opts, ClassStore repository)
    {
        super(opts, repository);
        joinPoints = new String[] {
            "keso/core/Memory.get8(I)B"
        };
    }

    /**
     * Über affectInvokeVirtual kann das Weavelet das C-Programm
     * am Aufrufpunkt der Methode beeinflussen.
     */
    public boolean affectInvokeVirtual(IMInvoke node, IMMethod caller,
            IMMethod callee, IMNode obj, IMNode args[], Coder coder)
            throws CompileException {

        coder.chk_range(obj, args[0], 1, caller,
                            node.getBCPosition());

        coder.add("((volatile␣jbyte*)␣");
        coder.add_getField(mem_class, obj, "addr");
        coder.add(")[");
        args[0].translate(coder);
        coder.add("]");

        return true;
    }
}
```

Abbildung 5.16: Implementierung der nativen Methode `Memory.get8(int)` mit KNI, durch Einweben am Methodenaufruf

```
/* ... */

KESO_CHECK_NULLPOINTER(obj1, ERR267);
KESO_CHK_BOUNDS(c16_Memory_t, obj1, 1, ERR268);
b1 = ((volatile jbyte*) (C16_MEMORY_OBJ(obj1)->_addr))[1];
KESO_CHK_BOUNDS(c16_Memory_t, obj1, 5, ERR269);
b2 = ((volatile jbyte*) (C16_MEMORY_OBJ(obj1)->_addr))[5];

/* ... */
```

Abbildung 5.17: Von JINO erzeugte C-Funktion beim Einweben am Aufrufpunkt

```
/* ... */

b1 = ((volatile jbyte*) (0x20))[1];
b2 = ((volatile jbyte*) (0x20))[5];

/* ... */
```

Abbildung 5.18: Von JINO über KNI erzeugte C-Funktion

die Speichergrenzen überprüft, und bei einem dynamischen Speicherobjekt würden ohne Anpassung der Quellen die erforderlichen Laufzeitüberprüfungen eingefügt.

5.6 Zusammenfassung

In diesem Kapitel wurde JINO, der für KESO entwickelte Java-Bytecode zu C Übersetzer vorgestellt. Die von JINO generierten Datenstrukturen wurden speziell für den Einsatz auf kleinsten eingebetteten Systemen entwickelt. Die Objekte besitzen einen bidirektionalen Aufbau; dieser ermöglicht ein schnelles Auffinden der im Objekt gespeicherten Objektreferenzen, und die Speicherverwaltung benötigt dafür nur die Anzahl der Referenzen. Der Objektkopf benötigt nur 1–4 Byte im Objekt und zählt damit zu den kleinsten Objektköpfen für Java-Objekte. Die im Objektkopf gespeicherte Klassenkennung kodiert direkt den primären Typ des Objektes und adressiert sowohl den Eintrag im Klassenspeicher als auch den gültigen Eintrag in der Methodentabelle.

Beim Übersetzungvorgang vom Bytecode zu einer OSEK/VDX-Anwendung wird die Klassenhierarchie bereits zu einem frühen Zeitpunkt ausschließlich auf die benötigten Klassen reduziert. Zur besseren Analyse wird der Bytecode in eine SSA-Form umgewandelt, und in dieser werden globale und lokale Optimierungen zur Reduktion des

Ressourcenbedarfs umgesetzt.

Für den Zugriff auf native Funktionen wurde KNI entwickelt, welches direkt in den Übersetzungsvorgang eingreift und so im Gegensatz zum JNI keinen zusätzlichen Mehraufwand zur Laufzeit erfordert.

Alle Maßnahmen zusammen führen dazu, dass das am Ende von JINO erzeugte C-Programm kleiner ausfällt als die ursprünglichen Java-Quellen. Dass dies nicht nur für kleine Beispiele gilt, sondern auch für reale Anwendungen, wird im nachfolgenden Kapitel gezeigt.

Kapitel 6

Evaluation

6.1 Einleitung

In den vorangegangenen Kapiteln wurde die Architektur von KESO, einer Multi-JVM für tiefe eingebettete Systeme, und die Funktionsweise von JINO, dem für KESO entwickelten Java-zu-C-Übersetzer, beschrieben. Im Folgenden sollen nun Vergleichsmessungen und prototypische Anwendungen zeigen, dass es möglich ist, diese Architektur auf Mikrocontrollern einzusetzen und dass das Problem eines erhöhten Ressourcenbedarfs dem Einsatz von softwarebasiertem Speicherschutz nicht entgegen stehen muss.

6.2 Interdomänenkommunikation

Für die Untergliederung von Software in isolierte Schutzräume ist die Zeit, die zur Kommunikation zwischen den Schutzräumen benötigt wird, ein entscheidender Faktor. Ist die benötigte Zeit zur Kommunikation zwischen den Schutzräumen zu groß, so wird versucht, die Kommunikation zwischen Schutzräumen zu minimieren und eine feingranulare Unterteilung der Software in Schutzräume wird vermieden.

Dies kann bei bestehenden Systemen beobachtet werden. Betriebssysteme wie Unix, Windows XP und Mac OS X[1] untergliedern die Systemsoftware nicht in weitere Schutzräume. Selbst Gerätetreiber von Fremdherstellern teilen sich den Schutzraum mit dem

[1] Mac OS X besitzt zwar einen Microkern, die Systemdienste werden jedoch nur in einem großen Schutzraum zusammengefasst.

	Pentium M @ 2 GHz Taktzyklen	TC1796 @ 50 MHz Taktzyklen
lokaler Portalaufruf ohne Parameter (mit Referenztabelle)	84–85	94
lokaler Portalaufruf ohne Parameter	20–21	25
lokaler Portalaufruf ohne Parameter (mit In-Line-Expansion)	12–13	17–19
leerer virtueller Methodenaufruf	9	11

Tabelle 6.1: Kosten für die Interdomänenkommunikation von KESO

Systemkern und gefährden die Systemstabilität. Zusätzlich puffern Bibliotheksfunktionen die Daten von der Anwendungsebene zwischen, um die Kommunikation mit dem Systemkern zu minimieren.

Bei kleinsten eingebetteten Systemen sind die einzelnen Softwarekomponenten bezüglich Größe und Funktionalität mit Gerätetreibern in Desktopsystemen vergleichbar und der Mehraufwand für die Zwischenpufferung von Daten ist keine akzeptable Lösung, da hierfür zusätzliche Systemressourcen benötigt werden. Eine schnelle Kommunikation zwischen Schutzräumen ist daher noch entscheidender dafür, ob Komponenten in Schutzräume untergliedert werden und ein Speicherschutzkonzept wirklich zum Einsatz kommt.

In der Tabelle 6.1 sind die in KESO für einen minimalen Portalaufruf benötigten Prozessortaktzyklen aufgelistet. Gemessen wurde sowohl auf dem Tricore TC1796 als auch auf einem Pentium M. Der Pentium wurde als weitere Prozessorarchitektur ausgewählt, um die benötigten Taktzyklen später besser mit anderen Systemen vergleichen zu können. Zu diesem Zweck lief ein KESO-System in einem Linuxprozess.

Verwendet wurden der Java-Übersetzer von SUN in der Version 1.6.0 und der C-Übersetzer aus der *GNU Compiler Collection* (GCC) in der Version 4.2.3 für den Pentium und in der Version 3.4.5 für den TC1796. Für die Messung wurde jeweils der Mittelwert aus 1000 Portalaufrufen bestimmt. Die Messungen wurden viermal wiederholt, die Tabelle zeigt jeweils den größten und kleinsten Wert aus allen Durchläufen. Die Zeiten wurden auf dem Pentium mit Hilfe des im Pentium vorhandenen Maschinenbefehls *read time stamp counter* (RDTSC) bestimmt und auf dem TC1796 mit Hilfe des PowerTracer 32 von Lauterbacher.

Die ersten drei Zeilen zeigen jeweils eine Variante eines Portalaufrufs ohne Parameter und ohne Rückgabewert. Die Variante, die von JINO für den Portalaufruf automatisch erzeugt wird, benötigt 20–21 Taktzyklen. In dieser Zeitspanne schaltet die Methode des Stellvertreterobjektes die globale Domänenkennung und das aktuelle Taskobjekt auf

System	Verfahren	Architektur	Taktzyklen
L4Ka Hazelnut RC1 [3]	Short IPC	Pentium III @ 450 MHz	818
JX [41]	Portalaufruf	Pentium III @ 500 MHz	750
J-Kernel [49]	LRMI (MS-JVM)	PPro @ 200 MHz	440
KESO	Portalaufruf	Pentium M @ 2 GHz	12–85
CIAO [67]	Serviceaufruf	TC1796 @ 50 MHz	174
KESO	Portalaufruf	TC1796 @ 50 MHz	17–94

Tabelle 6.2: Kosten für die Kommunikation zwischen Schutzräumen

Zieldomäne um und passt die Objektreferenz auf das Dienstobjekt an. Anschließend wird die Methode des Dienstobjektes aufgerufen.

Wird für die Objekte keine Referenztabelle auf dem Methodenstack benötigt, so findet auch keine Partitionierung des Methodenstacks statt. Der in Kapitel 3.5.3 beschriebene Vorgang kann daher entfallen. Die Kosten für einen Portalaufruf mit partitioniertem Methodenstack sind deutlich höher und liegt bei 84–85 Taktzyklen. Diese aufwändigere Variante wird von JINO jedoch nur erzeugt, wenn der Aufruf blockieren kann und die Speicherbereinigung eine Referenztabelle benötigt.

Die dritte Zeile zeigt das Ergebnis einer weiteren Optimierung. Bei dieser wird durch In-Line-Expansion die Dienstmethode direkt im Methodenstumpf des Stellvertreterobjektes ausgeführt, der zusätzliche Methodenaufruf wird so vermieden und der Aufruf dauert nur noch 12–13 Taktzyklen.

Zum Vergleich listet die Tabelle noch den Zeitaufwand für einen normalen leeren virtuellen Methodenaufruf, welcher 9 Taktzyklen dauert. Der Mehraufwand für einen Portalaufruf liegt daher auf dem Pentium bei 3–12 Taktzyklen im Normalfall und bei bis zu 76 Taktzyklen bei einem blockierenden Portalaufruf.

Um den Mehraufwand für die Kommunikation zwischen Schutzräumen bewerten zu können, zeigt die Tabelle 6.2 den Zeitaufwand für eine vergleichbare Kommunikation zwischen zwei Schutzräumen bei anderen Betriebssystemen. Auch hier wird jeweils ein Aufruf ohne Parameter und Rückgabewert durchgeführt. Aufgrund von Unterschieden in der Prozessorarchitektur sind die Zahlen nicht direkt vergleichbar, tendenziell schneiden Prozessoren mit einer höheren Taktrate, welche meistens auch eine längere Instruktionspipeline besitzen, bei dieser Messung etwas schlechter ab, da sie bei einer falschen Sprungvorhersage oder bei einem benötigten Hauptspeicherzugriff in der gleichen Zeitspanne mehr Taktzyklen mit dem Warten auf Daten verbringen als langsamer getaktete Prozessoren.

Fiasco/L4 [2] und L4Ka [3] sind zwei L4-Mikrokernvarianten. Die L4-Mikrokerne sind für ihren schnellen Adressraumwechsel bekannt. Für den Wechsel zwischen zwei

Schutzräumen wurde ein zweimaliger Adressraumwechsel gewertet. Den einfachen Adressraumwechsel schafft der L4Ka in nur 409 Taktzyklen.

JX ist wie bereits erwähnt ein Java Betriebssystem mit einer zu KESO vergleichbaren Architektur, jedoch mit dem wesentlichen Unterschied, dass Domänen und Programmfäden hier dynamisch zur Laufzeit erzeugt und beendet werden können. Bei einem Portalaufruf findet bei JX unter anderem ein Wechsel zu einem anderen Programmfaden statt, was zusammen mit den dynamischen Strukturen der Domäne zu dem Mehraufwand im Vergleich zu KESO führt.

J-Kernel ist eine rein Java-basierte Lösung für die Umsetzung von Schutzräumen. Die Schutzräume werden hier durch die Verwendung von unterschiedlich Klassenladern erzeugt. Jeder Klassenlader erzeugt einen eigenständigen Namensraum und erzeugt so für jeden Schutzraum getrennt Klassenfelder. Für die Kommunikation wurde ein lokaler Fernmethodenaufruf über den *Remote Method Invocation* (RMI) Mechanismus genutzt.

CIAO [51] ist ein in Aspect C++ entwickeltes Betriebssystem für kleinste eingebettete Systeme, welches eine zu Autosar OS vergleichbare Funktionalität bietet. Das System besitzt einen hardwarebasierten Speicherschutz und nutzt dafür die vom TC1796 angebotenen Begrenzungsregister. Das aspektorientierte Design dient unter anderem dazu, den Speicherschutz flexibel an die Anforderungen des Systemes anzupassen. Wie bei KESO sind auch bei CIAO die Schutzräume statisch konfiguriert.

Die Zahlen zeigen, dass die Kommunikation zwischen Schutzräumen bei KESO im Vergleich zu Systemen mit dynamischen Schutzräumen wesentlich schneller geschieht. Auch im Vergleich zu einem anderen statischen System, wie CIAO, welches auch keine Adressraumumschaltung benötigt, erzeugt KESO mit seinem konstruktiven Speicherschutzkonzept einen geringeren Mehraufwand bei einem Schutzraumwechsel. Der Mehraufwand für die Kommunikation zwischen Schutzräumen liegt bei KESO in den meisten Fällen nur bei 3–12 Taktzyklen im Vergleich zu einem normalen virtuellen Methodenaufruf und sollte daher kein Gegenargument für den Verzicht auf ein System mit feingranularen Schutzräumen darstellen.

6.3 Speicherbedarf für Methodentabellen

Die Verwendung von vielen virtuellen Methoden führt dazu, dass ein erheblicher Speicherbedarf für Methodentabellen entsteht. JINO wandelt daher, wenn möglich, virtuelle Methodenaufrufe in statische Methodenaufrufe um oder bietet die Möglichkeit, diese auf bedingte Sprünge abzubilden. Werden trotz dieser Maßnahmen noch Methodentabellen benötigt, so bietet JINO mit der in Kapitel 4.4 beschriebenen EHT und CEHT eine sehr effiziente und Speicherplatz sparende Variante von Methodentabellen an.

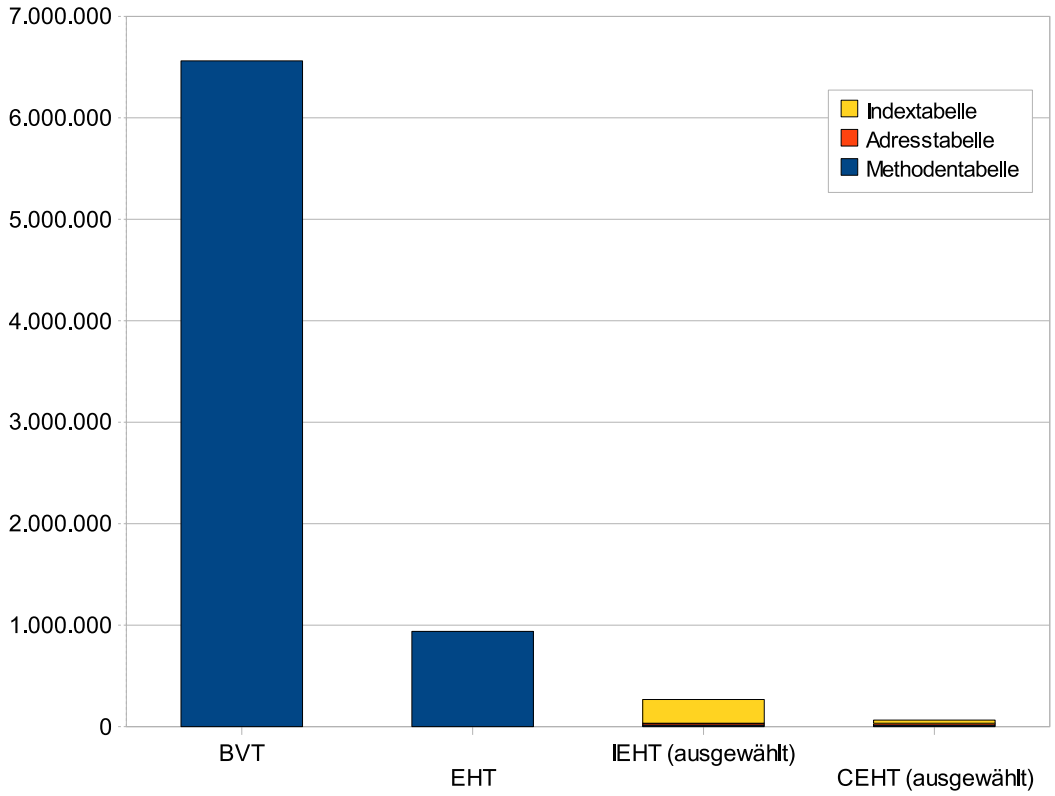

Abbildung 6.1: Speicherbedarf für die Methodentabellen (mit BVT als Vergleichswert)

Um den Speicherbedarf der EHT im Vergleich zu einer nicht optimierten BVT und das Potenzial der Kompression zu zeigen, wurden die Methodentabellen für eine große Klassenhierarchie mit vielen Schnittstellen bestimmt, wie sie in gängigen Java-Anwendungen vorkommen. Als Programmbasis dienen hierfür die Klassenbibliothek aus dem ClassPath-Projekt in der Version 0.12. Es handelt sich dabei um eine offene Implementierung der Java Standard Klassenbibliothek. Die Bibliothek enthielt 2715 Klassen; jede Klasse besitzt im Durchschnitt 36 Methoden. Dazu kommen noch 1792 Schnittstellen, die insgesamt 2233 Methoden definieren.

Alle Optimierungen, die JINO zur Reduktion der Klassen und Methoden bietet, wurden abgeschaltet, so dass für alle Methoden und alle Klassen eine Methodentabelle erzeugt wurde.

Die Abbildung 6.1 zeigt den Speicherbedarf für BVT, EHT, IEHT und CEHT im Vergleich, wobei bei der IEHT und der CEHT nur für ausgewählte Methodentypen eine weitere Indirektionsstufe erzeugt wurde. Für Methodentabellen, für die ein Index keinen

geringeren Speicherbedarf bedeuten würde, da sie zu wenig leere oder gleiche Einträge besitzen, wurde eine normale EHT erzeugt. Die Balken zeigen jeweils die Summe aus Indextabelle, Adresstabelle und Methodentabelle.

Die BVT benötigt mit 6.561 Kilobyte ungefähr siebenmal mehr Speicher als eine sortierte EHT, die für die Methodentabelle der vollständigen Klassenbibliothek noch immer 938 Kilobyte benötigen würde. Durch die Einführung der zusätzlichen Indirektionsstufe bei der IEHT kann der Speicherbedarf, aufgrund der unterschiedlichen Wortbreite für Indextabelle und Adresstabelle, bereits auf 267 Kilobyte reduziert werden. Komprimiert man nun die benötigte Indextabelle, indem man die Indexfolgen wie beschrieben wiederverwendet, so benötigt die als CEHT bezeichnete Tabelle mit 64 Kilobyte nur noch etwa 7% des Speichers im Vergleich zu einer EHT.

Abbildung 6.2 zeigt nochmals den Speicherbedarf für die EHT, IEHT und CEHT ohne die BVT. In dieser Abbildung ist Speicherbedarf für die Indextabelle (gelb), die Adresstabelle (rot) und die reine Methodentabelle (blau) zu erkennen. Der Speicherbedarf für die Adressen ist deutlich geringer als der für den Index, was daher rührt, dass die meisten Klassen in der Klassenbibliothek von ihrer Oberklasse die Methoden erben. Für Methodentabellen, die viele unterschiedliche Methoden enthalten, wurden keine Indextabelle und keine Adresstabelle erzeugt, sondern eine Methodentabelle ohne zusätzliche Indirektion.

Bei den zwei zusätzlich dargestellten Varianten von IEHT und CEHT wurden für alle Methoden — nicht nur für ausgewählte — die Index- und Adresstabelle erzeugt. Der Speicherbedarf für die Tabellen verändert sich dabei nur geringfügig. Die Methodentabelle hat jedoch den Vorteil, dass der Methodenaufruf eine Indirektionstufe weniger benötigt und damit der Speicherbedarf für Maschinenprogramm kleiner ausfällt und der Methodenauruf schneller ausgeführt wird.

Durch die Verwendung der CEHT kann der Speicherbedarf für die Methodentabelle deutlich reduziert werden. Erst bei größeren Mikrocontrollern und größeren Anwendungen entstehen große, lose besetzte Methodentabellen, die für eine Kompression der Methodentabellen interessant sind.

6.4 Prototypische Anwendungen

6.4.1 Hau den Lukas — C vs. Java

Die erste Anwendung, die mit KESO umgesetzt wurde, diente zum Vergleich der Echtzeitfähigkeit und des Ressourcenbedarfs von KESO mit einer in C entwickelten Anwendung. Bei der Anwendung handelt es sich um eine Steuerung für einen elektronischen Hau-den-Lukas.

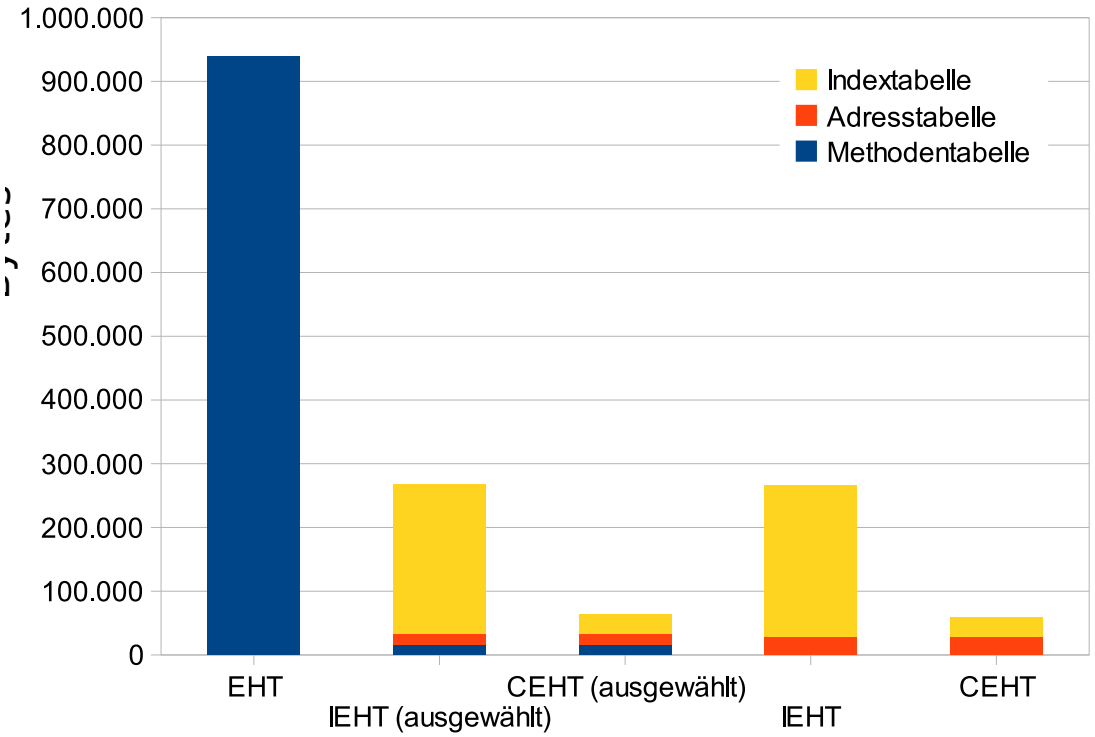

Abbildung 6.2: Speicherbedarf für die Methodentabellen (ohne BVT)

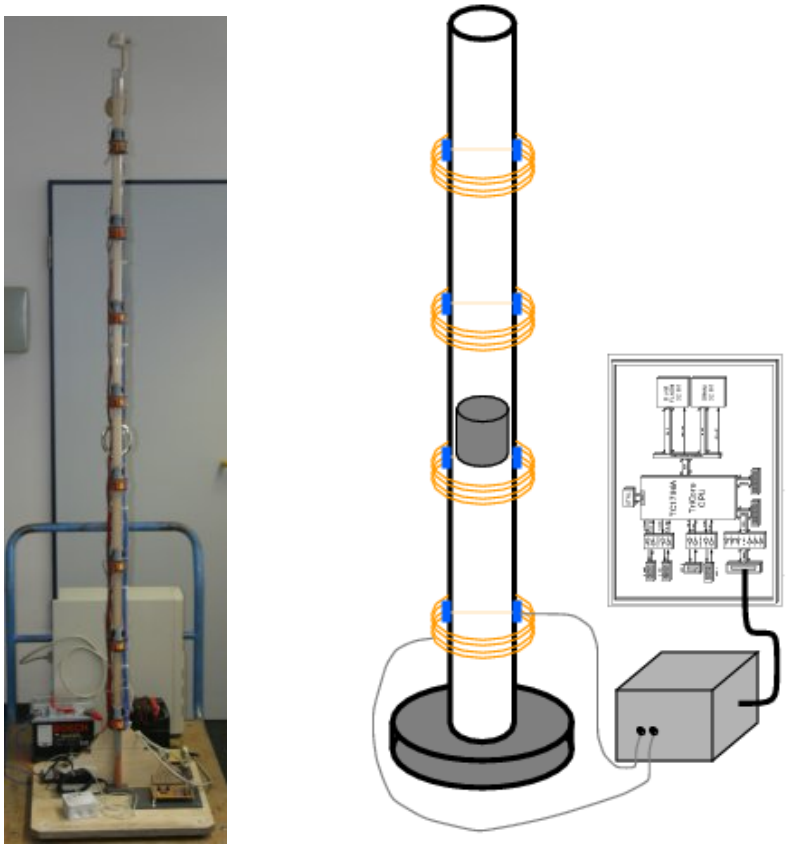

Abbildung 6.3: Versuchsaufbau für den elektrienschen Hau-den-Lukas

Die Abbildung 6.3 zeigt ein Foto und eine schematische Darstellung des Versuchsaufbaus. Ziel der Anwendung ist es, einen Eisenkern im Inneren einer Plexiglasröhre mit Hilfe von Elektrospulen und unter Einwirkung der Schwerkraft zu bewegen. Um die Position des Eisenkerns zu ermitteln, sind auf der Höhe der Spulen Lichtschranken angebracht, die dem Steuerrechner eine Rückmeldung geben. Gesteuert wird der Aufbau von einem TriCore TC1796.

Der TriCore TC1796 ist ein 32-Bit-Mikrocontroller mit einer Harvard-Architektur. Der Programmspeicher auf dem Chip besteht aus 2 MByte Flash, 48 KByte Scratchpad RAM, 16 KByte Cache und 16 KByte ROM. Für die Daten stehen 128 KByte Flash und insgesamt 80 KByte SRAM[2] zur Verfügung. Der Mikrocontroller kann mit bis zu 150 MHz betrieben werden, und die Instruktionen der RISC Architektur werden in einem Taktzyklus ausgeführt.

Der Mikrocontroller zählt zu den größten Mikrocontrollern, die im Automobil für Steuerungsaufgaben verbaut werden. Man findet diesen Mikrocontroller üblicherweise in der Motorsteuerung. Für den Controller gibt es von verschiedenen Herstellern ein OSEK/VDX-kompatibles Betriebssystem.

Der Testaufbau diente in der Vorlesung für Echtzeitsysteme[3] als Versuchsaufbau für eine Anwendung mit harten Echtzeitanforderungen. Die Echtzeitanforderungen ergeben sich aus dem Umstand, dass die Spulen immer im genau richtigen Zeitpunkt aktiviert und deaktiviert werden müssen. Nur so vollzieht der Eisenkern die gewünschte Positionsänderung in der Röhre. Darüber hinaus ist es wichtig, die Spulen nicht zu lange unter Strom zu halten, da der hohe Stromfluss die Spulen innerhalb von 20 Sekunden so stark erwärmt, dass die Plexiglasröhre anfangen würde zu schmelzen.

Von Studenten wurde basierend auf einem OSEK/VDX-System eine Steuerungssoftware in C entwickelt, die es ermöglicht, den Eisenkern in der Plexiglasröhre nach unterschiedlichen Ablaufplänen zu bewegen. Die Steuersoftware unterteilt den Ablauf in elementare Aktionen, wie das Anheben oder Absenken des Eisenkernes um eine Spule. Diese elementaren Bewegungen können anschließend zu komplexeren Kompositionen zusammengefügt werden. Ein Ablaufplan ist eine Komposition aus elementaren Aktionen. Der Ablaufplan wird als eine Folge von Bytes gespeichert. Erkennt die Steuerungssoftware, dass eine elementare Bewegung nicht erfolgreich verlaufen ist, so wird in einen Notbetrieb geschaltet. Der Notbetrieb stellt sicher, dass der Eisenkern beim Fallen abgebremst wird, so dass der Aufbau nicht beschädigt wird.

[2] Der Datenspeicher ist auf unterschiedliche Einheiten aufgeteilt, welche unterschiedliche Anforderungen erfüllen. 56 KByte dienen als lokaler Datenspeicher, welcher direkt an der Zentralenrecheneinheit angeordnet ist, 8 KByte besitzen zwei Datenbusse zur schnellen Kommunikation, 64 KByte normales SRAM und 8 KByte die auch im Energiesparmodus erhalten bleiben.

[3] Die Vorlesung wird seit dem Sommersemester 2006 an der Friedrich-Alexander Universität Erlangen-Nürnberg gehalten. URL: http://www4.informatik.uni-erlangen.de/Lehre/SS08/V_EZS2/

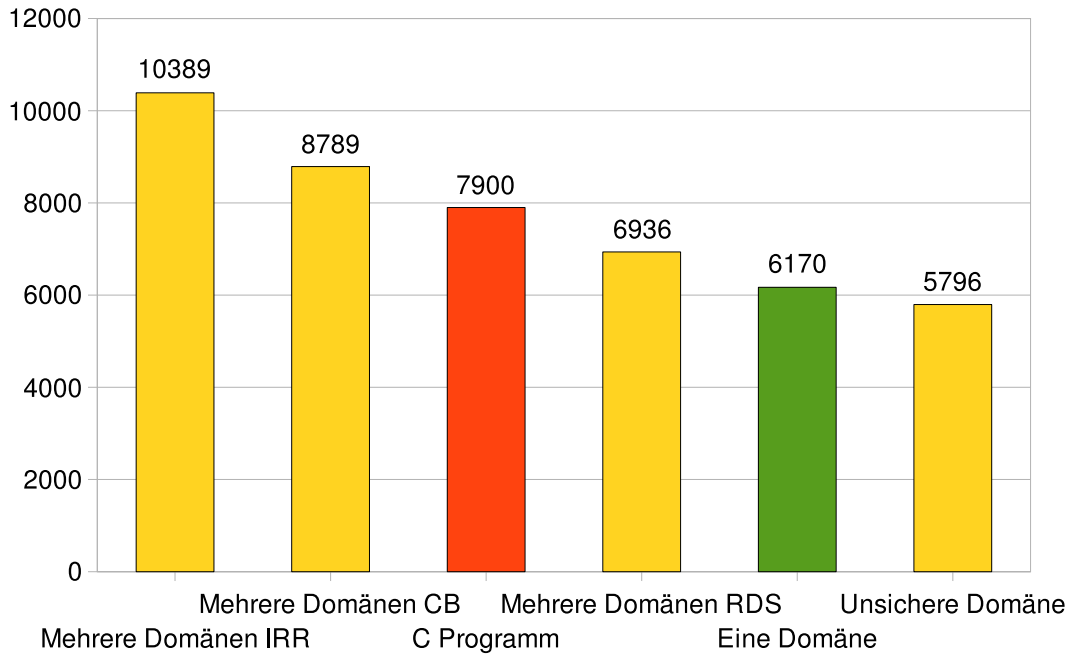

Abbildung 6.4: Programmumfang: C im Vergleich zu Java

Für KESO wurde die C-Anwendung nach Java portiert. Der Aufbau der Java-Anwendung folgt der C-Vorlage, es wurde kein Objekt-orientierter Neuentwurf der Anwendung vorgenommen. Die C-Funktionen wurden als statische Methoden umgesetzt und auf Klassen aufgeteilt. Die Ablaufpläne der ursprünglichen Anwendung können auch von der Java-Anwendung interpretiert werden und verhalten sich auch im Zeitverhalten identisch. Weder die in C noch die in Java implementierte Anwendung alloziert, nach der Initialisierungsphase, dynamisch Speicher. Für KESO wurde daher die minimale Speicherverwaltung aus Kapitel 3.5.2 verwendet.

Die beiden Anwendungen sind aufgrund der identischen Funktionalität und des identischen Aufbaus gut vergleichbar. Beide Anwendungen verhalten sich im Testlauf identisch. Die Echtzeitanforderungen werden von beiden Anwendungen erfüllt.

Aufgrund zusätzlicher Laufzeitüberprüfungen für die Arraygrenzen und die Typsicherheit ist damit zu rechnen, dass die Java-Anwendung umfangreicher ausfällt als die ursprüngliche, reine C-Anwendung. Auch eine automatische Speicherbereinigung und die Verwaltung von getrennten Schutzräumen sollte zu einem Mehrbedarf an Programmspeicher führen.

Abbildung 6.4 zeigt die Größe der C-Anwendung (rot) und die Größe von unterschiedlichen Konfigurationen für die Java-Anwendung. Der grüne Balken zeigt die Programm-

größe für eine Konfiguration mit nur einer Domäne und mit der minimalen Speicherverwaltung. Diese Konfiguration ist abgesehen vom zusätzlich bestehenden konstruktiven Speicherschutz mit der C-Anwendung vergleichbar und entspricht so dem gesetzten Ziel. Die weiteren Balken veranschaulichen unterschiedliche Konfiguration von KESO, angefangen von einer Konfiguration mit echtzeitfähiger Speicherverwaltung und mehreren Domänen bis hin zu einer Konfiguration ohne Laufzeitüberprüfungen.

Die Konfigurationen mit minimaler Speicherverwaltung sind kleiner als die ursprüngliche, in C geschriebene, Anwendung. Dies zeigt, dass die durch den sprachbasierten Speicherschutz entstandenen Mehrkosten durch die globalen Optimierungen von JINO mehr als ausgeglichen wurden.

Bei der kleinsten Konfiguration wurden die Laufzeitüberprüfungen deaktiviert. Ohne Bereichsüberprüfung bietet KESO zwar keinen zuverlässigen Speicherschutz mehr. Betrachtet man die Konfiguration jedoch im Hinblick auf die möglichen Programmierfehler und im Besonderen auf die Art von Programmierfehlern, die den Speicherzugriff betreffen und die auch mit dem MISRA C-Standard vermieden werden sollen, so bietet Java diese Fehlerquellen weiterhin nicht. Die Konfiguration bietet daher immer noch einen kleinen Mehrwert im Vergleich zur reinen C-Anwendung.

Eine einzige prototypische Anwendung ermöglicht noch keine allgemeine Aussage darüber, um wie viel der Ressourcenbedarf durch den Einsatz von KESO als Laufzeitumgebung steigt oder fällt. Die Messergebnisse zeigen jedoch, dass der Ressourcenbedarf der Java-Anwendung mit dem Bedarf einer C-Anwendung konkurrenzfähig ist und nicht zwingend aufgrund des Mehrwertes auch ein Mehrbedarf entsteht. Je nachdem, wie groß das Optimierungspotenzial ist und wie viel zusätzliche Funktionalität von KESO genutzt wird, fällt der Ressourcenbedarf höher oder geringer aus.

6.4.2 Robertino

Der beim ersten Prototypen verwendete Mikrocontroller TC1796 ist, wie bereits erwähnt, einer der größeren Mikrocontroller im Automobil, und auch wenn ein Trend zu größeren Mikrocontrollern erkennbar ist, werden die kleinen 8-Bit-Mikrocontroller in Zukunft weiterhin eine Vielzahl von Aufgaben übernehmen.

Für die zweite prototypische Anwendung wurde daher ein 8-Bit-Mikrocontroller als Zielplattform ausgewählt. Bei dem Demonstrator handelt es sich um einen Robertino [83]. Dieser Roboter wurde ursprünglich für die Fußball-Robotermeisterschaft entwickelt. Er besitzt drei über einen CAN-Bus mit einem Industrie-PC verbundene Antriebseinheiten. Jede Antriebseinheit ist mit einem 8-Bit-Mikrocontroller ausgestattet. Bei dem Mikrocontroller handelt es sich um einen ATMega8535 von AVR. Dieser übernimmt die Motorsteuerung, das Auslesen von jeweils zwei Abstandssensoren und die Kommunikation mit den anderen Knoten über einen CAN-Bus.

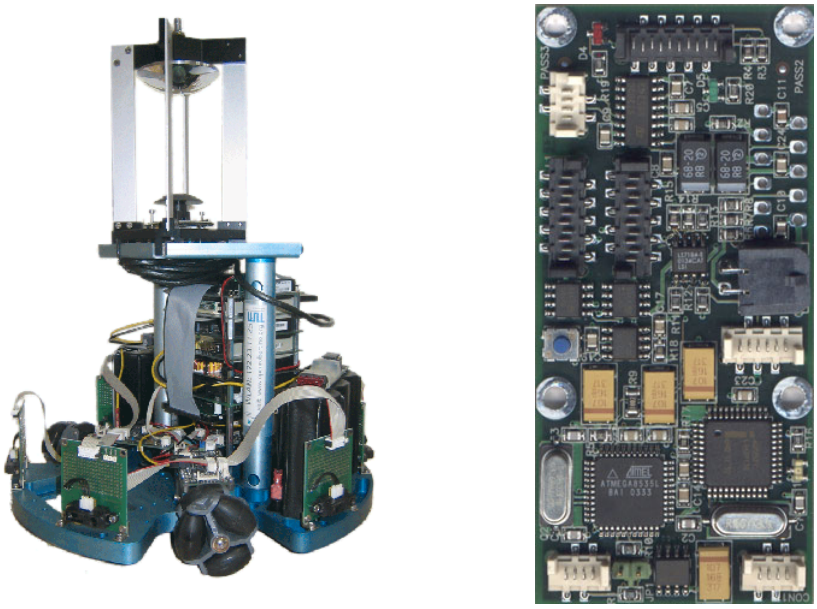

Abbildung 6.5: Robertino

Die Abbildung 6.5 zeigt den Roboter und eine der drei für Motorsteuerungen genutzte Platinen mit dem ATMega8535. Für die Demonstration werden nur diese drei Motorsteuerungen und Abstandssensoren genutzt, der Industrie-PC und die Bildverarbeitung bleiben außen vor.

Der ATmega8535 ist ein 8-Bit-Mikrocontroller mit 8 KByte Programmspeicher (Flash) und 512 Byte Arbeitsspeicher (SRAM). Die Abbildung 6.6 zeigt den schematischen Aufbau der in Java für die Motorsteuerung entwickelten Anwendung. Die einzelnen Komponenten der Motorsteuerung wurden ohne Ausnahme in Java programmiert. Es wurden daher Treiber für den SPI-Bus, den CAN-Controller, die Motorsteuerung, die Puls-Weiten-Modellierung (PWM) und die Analogdigitalwandler des Mikrocontrollers implementiert.

Aufbau der Steuerung

Darüber hinaus wurde noch eine Komponente entwickelt, welche den Roboter autonom steuert, so dass dieser sich durch einen Raum bewegt und Hindernissen ausweicht. Da die Abstandssensoren nur einen recht schmalen Bereich erfassen und so Ecken und schlanke Stuhlbeine nicht erkannt werden, vollführt der Roboter während seiner Fahrt eine alternierende Drehung, um die eigene Achse, ohne dabei die Fahrtrichtung zu ändern. Auf diese Weise wird von der Messlanze ein größerer Bereich vor dem

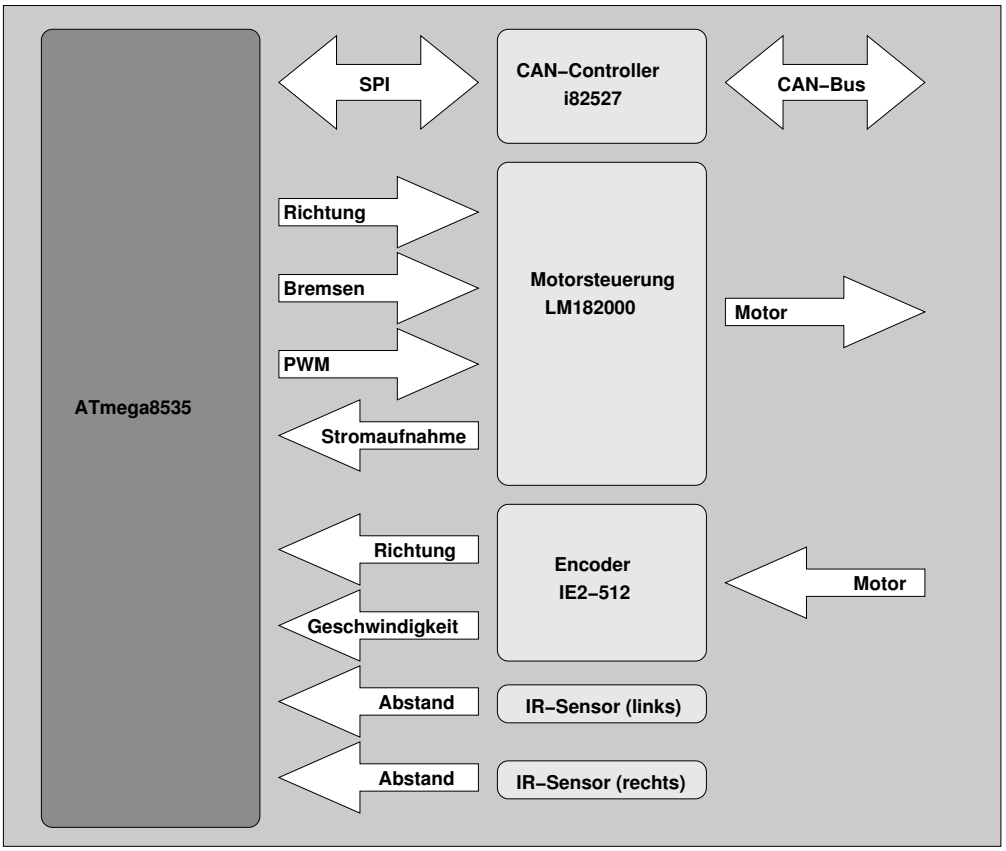

Abbildung 6.6: Schematischer Aufbau der Motorsteuerung

Roboter abgetastet. Stößt der Roboter trotz dieser Maßnahme auf ein unüberwindbares Hindernis, so blockieren die Räder und die Stromaufnahme der Motoren nimmt zu. Die Motorsteuerung erkennt diese Situation und meldet dies der Anwendung. Die Anwendung setzt in diesem Fall den Roboter ein kurzes Stück zurück und ändert anschließend seine Fahrtrichtung um 90 Grad.

Im Gegensatz zum elektronischen Hau-den-Lukas wurde diese Anwendung nicht portiert, sondern neu entwickelt. Alle Teile der Anwendung — Treiber, Kommunikationsschicht und Steuerlogik — sind in Java geschrieben. Bei der Anwendungsentwicklung wurde darauf geachtet, dass nach der Initialisierungsphase durch die Anwendung keine dynamische Speicheranforderung mehr stattfindet, es war daher möglich, auch hier die minimale Speicherverwaltung aus Kapitel 3.5.2 zu verwenden.

Verteilte Anwendung

Die drei Motorsteuerungen bilden zusammen eine verteilte Anwendung. Der Portalmechanimus, welcher zur Kommunikation zwischen Domänen dient, wurde dazu um Fernmethodenaufrufe erweitert, so dass eine Verteilung der Domänen auf die drei Mikrocontroller ermöglicht wurde.

Die von JINO automatisch erzeugten Stellvertreterobjekte leiten bei einer verteilten KESO-Anwendung die Methodenaufrufe über ein frei konfigurierbares Netzwerk an einen entfernten Mikrocontroller weiter. Auf dem entfernten Knoten werden die weitergeleiteten Aufrufe anschließend an die Dienstobjekte weitergeleitet. Auch hierfür wird von JINO der benötigte Programmcode automatisch erzeugt, so dass aus Sicht der Anwendung ein Portalaufruf an einer lokalen Domäne sich nicht von einem Portalaufruf an einer entfernten Domäne unterscheidet. Ein von einer Domäne angebotener Dienst kann daher ortstransparent genutzt werden.

In der KESO-Konfiguration ist es, wie in Kapitel 5.1 bereits erläutert, möglich, mehrere Mikrocontroller und die verwendeten Netzwerke zu beschreiben. Die Konfiguration kann dazu mit dem *KESO Graphic Editor* (KGE) erstellt werden, welcher eine schematische Darstellung der Konfiguration bietet. Die Abbildung 6.7 zeigt eine mit dem KGE erzeugte Grafik.

Alle drei Mikrocontroller implementieren für die reine Motorsteuerung den gleichen Dienst. In der Konfiguration sind die Dienste mit **Srv:drive0** bis **Srv:drive2** bezeichnet. Diese Dienste erlauben es, die am jeweiligen Mikrocontroller angeschlossenen Sensoren auszulesen und den Motor zu steuern. Einer der Mikrocontroller enthält zusätzlich die beschriebene Steuerlogik. Dieser Mikrocontroller ist in der Darstellung der **Motor3**, welcher alle drei Dienste importiert.

Durch eine einfache Umkonfiguration kann die Steuerlogik zum Zeitpunkt der Integration von einem Knoten auf einen anderen verlagert werden. Die Abbildung 6.8 zeigt eine

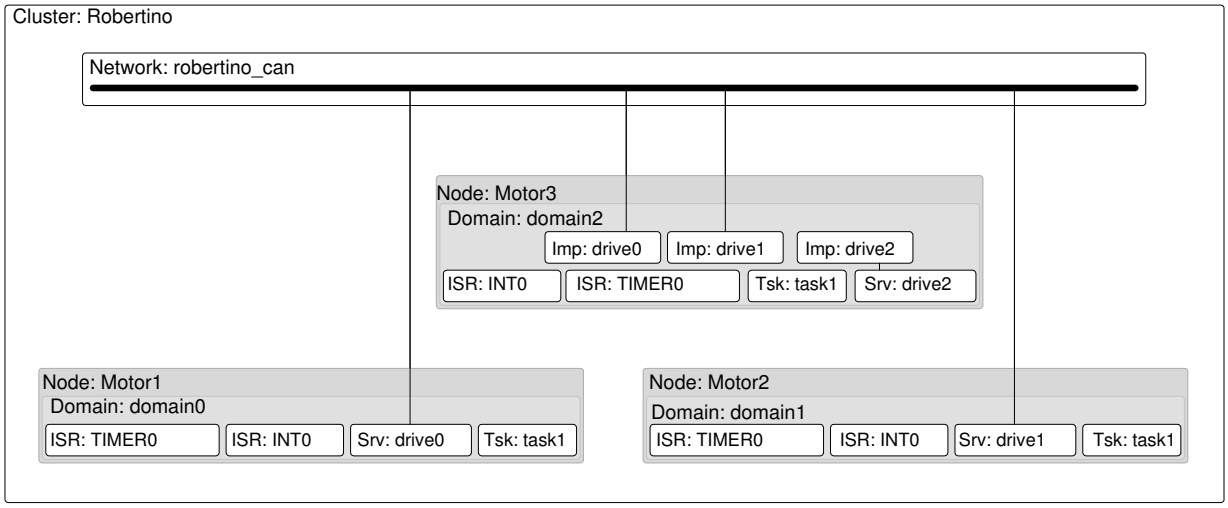

Abbildung 6.7: Konfiguration der Robertinosteuerung mit der Steuerlogik auf einem der Motorsteuergeräte.

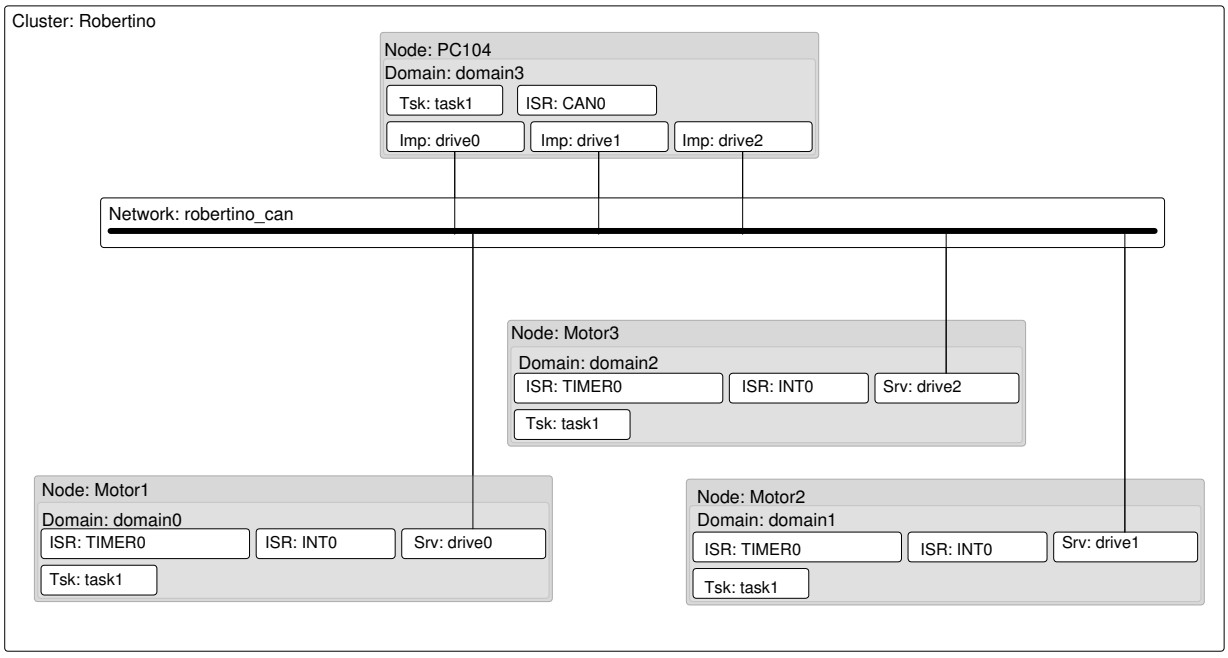

Abbildung 6.8: Konfiguration der Robertinosteuerung mit der Steuerlogik auf dem PC104.

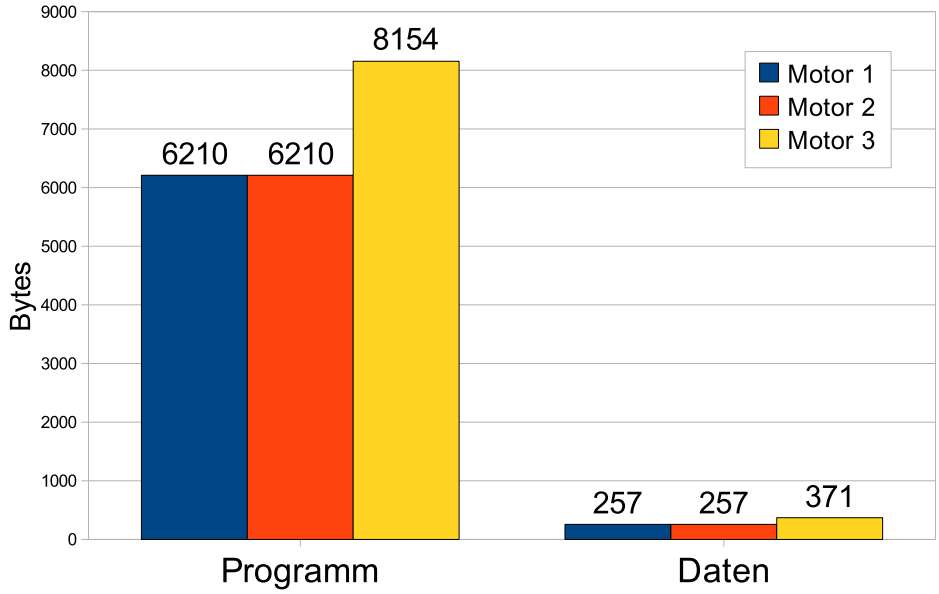

Abbildung 6.9: Programm- und Datenspeicherbedarf

Konfiguration, bei der die Steuerlogik auf den Industrie-PC verlagert wurde. Durch eine solche Konfiguration stünden für die Steuerlogik weitere Systemressourcen zur Verfügung.

Speicherbedarf der Anwendung

Die Abbildung 6.9 zeigt den benötigten Speicherbedarf für Programm und Daten. Der vorhandene Programmspeicher von 8200 Byte wird auf dem Mikrocontroller, der die Steuerlogik enthält, bis auf 46 Byte ausgeschöpft. Die Angaben für den Datenspeicherbedarf verstehen sich ohne den Bedarf für den Methodenstack. Für den Methodenstack verbleiben noch 149 Byte von den 520 Byte Arbeitsspeicher, was für die Anwendung mehr als ausreichend ist. Wird die Steuerlogik auf den Industrie-PC verlagert, so haben alle drei Motorsteuerungen den gleichen Programm- und Datenspeicherbedarf.

Skalierbarkeit und Flexibilität

Mit dem Robertino als zweite prototypische Anwendung wurde anhand einer nicht trivialen Anwendung gezeigt, dass es auch möglich ist, auf eingebetteten Systemen mit 8-Bit-Mikrocontrollern KESO als eine sichere Laufzeitumgebung zu verwenden. Die Anwendung wurde in Java vollständig neu als objektorientierte Anwendung entwickelt.

Die KESO-Laufzeitumgebung ermöglicht es darüber hinaus, einzelne Funktionen flexibel zwischen Mikrocontrollern zu verschieben, solange die Anwendungssoftware nicht direkt eine Hardwarekomponente nutzt. Für die Rekonfiguration muss die Anwendungssoftware nur als Java-Bytecode vorhanden sein. Die eigentlichen Quellen der Anwendung werden nicht benötigt, was neben einer robusteren Software einen weiteren Vorteil der Architektur zeigt.

Kapitel 7

Zusammenfassung

7.1 Herausforderungen

Eingebettete Systeme werden oft in großen Stückzahlen produziert, und die Kosten für jedes gefertigte Stück sind ein entscheidender Faktor für den zu erzielenden wirtschaftlichen Gewinn. Es werden daher bevorzugt kleine und kostengünstige Mikrocontroller mit einer Wortbreite von 8 oder 16 Bit und nur wenigen Kilobyte an Arbeitsspeicher in diesem Bereich eingesetzt.

Die Software für diese Systeme wird überwiegend in C und Assembler entwickelt. Dieser Ansatz ermöglicht die Umsetzung eines möglichst großen Funktionsumfanges, bei gleichzeitig geringen Speicheranforderungen.

Neben den reinen Hardwarekosten werden jedoch auch weitere Möglichkeiten zur Kostenreduktion gesucht und genutzt. Ein gutes Beispiel hierfür sind die eingebetteten Systeme im Automobil. Durch die Kooperation von mehreren Herstellern mit OSEK/VDX wurde eine Spezifikation für eine Systemsoftware geschaffen, die die Wiederverwendbarkeit von Software im Automobil fördert und damit auch das Ziel einer Kostenreduktion verfolgte.

Die Steuergeräte im Automobil bilden ein festes Netzwerk aus Mikrocontrollern, welche im Zusammenspiel eine große Anzahl von Aufgaben gemeinsam erfüllen. Die Anzahl von Mikrocontrollern hat dabei in den letzten Jahren im Automobil stark zugenommen. Es ist keine Seltenheit, dass 40–80 Mikrocontroller pro Fahrzeug verbaut werden — und die Tendenz ist weiter steigend. Es werden inzwischen Anstrengungen unternommen, dieser Tendenz entgegenzuwirken, da die große Anzahl von kleinen Mikrocontrollern die Kosten für die Vernetzung steigen lässt.

Durch das Loslösen der Software von ihren Hardwarekomponenten soll es in Zukunft möglich werden, die Software möglichst frei zu platzieren. So wird es möglich, die

Software abhängig von ihrer Funktionalität und den örtlichen Gegebenheiten sinnvoll zu platzieren. Zum Beispiel kann die Steuerlogik für die Messung des Reifendrucks auf einem Steuergrät zusammen mit der Steuerlogik für den Blinker ihre Aufgabe erfüllen.

Durch eine freie Positionierung von Softwarekomponenten können bestehende Hardwareressourcen besser genutzt werden. Auch größere Mikrocontroller sind denkbar, die die Aufgaben für mehrere kleine Mikrocontroller übernehmen, was unter Umständen preisgünstiger ist.

Das Zusammenführen von Softwarekomponenten unterschiedlicher Hersteller auf einem Mikrocontroller führt jedoch auch zu neuen Herausforderungen an die Systemsoftware. So fehlt auf den Mikrocontrollern in der Regel die Möglichkeit, den Arbeitsspeicher vor fehlerhaften und ungewollten Speicherzugriffen zu schützen. Programmierfehler in einer Softwarekomponente gefährden so die Stabilität der gesamten auf einem Controller laufenden Software.

Ein Ziel dieser Arbeit war es daher, einen Speicherschutz zu entwickeln, welcher die besonderen Anforderungen des Anwendungsgebietes berücksichtigt. Folgende Anforderungen wurden besonders berücksichtigt:

Geringer Ressourcenbedarf: Der zusätzliche Bedarf an Hardwareressourcen für den Speicherschutz, muss so gering wie möglich ausfallen, um den Einsatz auch auf möglichst preiswerter Hardware zu ermöglichen.

Hardwareunabhängigkeit: Der Speicherschutz muss auf einer Vielzahl von unterschiedlichen Architekturen realisierbar sein, da je nach Anforderung der Anwendung unterschiedliche Mikrocontroller und Prozessorarchitekturen zum Einsatz kommen.

Echtzeitfähigkeit: Die mit dem Speicherschutz verbundenen Algorithmen und Aktionen müssen ein zeitlich vorhersagbares Verhalten aufweisen, eingebettete Systeme überwiegend Echtzeitanforderungen erfüllen müssen.

Als Ansatz wurde eine rein softwarebasierte Lösung verfolgt, da diese auf den unterschiedlichen Mikrocontrollern umsetzbar ist und keine zusätzliche Unterstützung durch die Hardware benötigt. Als Grundlage diente dabei ein OSEK/VDX-basiertes Betriebssystem, da vergleichbare Systeme im Automobilbereich bereits etabliert sind.

7.2 Geleisteter Beitrag

Um die Ziele der Arbeit zu erreichen wurde eine Java Laufzeitumgebung entwickelt, die es ermöglicht, mehrere Anwendung voneinander isoliert auf einem OSEK/VDX-System

laufen zu lassen. Die Anwendungen werden hierzu auf einzelne Schutzräume aufgeteilt, die als Domänen bezeichnet werden. Ein Domäne entspricht von der Funktion her einer Schutzdomäne, wie sie im Bereich des hardwarebasierten Speicherschutzes bekannt ist [35, 45].

Jede Domäne stellt aus der Sicht der Anwendung eine eigenständige JVM dar, mit einer eigenständigen Speicherverwaltung und eigenen globalen Variablen. Die Typsicherheit vom Java Bytecode stellt sicher, dass die Anwendung keinen Speicherzugriff auf eine ungültige Speicherstelle durchführen kann. Die zusätzliche Unterteilung in Domänen verhindert eine unbeabsichtigte Veränderung von globalen Zuständen in einer anderen Anwendung und ermöglicht es, mehrere Instanzen von einer Anwendungssoftware auf dem gleichen Mikrocontroller ablaufen zu lassen.

Zur Kommunikation zwischen den Domänen implementiert KESO Portale. Bei Portalen handelt es sich um leichtgewichtige Fernmethodenaufrufe. Die Domäne bietet hierzu eine Schnittstelle als Dienst an, der in einer anderen Domäne über ein Stellvertreterobjekt genutzt werden kann. Das Stellvertreterobjekt wird dafür über einen Namensdienst angefordert.

Im Gegensatz zu einem einfachen Java-Methodenaufruf werden die Parameter als Kopie und nicht als Referenz übergeben. Durch dieses Vorgehen entstehen keine Objektreferenzen, die über Domänengrenzen hinweg verweisen. Die Stellvertreterobjekte werden automatisch beim Übersetzungsvorgang erzeugt und sind auf einen geringen Speicherbedarf ausgerichtet.

Die Protale in KESO sind zu den Portalen in JX [41, 102] vergleichbar. Der lokale Aufruf wurde bei KESO noch schlanker umgesetzt, so dass er nur geringfühgig teuerer ausfällt als ein regulärer Methodenaufruf.

Die Architektur kann auf mehrere Mikrocontroller die einen festen zusammenhängenden Verbund bilden ausgeweitet werden. Die Domänen werden dabei einem Knoten im Verbunden zu gewiesen. Durch Umkonfiguration können die Domänen verlagert werden. Die Kommunikation zwischen den Domänen erfolgt dabei weiterhin über die Protale, welche die Aufrufdaten verpacken und über konfigurierbare Netzwerke weiterleiten. Dienste, die Portale anbieten, können über eine Globalennamensdienst angesprochen werden.

Um ohne zusätzlichen Mehraufwand mit bestehenden nativen Bibliotheken und der Hardware kommunizieren zu können, wurde das *KESO native Interface* (KNI) entworfen. Das KNI kann im Gegensatz zum standardkonformen JNI in den Übersetzungvorgang eingreifen und so die gewünschte Funktionalität direkt in die Anwendung einweben.

Das Einweben von Anweisungen direkt in den Programmablauf ist eine Technik, die auch der von C bekannte Preprozessor leistet. Im Gegensatz zu diesem ist es beim KNI jedoch nur möglich, an den Punkten von Methodenaufrufen auf die Anwendung einzuwirken,

und die Instrumentierung ist in einer Klasse separat von der eigentlichen Anwendung beschrieben.

In dieser Hinsicht ist der Ansatz eher mit Techniken, die auch von AOP genutzt werden, vergleichbar [43, 57, 94]. Im Gegensatz zu einem AOP-orientierten Betriebssystem wie Ciao [67] wurde die KNI-Schnittstelle bei KESO nicht dazu verwendet, die Software in einzelne Belange zu untergliedern.

Die Unterteilung der Anwendungssoftware in getrennte Domänen ermöglicht für jede Domäne eine eigenständige und angepasste Speicherverwaltung. Im Kapitel 3.3 wurden hierfür drei unterschiedliche Speicherverwaltungen vorgestellt, die im Rahmen der Arbeit entwickelt wurden. Diese erfüllen unterschiedliche Anforderungen im Hinblick auf Speicherbedarf und Echtzeitfähigkeit.

Besonders die minimale Speicherverwaltung verfolgt einen neuen Ansatz, der mit dem ScopedMemory aus der RTSJ [25] vergleichbar ist. Der für die Speicherverwaltung relevante Bereich wird jedoch fest auf die Grenzen einer Domäne ausgedehnt, was die Umsetzung wesentlich vereinfacht und im Gegensatz dazu nur geringe Nachteile hat, da eine Domäne auch beliebig klein sein kann.

Der Funktionsumfang der JVM wurde an die Anforderungen und das zu Grunde liegende OSEK/VDX-System angepasst. Zum Einen ist dies nötig um der Anforderung eines möglichst geringen Speicherbedarf gerecht zu werden, zum Anderen sollten die von OSEK/VDX gebotenen Konzepte, wie das *Immediate Ceiling Priority Protocol* (ICPP), nicht zerstört werden.

Die Anpassung der JVM an die Anforderungen eines Einsatzgebietes ist nicht neu, sondern ist in der *Java 2 Micro Edition* (J2ME) vorgesehen. Auch wurde im Rahmen des AJACS-Projektes [8] bereits eine JVM auf einem OSEK/VDX-System realisiert. KESO bietet jedoch darüber hinaus die Möglichkeit von mehreren JVM-Instanzen und besitzt eine konsequente Anpassung der JVM auf die Anforderungen des Einsatzgebietes und Funktionalität von OSEK/VDX.

Um die Zielsetzung der Arbeit zu erreichen und einen möglichst geringen Speicherbedarf zu erzielen, wurde nicht nur die Funktionalität der JVM eingeschränkt, sondern es wurden auch die internen Datenstrukturen der JVM speziell für die Anforderungen entwickelt.

Der bidirektionale Objektaufbau mit einem in der Größe variablen Objektkopf von minimal einem Byte ist neuartig und ist das Resultat einer konsequente Umsetzung von bestehenden Arbeiten [39, 87] und der Konzentration auf die benötigte Funktionalität. Die im Objektkopf gespeicherte Klassenkennung erlaubt eine direkte Typüberprüfung der Objektklasse und mit nur einem weiteren Speicherzugriff die Bestimmung von zusätzlich benötigten Informationen, wie die Objektgröße und die Anzahl der im Objekt gespeicherten Objektreferenzen. Auch die für KESO entwickelten horizontalen Methodentabellen nutzen die Klassenkennung als Index, um zur Laufzeit das Sprungziel eines Methodenaufrufes zu ermitteln.

Da jede Domäne eine eigene JVM darstellt, kann die Anwendungssoftware mit jedem Übersetzer, der standardkonformen Java Bytecode erzeugt, übersetzt werden. Der Java-Bytecode ermöglicht es, dass die Anwendung in einer von der Hardware unabhängigen Form von den Zulieferern geliefert werden kann, ohne dass diese die eigentlichen Quellen der Anwendung weitergeben müssen.

Der im Rahmen der Arbeit entwickelte Übersetzer erzeugt aus dem Bytecode der Anwendung und einer Konfiguration ein C-Programm und eine passende Konfiguration für OSEK/VDX. Durch den Umweg über die C-Programmiersprache passt sich KESO in den Übersetzungsprozess von OSEK/VDX ein. Außerdem muss durch diesen Zwischenschritt der Übersetzer nicht an die Hardware angepasst werden, und die bestehende Technologie zur Erzeugung von effizienten Maschinenprogrammen durch den C-Übersetzer kann wiederverwendet werden.

Der Bytecode wird während der Übersetzung analysiert, und die Datenstrukturen werden auf die Anforderungen der Anwendung abgestimmt. Zusätzlich erfolgen Optimierungen der Anwendungssoftware, die aufgrund der globalen Sicht und der Typsicherheit von Java einfacher möglich sind als dies im C-Übersetzer der Fall wäre.

Es gibt bereits Werkzeuge, die Java Bytecode vor der Laufzeit nach C übersetzen [86, 100]. Die Entwicklung eines weiteren Übersetzers war nötig, um einen geringeren Speicherverbrauch zu erzielen.

7.3 Erreichte Ziele

Mit KESO wurde eine Laufzeitumgebung entwickelt, die es ermöglicht, mehrere Anwendungen in voneinander isolierten JVMs auf einem OSEK/VDX-System auszuführen. Die Aufteilung der Anwendungen auf mehrere JVMs ermöglicht die mehrfache Instantiierung von Anwendungen und verhindert die Ausbreitung von Programmierfehlern, wie es ein hardwarebasierter Speicherschutz mit getrennten Adressräumen bieten würde.

Um eine Laufzeitumgebung, bestehend aus mehreren JVMs auf Mikrocontrollern, zu realisieren, die im Einsatzgebiet von kleinsten eingebetten Systemen verwendet werden, ist es nötig, besonderen Anforderungen zu genügen. Eine solche Laufzeitumgebung muss einen eingeschränkten Ressourcenbedarf bieten, auf unterschiedlicher Hardware laufen und Echtzeitanforderungen genügen.

Der eingeschränkte Ressourcenbedarf bezieht sich dabei auf einen möglichst geringen Bedarf an Arbeits- und Programmspeicher. Dieses Ziel wurde unter anderem durch die Ausrichtung auf eine statische Konfiguration der Laufzeitumgebung erreicht. Die statische Ausrichtung ermöglichte es, schlankere Datenstrukturen zu entwickeln, die bei ansonsten gleicher Funktionalität einen geringeren Speicherbedarf besitzen als in eine

JVM mit einem dynamischen Klassenlader. Die statische Konfiguration ermöglichte es durch Anwendungsanalyse, die JVM an die Anforderungen anzupassen.

Die Portierung einer C-Anwendung auf die entwickelte Laufzeitumgebung hat gezeigt, dass der zusätzliche Speicherverbrauch bei gleicher Funktionalität in einem Rahmen bleibt, der durch die bessere Übersetzertechnik ausgeglichen wird.

Die Laufzeitumgebung wurde in sehr unterschiedlichen Architekturen eingesetzt. In der Entwicklungsphase und für Vergleichsmessungen diente ein Unix mit einem Pentium-kompatiblen Prozessor als Grundlage. Für die Prototypen wurde Tricore TC1796 und ein AVR Atmega8535 verwendet. Damit wurde ein sehr weiter Bereich an möglichen Architekturen abgedeckt.

Auch Echtzeitanforderungen wurden bei der Entwicklung berücksichtigt. Die von der JVM benötigten Hilfsfunktionen besitzen mindestens eine Variante mit konstanter Laufzeit. Auch wurden die von OSEK/VDX bekannten Konzepte zur Synchronisation an die Anwendungsschicht weitergereicht, und es wurde bewusst vermieden, die guten Eigenschaften von OSEK/VDX zu zerstören Damit erreicht die entwickelte Laufzeitumgebung die am Anfang gesetzten Ziele.

7.4 Ausblick

Mit KESO wurde der Bereich der Steuergeräte für den Einsatz von Java-Bytecode erschlossen und so die Möglichkeit geschaffen, die für Java entwickelten Werkzeuge auch in diesem Bereich zu nutzen. Auch wenn die Vielzahl an Werkzeugen und die Verbreitung von Java für den Java-Bytecode als Zwischendarstellung sprechen, muss man eingestehen, dass er nicht den idealen Zwischencode für Steuergeräte darstellt.

Der Java-Bytecode besitzt keine Unterstützung für Ganzzahlen unterhalb von 32 Bit und keine vorzeichenlosen Ganzzahlen. Es gibt aus dem Objekt keinen strukturierten Datentyp. Diesen Nachteil kann man durch Analyse und Übersetzertechniken zum Teil wieder ausgleichen, eine flexiblere oder speziell für Steuergeräte entwickelte Zwischendarstellung wäre für das Erreichen der Ziele wahrscheinlich noch besser geeignet gewesen.

Der im Vergleich zu anderen JVM geringe Speicherbedarf wurde unter anderem dadurch erreicht, dass möglichst viele Information über die statische Konfiguration des Gesamtsystems genutzt wurden. Bedeutet dieses im Umkehrschluss, dass der Preis, der für das dynamische Laden von Klassen gezahlt werden muss, entsprechend groß ist und daher auf kleinen Mikrocontrollern das Laden von Klassen nicht umsetzbar ist? Dieses ist nicht zwingend der Fall.

Integriert man den Klassenlader auf dem Mikrocontroller, so ist der bei KESO verfolgte Ansatz nicht mehr möglich. Das bedeutet jedoch nicht, dass es nicht möglich sein sollte,

durch einen externen Klassenlader und durch einen leichtgewichtigen Updatemechanismus einen Klassenlader für KESO zu entwickeln.

7.5 Abschließende Anmerkung

Die Anzahl von Schaltelementen auf einem Computerchip verdoppelt sich alle 18 Monate. Aufgrund dieser Entwicklung können Rechner immer größere Datenmengen aufnehmen. Die Geschwindigkeit, mit der diese Entwicklung voranschreitet, wurde 1965 durch Gordon Moore in dem nach ihm benannten „Mooresches Gesetz“ das erste Mal postuliert und 1975 nach unten korrigiert [70]. Auf Grund dieser Gesetzmäßigkeit könnte man vermuten, dass der Speicherbedarf einer Anwendung immer weniger wichtig wird, da sich in absehbarer Zeit das Angebot an Speicher verdoppelt.

Es gibt jedoch Gründe, die dieser Sichtweise widersprechen. Ein Grund sind die Stückkosten, die in der Arbeit bereits mehrfach angeführt wurden und die im Massenmarkt entscheidend sind. Weitere Gründe, die einem Speicherausbau entgegenwirken, sind der höhere Energieverbrauch, größere Abmessungen und steigende Komplexität von Hardware und Software.

Die steigende Packungsdichte von Schaltelementen wird daher nicht automatisch in größere Speicher investiert, sondern ermöglicht neue Einsatzgebiete. Beispiele sind hier kleine Sensorknoten, die über Jahre hinweg Messungen durchführen, oder „intelligente“ Kleidung.

Die Herausforderung, schlanke Software zu entwickeln, ist trotz dem „Mooreschen Gesetzes“ weiterhin wichtig. KESO und die im Rahmen dieser Arbeit entwickelten Konzepte werden daher auch in Zukunft einen Beitrag leisten.

Kapitel 8

Summary

8.1 Challenges

Embedded systems are often produced in great numbers, and the cost of every item is a deciding factor for the required profit to be achieved. Hence the preferred microcontrollers in this area are small and cost saving with a word width of 8 or 16 bit and a working memory of just a few kilobytes.

The software for these systems is mainly developed in C and Assembler. This enables the implementation of as high a number of features as possible while the memory requirements are being kept low.

Further possibilities for reducing cost, apart from solely hardware, are required and used. One good example is the embedded systems in cars. The cooperation of several manufacturers with OSEK/VDX enabled the specification of system software that promotes the reusability of software in cars — thereby at the same time reducing cost.

Control devices in a car are parts of a fixed network of microcontrollers which perform together a great number of tasks. In the last few years, the number of microcontrollers in cars has risen enormously. It is not uncommon for 40 to 80 microcontrollers to be used per car — and the trend continues upward. There are now efforts to counteract this trend as the cost for the networks is rising due to the number of microcontrollers.

By detaching the software from the hardware components, it should be possible to place the software as freely as possible. Hence it will be possible to place the software sensibly according to its functionality and its local conditions. For example, the control logic for the measurement of the tyre pressure can fulfill its task together with the control logic for the indicator on one control unit.

By positioning software components freely, existing hardware resources can be used more efficiently. Bigger microcontrollers, which are able to take over the tasks of several small microcontrollers, are also conceivable. This may also reduce costs.

Merging software components of different manufacturers in one microcontroller, however, leads to new challenges for the system software. Microcontrollers usually lack the ability to protect the working memory from uncontrolled or unintended memory access. Programming errors of one software component might hence endanger the stability of the entire software.

One objective of this project was therefore to develop a memory protection that fulfills the specific requirements of this application area. The following requirements were particularly considered:

Reducing resources: Any additional hardware requirement for the memory protection must be as low as possible to enable operation on inexpensive hardware.

Hardware independence: The memory protection has to be feasible on a number of different architectures, as different microcontrollers and processors have to be used according to the requirements.

Real-time capability: The algorithms and actions in connection with the memory protection have to be predictable, as embedded systems have to fulfill real-time tasks.

The approach was through a purely software-based solution. This is implementable on different microcontrollers and does not require any additional support by the hardware. The basis for this forms an OSEK/VDX- based operating system, as comparable systems are already established in the car industry.

8.2 Contribution

In order meet the objectives of this project, a Java runtime environment was delevoped that enabled the concurrent running of several applications on an OSEK/VDX system while being isolated. The applications are divided onto several protected domains. A domain functions as a protection domain similarly to the protection domain known for hardware-based protection domains [35, 45].

Every domain represents an independent JVM, with an independent memory management and individual global variables. The security of the Java bytecode ensures that the application cannot access the memory on an invalid memory location. The additional division into domains prohibits an unintentional change of global conditions in another

application and enables the running of several instances of one application software on the same microcontroller.

To enable communication between the domains, KESO implements portals. Portals are implementing a lightweight remote method invocation. The domain offers an interface as a service that can be used via a proxy object in another domain. The proxy object can be requested via a name service.

Contrary to a normal method invocation the parameters are transferred as copies, not as references. This prohibits object references from overstepping domain limits. The proxy objects are created automatically during the translation process and require only a small amount of memory capacity.

The portals in KESO are comparable to the portals in JX [41, 102]. The local method invocation follows a lean implementation with KESO and is therefore only marginally more expensive than a regular method invocation.

The architectur can be expanded to several microcontrollers of fixed composition. The domains are assigned to a node in the composition. The domains can be transferred by reconfiguration. Communication between domains is still via portals.

In order to be able to communicate with existing native libraries and the hardware without additional cost, KESO native interface (KNI) was designed. KNI, unlike the standard JNI, is able to intervene in the translation process and in this way weaving the required functionality directly into the application.

The weaving in of commands directly into the programme is a technology which is also being performed by the C preprocessor. KNI, however, only offers that possibility at the point of the method invocation, and the implementation is encapsulated in a class separate from the actual application.

In this regard, the approach resembles technologies that are also used by AOP [43, 57, 94]. In contrast to an AOP-orientated operating system like Ciao [67] the KNI interface with KESO is not used to encapsulate the different concurns within the software.

The separation of the application software into separate domains allows each domain its separate and adapted memory management. In chapter 3.3 three different memory managements were introduced that have been developed within this project. They meet different requirements regarding memory requirements and real-time capability.

The minimal memory requirement in particular pursues a new approach, comparable with the ScopedMemory of RTSJ [25]. The relevant area for the memory management however is fixed to the limits of the domain, thereby facilitating the implementation. The disadvantages are minor, as a domain can be as small as required.

The functionality of the JVM was adapted to the requirements and the underlying OSEK/VDX system. This was necessary to make the memory capacity as small as

possible on the one hand, and to ensure that OSEK/VDX concepts such as the Immediate Ceiling Priority Protocol (ICPP) would not be destroyed on the other hand.

The adaptation of the JVM to the requirements of an operational area is nothing new, but is indeed provided in the Java 2 Micro Edition (J2ME). A JVM on a OSEK/VDX system has also already been created under the AJACS project [8]. Over and beyond that, KESO offers the possibility of several JVM instances and has been adapted to the requirements of the operation area and the functionality of OSEK/VDX.

To meet the objectives of this project and minimalise memory requirements, not only was the functionality of the JVM restricted, but internal data structures specific to the requirements were created.

The bi-directional object structure with a variable object header of a minimum of 1 byte is novel and is the result of a consistent implementation of existing research [39, 87] and focus on the required functionality. The saved class ID allows an immediate checking of the object type and just one further memory access allows the determination of further information such as object size and the number of object references saved in the object. The novel horizontal method tables also use the class ID for address resolution.

As each domain represents its own JVM, the application software can be translated with every translator that generate standard Java bytecode. The Java bytecode makes it possible for the application to be delivered independent of hardware. The actual sources do not have to be disclosed.

The translator which has been created for this project produces a C- programme and a OSEK/VDX compatible configuration. KESO therefore fits into the translation process of OSEK/VDX via the C-programming language. In addition to this, the translator does not need to be adapted to the hardware and the existing compiler technology can be reused to create efficient machine programmes.

During the translation, the bytecode is being analysed, and the data structure matched to the requirements of the application. In addition to that, the application software is being optimized. Due to the global sight and type safety of Java, this is easier than it would be in the c-compiler.

Tools that translate the Java bytecode to C are already in existence [86, 100]. The creation of a new translator was however necessary to meet the demand of a smaller memory capacity.

8.3 Achieved objectives

KESO is a newly created runtime environment that enables several applications to be run simultaneously in isolated JVMs on an OSEK/VDX system. Splitting the applications

between several JVMs allows the multiple instantiation of applications and prevents the spread of programming errors as would be the case with a hardware based memory protection.

In order to create a runtime environment consisting of several JVMs on microcontrollers to be operated on deeply embedded systems, specific requirements have to be met. The runtime environment must use restricted resources, must be able to run on different hardware systems and must meet real-time requirements. Restricted resource requirements in this context means as small a requirement as possible of RAM and program memory. This objective was met by the creation of a static configuration of the runtime environment. The static configuration made possible the creation of leaner data structures. Whilst maintaining functionality, they require a smaller memory capacity than a JVM with dynamic class loader. The static configuration, by application analysis, allowed the JVM to be adapted to the requirements.

Porting of a C-application onto the runtime environment proved that the additional memory use is offset by the improved translation technology while maintaining equal functionality.

The runtime environment was used in various layouts. A Unix with a Pentium-compatible processor served as a base during the development phase and for comparative measurements; for the prototypes a Tricore TC1796 and an AVR Atmega8535 were used. Hence a wide choice of possible layouts was covered.

Real-time requirements were considered during the development at all times. Auxiliary functions required by the JVM contain at least one variant with constant runtime. OSEK/VDX-known concepts of synchronization were passed on to the application layer, and their useful properties retained. Hence the created runtime environment meets the required objectives.

8.4 Outlook

KESO opened up the area of control units for Java bytecode and it is therefore possible to use the Java tools in this area. Even though the multitude of tools and the common use of Java are an argument for the Java bytecode, it has to be admitted that it is not the ideal intermediate code for control units.

Java bytecode has no support for whole numbers under 32 bit and no undisseminated whole numbers. There is no structured data type out of the object. This disadvantage can be partly offset by analysis and translation technology; however, it would probably have been better to have used a more flexible intermediate representation, or one specifically created for control units.

The comparably low memory requirements were achieved partly by using as much information as possible via the static configuration of the overall system. Does this then mean that the price for the dynamic loading of classes is accordingly high and therefore the loading of classes on small microcontrollers cannot be implemented? This is not necessarily the case.

If the class loader is integrated into the microcontroller, the approach pursued by KESO is no longer possible. It would however not be impossible to create a class loader for KESO by designing an external class loader and a lightweight update mechanism.

8.5 Concluding Remarks

The amount of switching elements on a computer chip doubles every 18 months. As a result of this development, computers can hold ever larger amounts of data. The speed of this development has been described for the first time in 1965 by Gordon Moore ('Moore´s Law'), and has been revised downwards in 1975 [70]. According to this law, the assumption would be that the memory capability of an application loses in importance as the memory capability will double in the foreseeable future.

There are, however, reasons that contradict this view. One reason is the unit cost, as mentioned before, that is a deciding factor in the mass market. Further reasons against the expansion of memory are higher energy consumption, larger size and increasing complexity of hardware and software.

The increasing packing density of switching elements is therefore not automatically invested in bigger memory capacities but allows new operational areas. Examples would be small sensor nodes that carry out measurements over years, or 'intelligent' clothing.

The challenge of creating lean software is important in spite of 'Moore´s Law'. KESO and the concepts created in this project will therefore be a valid contribution even in the future.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[1] *ISO/IEC 23271:2003: Information technology — Common Language Infrastructurer (CLI) Partitions I to VI*. Int. Symp. on, Geneva, Switzerland, 2006.

[2] Fiaso project homepage, 2008. `http://os.inf.tu-dresden.de/fiasco/`.

[3] L4Ka: Hazelnut project homepage, 2008. `http://l4ka.org/projects/hazelnut/eval.php`.

[4] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. MACH: A new kernel foundation for UNIX development. In *USENIX Summer Conference*, pages 93–113. USENIX, 1986.

[5] O. Agesen, D. Detlefs, and J. E. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. *SIGPLAN Not.*, 33(5):269–279, 1998.

[6] G. Agosta, S. C. Reghizzi, and G. Svelto. Jelatine: a virtual machine for small embedded systems. In *JTRES '06: 4th Int. W'shop on Java Technologies for real-time & embedded Systems*, pages 170–177, New York, NY, USA, 2006. ACM Press.

[7] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 1–10, New York, NY, USA, 2006. ACM.

[8] AJACS: Applying Java to automotive control systems concluding paper V2.0, July 2002. `http://www.ajacs.org/`.

[9] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–239, Feb. 2000.

[10] C. S. Ananian and M. Rinard. Data size optimizations for java programs. In *2003 Joint LCTES & SCOPES Conferences (LCTES/SCOPES '03)*, pages 59–68, New York, NY, USA, 2003. ACM.

[11] A. W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, New York, NY, USA, 1998.

[12] A. W. Appel, J. R. Ellis, and K. Li. Real-Time Concurrent Collection on Stock Multiprocessors. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '88)*, pages 11–20, 1988.

[13] C. Artho and A. Biere. Subroutine inlining and bytecode abstraction to simplify static and dynamic analysis. *Electronic Notes in Theoretical Computer Science*, 141(1):109–128, Dec. 2005.

[14] AUTOSAR homepage. `http://www.autosar.org/`, visited 2009-03-26.

[15] G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Trans. Program. Lang. Syst.*, 27(4):583–630, 2005.

[16] D. F. Bacon, P. Cheng, and V. T. Rajan. The Metronome: A simpler approach to garbage collection in real-time systems. In R. Meersman and Z. Tari, editors, *Proceedings of the OTM Workshops: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume 2889 of *LNCS*, pages 466–478. Springer, Nov. 2003.

[17] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. *SIGPLAN Not.*, 38(1):285–298, Jan. 2003.

[18] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In *16th Eur. Conf. on OOP (ECOOP '02)*, pages 111–132, 2002.

[19] D. F. Bacon, R. B. Konuru, C. Murthy, and M. J. Serrano. Thin Locks: Featherweight Synchronization for Java. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '98)*, pages 258–268, 1998.

[20] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, 1996.

[21] H. G. Baker. The Treadmill: Real-time garbage collection without motion sickness. *SIGPLAN Not.*, 27(3):66–70, 1992.

[22] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN — an extensible microkernel for application-specific operating system services. In *ACM SIGOPS European Workshop*, pages 68–71, 1994.

[23] B. N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E. G. Sirer. Protection is a software issue. In *5th W'shop on Hot Topics in Operating Systems (HotOS-V)*, page 62, Washington, DC, USA, 1995. IEEE.

[24] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[25] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. AW, 1st edition, Jan. 2000.

[26] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262, New York, NY, USA, 1984. ACM.

[27] Y. Chang and A. Wellings. Hard real-time hybrid garbage collection with low memory requirements. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 77–88, Washington, DC, USA, 2006. IEEE Computer Society.

[28] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *18th ACM Symp. on OS Principles (SOSP '01)*, pages 73–88, New York, NY, USA, 2001. ACM.

[29] M. Condict, D. Bolinger, D. Mitchell, and E. McManus. Microkernel modularity with integrated kernel performance. Technical report, OSF Research Institute, Cambridge, MA., Apr. 1994.

[30] M. E. Conway. Proposal for an UNCOL. *CACM*, 1(10):5–8, 1958.

[31] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. *DARPA Information Survivability Conference & Exposition*, 02:1119, 2000.

[32] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[33] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct 1991.

[34] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *LNCS*, 952:77–101, 1995.

[35] J. B. Dennis. Segmentation and the design of multiprogrammed computer systems. *JACM*, 12(4):589–602, 1965.

[36] S. Dorward, R. Pike, D. Presotto, D. M. Ritchie, H. Trickey, and P. Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[37] M. F. Fernández. Simple and effective link-time optimization of modula-3 programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '95)*, pages 103–115, New York, NY, USA, 1995. ACM.

[38] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. *Softw. Pract. Exper.*, 30(3):199–232, 2000.

[39] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 27–40, Apr. 2001.

[40] J. Y. Gil and Y. Zibin. Efficient subtyping tests with PQ-encoding. *ACM Trans. Program. Lang. Syst.*, 27(5):819–856, 2005.

[41] M. Golm. *The Structure of a Type-Safe Operating System*. Dissertation, Friedrich-Alexander University, Erlangen-Nuremberg, Dec. 2002.

[42] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *2002 USENIX TC*, pages 45–58, Berkeley, CA, USA, June 2002. USENIX.

[43] W. Gong and H.-A. Jacobsen. *AspeCt-oriented C Language Specification (Version 0.8)*. Middleware Systems Research Group, Department of Computer Science, University of Toronto, Jan. 2008. `http://www.aspectc.net/acc_manual.pdf`.

[44] J. Gosling. Java intermediate bytecodes: ACM SIGPLAN workshop on intermediate representations (IR'95). *SIGPLAN Not.*, 30(3):111–118, 1993.

[45] R. M. Graham. Protection in an information processing utility. *CACM*, 11(5):365–369, May 1968.

[46] B. Graunitz. 32-bit-MCUs dominieren automotive-applikationen. www.elektroniknet.de, Oct. 2008.

[47] N. Hardy. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985.

[48] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symp. on OS Principles (SOSP '97)*, New York, NY, USA, Oct. 5–8 1997. ACM.

[49] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proc. of the 1998 USENIX Annual Technical Conference*, pages 259–270, 1998.

[50] HIS. OSEK OS extensions for protected applications. Technical report, Daimler-Chrysler AG, June 2003.

[51] W. Hofer, D. Lohmann, and W. Schröder-Preikschat. Concern impact analysis in configurable system software—the AUTOSAR OS case. In *7th AOSD W'shop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)*, pages 1–6, New York, NY, USA, Mar. 2008. ACM.

[52] Java Native Interface Specification 1.1. Sun Microsystems, Aug. 2005. URL: http://tinyurl.com/2wxzf9.

[53] R. K. Johnsson and J. D. Wick. An overview of the Mesa processor architecture. *SIGARCH Comput. Archit. News*, 10(2):20–29, 1982.

[54] M. S. Johnstone. *Non-compacting memory allocation and real-time garbage collection*. PhD thesis, 1997. Supervisor-Paul R. Wilson.

[55] T. B. S. Jr. UNCOL: The myth and the fact. *Annual Review in Automatic Programming*, 2:325–344, 1960.

[56] JSR 1: Real-time Specification for Java. Sun Microsystems JCP, May 2006.

[57] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *15th Eur. Conf. on OOP (ECOOP '01)*, volume 2072 of *LNCS*, pages 327–353. Springer, June 2001.

[58] A. Krall, J. Vitek, and N. Horspool. Near optimal hierarchical encoding of types. In M. Aksit and S. Matsuoka, editors, *11th Eur. Conf. on OOP (ECOOP '97)*, pages 128–145, Finland, 1997. Springer.

[59] R. Kumar, E. Kohler, and M. Srivastava. Harbor: Software-based memory protection for sensor nodes. In *IPSN '07: 6st Int. Conf. on Information Processing in Sensor Networks*, pages 340–349, New York, NY, USA, 2007. ACM.

[60] J2ME building blocks for mobile devices — white paper on KVM and the connected, limited device configuration (CLDC), May 2000. `http://java.sun.com/products/cldc/wp/KVMwp.pdf`.

[61] B. Lampson. Personal distributed computing: The Alto and Ethernet software. In *Proceedings of the ACM Conference on The history of personal workstations*, pages 101–131, New York, NY, USA, 1986. ACM.

[62] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[63] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[64] S. Liang. *The Java™ Native Interface: Programmer's Guide and Specification*. AW, July 1999.

[65] J. Liedtke. On μ-kernel construction. In *15th ACM Symp. on OS Principles (SOSP '95)*, ACM OSR. ACM, Dec. 1995.

[66] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. AW, second edition, 1999.

[67] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, and W. Schröder-Preikschat. Configurable memory protection by aspects. In *4th W'shop on Progr. Lang. and OSes (PLOS '07)*, pages 1–5, New York, NY, USA, Oct. 2007. ACM.

[68] S. Macrakis. From UNCOL to ANDF: Progress in standard intermediate languages. Technical report, Open Software Foundation, 1993.

[69] D. Mcilroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering*, pages 88–98, 1968.

[70] G. E. Moore. Cramming more components onto integrated circuits. pages 56–59, 2000.

[71] P. A. Nelson. A comparison of PASCAL intermediate languages. In *SIGPLAN '79: Proceedings of the 1979 SIGPLAN symposium on Compiler construction*, pages 208–213, New York, NY, USA, 1979. ACM.

[72] D. Novillo. TreeSSA a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[73] OSEK/VDX group homepage. `http://www.osek-vdx.org/`.

[74] OSEK/VDX Group. OSEK implementation language specification 2.5. Technical report, OSEK/VDX Group, 2004. `http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf`, visited 2009-09-09.

[75] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, Feb. 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2009-09-09.

[76] R. Pedersen and M. Schoeberl. Exact roots for a real-time garbage collector. In *JTRES '06: 4th Int. W'shop on Java Technologies for real-time & embedded Systems*, pages 77–84, New York, NY, USA, 2006. ACM Press.

[77] R. Petschacher. Halbleiter als Innovationsmotor im Auto – Semiconductors are Drivers for Automotive Innovations. *e & i Elektrotechnik und Informationstechnik*, 123(10):445–450, Oct. 2006.

[78] G. Phipps. Comparing observed bug and productivity rates for Java and C++. *Softw. Pract. Exper.*, 29(4):345–358, 1999.

[79] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[80] F. Pizlo and J. Vitek. An empirical evaluation of memory management alternatives for real-time Java. In *27th IEEE Int. Symp. on Real-Time Systems (RTSS '06)*, pages 35–46. IEEE, Dec. 2006.

[81] E. Quinn and C. Christiansen. Java Pays – Positively. IDC Bulletin W16212, May 1998.

[82] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An operating system for a personal computer (summary). In *7th ACM Symp. on OS Principles (SOSP '79)*, pages 106–107, New York, NY, USA, 1979. ACM.

[83] OpenRobertino. `http://www.openrobertino.org/`.

[84] S. G. Robertz and R. Henriksson. Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In *2003 Joint LCTES & SCOPES Conferences (LCTES/SCOPES '03)*, pages 93–102, New York, NY, USA, 2003. ACM.

[85] I. Rogers and C. C. Kirkham. JikesNODE and PearColator: A Jikes RVM operating system and legacy code execution environment. In *2nd ECOOP Workshop on Programm Languages and Operating Systems (ECOOP-PLOS '05)*, July 2005.

[86] G. Sally. Embedded java with GCJ. *Linux J.*, 2006(145):5, 2006.

[87] L. K. Schubert, M. A. Papalaskaris, and J. Taugher. Determining type, part, color and time relationships. *IEEE Comp.*, 16(10):53–60, 1983.

[88] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE TC*, 39(9):1175–1185, 1990.

[89] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *17th ACM Symp. on OS Principles (SOSP '99)*, ACM OSR, pages 170–185, New York, NY, USA, 1999. ACM.

[90] F. Siebert. The impact of realtime garbage collection on realtime Java programming. *OO Real-Time Distributed Computing*, 00:33–40, 2004.

[91] F. Siebert. Realtime garbage collection in the jamaicavm 3.0. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 94–103, New York, NY, USA, 2007. ACM.

[92] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *CACM*, 36(2):69–81, Feb. 1993.

[93] Electronic payment product range, 2006. `http://www.gi-de.com/`.

[94] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *40th Int. Conf. on Technology of OO Languages and Systems (TOOLS Pacific '02)*, pages 53–60, Sydney, Australia, Feb. 2002.

[95] Sun Microsystems. Virtual machine specification, Java Card platform version 2.2.2, Mar. 2006.

[96] Java HotSpot, 1999. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html`.

[97] N. Szyperski. *Component Software – Beyond Object-Oriented Programming*. AW, Boston, MA, USA, 2002.

[98] A. S. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson. A practical tool kit for making portable compilers. *CACM*, 26(9):654–660, 1983.

[99] J. Turley. The two percent solution. *embedded.com*, Dec. 2002. `http://www.embedded.com/story/OEG20021217S0039`, visited 2009-03-26.

[100] A. Varma and S. S. Bhattacharyya. Java-through-c compilation: An enabling technology for java in embedded systems. *date*, 3:30161, 2004.

[101] J. Vitek, N. Horspool, and A. Krall. Efficient type inclusion tests. In T. Bloom, editor, *12th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '97)*, pages 142–157, Atlanta, 1997.

[102] C. Wawersich, M. Felser, M. Golm, and J. Kleinöder. The role of IPC in the component-based operating system JX. In *5th ECOOP W'shop on Object Orientation and Operating Systems (ECOOP-OOOS '02)*, pages 43–48, June 2002.

[103] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1992.

[104] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, 1990.

[105] Y. Zibin and J. Gil. Efficient subtyping tests with PQ-encoding. In *16th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '01)*, pages 96–107, 2001.

[106] B. Zorn. Barrier methods for garbage collection, Nov. 1990. Technical Report CUCS -494-90, University of Colorado at Boulder.