Masterarbeit

SILC
Secure Intermediate Language Compiler

Lothar Heinrich

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O.Univ.-Prof.Dipl.-Ing.Dr.techn. Reinhold Weiß

Begutachter: Ao.Univ.-Prof., Dipl.-Ing. Dr.techn. Eugen Brenner

Graz, im September 2007

Master Thesis

SILC
Secure Intermediate Language Compiler

Lothar Heinrich

Institute for Technical Informatics
Graz University of Technology
Head: O.Univ.-Prof.Dipl.-Ing.Dr.techn. Reinhold Weiß

Supervisor: Ao.Univ.-Prof., Dipl.-Ing. Dr.techn. Eugen Brenner

Graz, September 2007

Kurzfassung

Diese Arbeit widmet sich in einer theoretischen Aufarbeitung und mit einer praktischen Implementierung dem Thema Wertebereichsüberprüfung. Ziel ist, implizite Wertebereichsüberprüfungen durch Hardwareerweiterungen zu unterstützen und damit der Verwendung dieser Vorschub zu leisten. Damit soll diese Arbeit helfen, die Qualität von Software zu verbessern. Die These ist, dass bei einer guten Hardwareunterstützung vermehrt Softwarehersteller Hochsprachen verwenden, die durch die konsequente Verwendung von automatischen Wertebereichsüberprüfungen helfen, die Fehlerträchtigkeit von Programmen zu reduzieren.

In einer einleitenden Behandlung der Thematik werden Anwendungsbereiche dieser Technik aufgezeigt. Eine statistische Auswertung untermauert die nach wie vor bestehende Notwendigkeit in diesem Bereich Fortschritte zu erzielen.

Ein Überblick über vergleichbare Arbeiten zeigt den aktuellen Stand der Forschung in diesem Bereich.

Aufbauend auf dem Projekt SCPU wurde die CLI Laufzeitumgebung portable.NET auf eine spezielle Hardwareplattform portiert und in der Weise modifiziert, dass implizite *„bound checks“* durch die eigens dafür geschaffenen Prozessorinstruktionen unterstützt werden.

Abschließend werden gewonnene Erkenntnisse und Resultate präsentiert. Dabei wird auch auf etwaige Verbesserungsvorschläge und bestehende Unzulänglichkeiten der Implementierung eingegangen.

Abstract

This thesis devotes to the subject of bound checking with a theoretical part and with a practical implementation. The intention is, to abet automatic bound checks by the use of hardware extensions. The aim of this work is to improve software quality and reduce program errors in the way that good hardware support of bound checks could help to increase the usage of high level programming languages, which consequently use automatic bound checks.

The introduction shows the scope of applications of these techniques. In a statistical analysis, the ongoing need of research in this subject is pointed out.

A survey of related works shows the current research level.

Based on the SCPU project portable.NET, an implementation of the CLI runtime environment, was ported to the special hardware platform. The CIL interpreter was adapted to use the special instructions for build-in bound checks.

Finally, all findings and results, which also elaborate on possible improvements and current deficiencies, are presented.

Vorwort

An dieser Stelle möchte ich allen, an diesem Projekt beteiligten Personen für die ausgesprochen konstruktive Zusammenarbeit danken. Die Idee zu diesem Forschungsprojekt stammt von Ao.Univ.-Prof., Dipl.-Ing. Dr.techn. Eugen Brenner, der als Projektkoordinator insbesondere in entscheidenden Situationen das Projekt in die richtige Richtung gelenkt hat. Als technischer Projektleiter hat Dipl.-Ing. (FH) Dipl.-Ing. Michael Georg Grasser durch fundierte Fachkenntnis und konsequente Motivation das Projekt maßgeblich vorangetrieben. Als Teil des Hardware-Entwicklungsteams haben Georg Hofer, Johannes Pribsch und Thomas Hodanek mit SCPU die Basis geschaffen, um SILC überhaupt erst zu ermöglichen. Besonderer Dank gebührt auch M.Sc. Ph.D. Roderick Paul Bloem, der mir mit seiner Kenntnis im Compilerbau stets mit Rat und Tat zur Seite stand.

Im Zuge des Projektes unterstützte das Entwicklungsteam um Jiri Gaissler, allen voran Daniel Hellstrom, durch schnelle, kompetente Hilfe in Form von Forenbeiträgen und Quellcodepatches den Projektfortschritt.

Auch auf Seite des Open Source Projektes portable.NET haben die Entwickler mit Geduld und Kompetenz alle anstehenden Fragen beantwortet und ausgesprochen schnell Fehlerbehebungen in ihrem Projekt durchgeführt. Besonders hervorzuheben sind in diesem Zusammenhang Klaus Treichel und die unter dem Pseudonym auftretende Entwicklerin Gopal V.

Nicht zuletzt haben mich natürlich auch meine Familie, meine Freunde und vor allem meine Partnerin Mag. Katharina Schwarz durch Gespräche und Ratschläge gerade in schwierigen Projektphasen unterstützt und meine Motivation am Leben erhalten.

Für meinen auf tragische Weise
verunglückten Freund Stefnir

Inhaltsverzeichnis

1 Einleitung

1.1 Aufbau der Arbeit

Diese Arbeit gliedert sich thematisch in drei Teile. In einer theoretischen Aufarbeitung wird die zugrunde liegende Fragestellung erörtert und präzisiert und vergleichbare Arbeiten zusammengefasst. Die praktische Umsetzung im Zuge dieser Arbeit wird im Anschluss behandelt, wobei auf die konzeptionellen Merkmale der Plattform eingegangen wird und die konkrete Implementierung dokumentiert wird. Abschließend werden Resultate diskutiert und Potential für aufbauende Arbeiten aufgezeigt.

1.2 Ziel der Arbeit

Diese Arbeit widmet sich in einer theoretischen Aufarbeitung und mit einer praktischen Implementierung dem Thema Wertebereichsüberprüfung. Unter dem Begriff Wertebereichsüberprüfung ist in diesem Zusammenhang eine Überprüfung eines Wertes auf die Einhaltung wohldefinierter Grenzen zu verstehen. Verlässt der Wert den gültigen Wertebereich, ist im Allgemeinen das Softwarekonstrukt in einem unerwünschten Zustand. Bestenfalls erfolgt danach eine Ausnahmebehandlung, um das System in einen gewünschten, erlaubten Zustand zu versetzen. In englischer Fachliteratur wird von „*bound checking*“ gesprochen.

Die Überprüfung von Wertebereichen ist in vielen typische Anwendungen enthalten. Beispielhaft sei hier ein Speicherzugriff auf einen mehrdimensionalen Datentyp (Feld, engl. Array) genannt. Um einen fehlerfreien Zugriff in jedem Fall zu gewähren, muß entweder durch entsprechende Überlegungen und Überprüfungen dafür Sorge getragen werden, dass der Zugriff niemals auf Indizes außerhalb der Grenzen des Feldes erfolgen kann, oder die verwendete Softwareplattform bietet entsprechende Mechanismen an, die diese Überprüfungen automatisch

erledigt. Im ersten Fall hat man zumindest die Entscheidung zu verantworten, ob eine Überprüfung von Nöten ist, denn nicht in jedem Fall ist eine solche notwendig. Sind die Grenzen des Feldes und die möglichen Zugriffsindizes zum Zeitpunkt des Entwickelns und damit zum Zeitpunkt des Compilierens bekannt, kann unter Umständen ein fehlerhafter Zugriff gänzlich ausgeschlossen werden. Damit könnte eine Überprüfung des Zugriffsindex gänzlich entfallen. Kann dies nicht ausgeschlossen werden, ist durch programmtechnische Mittel eine Überprüfung vorzunehmen. Da diese händischen Implementierungen naturgemäß fehlerbehaftet sein können, empfiehlt es sich Programmiersprachen einzusetzen, die derlei Überprüfungen automatisch im Hintergrund durchführen. Java ist sicherlich eine der bekannteren Vertreter dieser Sprachgruppe. Aber auch Programmiersprachen wie C++ bieten über entsprechende Bibliotheken wie die STL[1] die Möglichkeit, durch den Einsatz von geeigneten Containerabstraktionen Feldzugriffe bequem abzusichern.

Ziel des praktischen Teils der Arbeit ist, compilergenerierte, sicherheitsrelevante Überprüfungen im Zuge eines Programmablaufes durch Ausnutzung spezieller Prozessorinstruktionen einer modifizierten SPARC V8 Architektur auf die Hardwareebene zu verlagern. Konkret wurde die Laufzeitumgebung portable.NET für die Common Intermediate Language (CIL) in die Linuxdistribution SnapGear eingebunden und anschließend so modifiziert, dass jegliche Zugriffe auf Arrayelemente durch Überprüfung der Zugriffsadresse mithilfe spezieller Mikroprozessorinstruktionen abgesichert werden. Der Rechercheteil dieser Arbeit zeigt aktuelle Implementationsvarianten unterschiedlicher Ausprägung ähnlicher Konzepte sowie eine motivierende Auswertung von Sicherheitsstatistiken, die eine Bearbeitung dieser Thematik, trotz einer Vielzahl von Arbeiten und Implementierungen rund um dieses Thema rechtfertigt.

1 STL ... **S**tandard **T**emplate **L**ibrary

1.3 Sicherheit in Hardware- und Softwarekomponenten

Eine allgemein gültige Definition von Sicherheit in Hard- und Softwaresystemen lässt sich nur schwer finden. Das Verständnis von Sicherheit in diesem Kontext differenziert stark je nach Sichtweise. Auch einer Klassifizierung von Programmiersprachen in "sichere" und "unsichere" sollte man mit Vorsicht begegnen. Die Praxis zeigt, dass unsichere Software mit jeder denkbaren Programmiersprache umgesetzt werden kann. Ungeachtet dessen steht es aber außer Zweifel, dass Programmiersprachen wie C, allen voran durch die gegebene größtmögliche Flexibilität, größeres Fehlerpotential bergen als streng konservative Sprachen, die wie im Fall von Java überdies auch noch in einer virtuellen Maschine verarbeitet werden.

Die Anforderung, dass mit Mitteln der Programmiersprache direkte Hardwarezugriffe, beispielsweise auf den Speicher oder die exekutierende Prozessoreinheit, möglich sind, ist zweifelsohne nach wie vor gegeben. Man denke nur an Digitale Signal Prozessoren (DSP), welche nach wie vor gerne in C, bei gegebenem Anlass sogar direkt in Assembler programmiert werden. Eine Rechtfertigung für derartiges Vorgehen findet sich sehr oft, wenn Geschwindigkeitsaspekte im Vordergrund stehen. Gerade in der digitalen Bildverarbeitung als Teilgebiet der Signalverarbeitung ist die Abarbeitungsgeschwindigkeit oftmals ein Faktor für die Funktionalität der Anwendung. Der Erfolg eines Objektverfolgers hängt beispielsweise in vielen Fällen von der zeitnahen Abarbeitung eines einzelnen Teilbildes ab. Dadurch, dass man benötigte Echtzeitanforderungen einhalten kann, können Objekte einfacher über eine Bildsequenz verfolgt werden. Die Objektbewegung zwischen zwei Messpunkten (Bildern) ist naturgemäß kleiner wenn die aufeinander folgenden Bilder in kurzem zeitlichen Abstand genommen werden können. Dadurch lassen sich einfacher – auch automatisch – Beziehungen zwischen detektierten Objekten knüpfen, und bekannte Objekte leichter verfolgen. Geschwindigkeit ist also in diesem konkreten Beispiel ein entscheidender Faktor.

Das schnellste System ist allerdings nur bedingt tauglich, wenn Programmierfehler zu sicherheitsrelevanten oder auch nur ergebnisrelevanten Beeinträchtigungen führen. Nicht ohne Grund ist man in der modernen Softwareentwicklung durchwegs bestrebt, die höchstmögliche Abstraktion zum Ausdrücken der Programmfunktionalität zu wählen. Zur Zeit hat vollständig automatisch generierter Code aus Modellen, wie es beispielsweise mit der *„Vienna Development Method“* (VDM) und der zugehörigen Entwicklungsumgebung VDM++ möglich ist, allerdings noch kaum Verbreitung. In der Praxis ist oft die höchste Stufe der Abstraktion eine Abbildung durch UML-Diagramme oder ähnlichem. Dabei muss allerdings immer noch ein Großteil des eigentlichen Programmcodes händisch ausformuliert werden und ist trotz sorgfältigem Vorgehen, unterstützt durch moderne Softwareentwicklungsmethoden, grundsätzlich fehleranfällig.

Ein Versuch, die Fehleranfälligkeit von Software zu reduzieren, ist sicherlich die Wahl einer Hochsprache wie Java oder C#. Beide Vertreter behaupten, einen geeigneten Mix aus Flexibilität und Einschränkung zu bieten. So können beide Sprachen weder mit Mehrfachvererbung noch mit direkter Pointermanipulation aufwarten. Eigenschaften, die gemeinhin als Fehlerträchtig angesehen werden. Gleichzeitig soll strikte Typsicherheit helfen, eine stabile Schnittstelle zu schaffen. Gerade diese Einschränkungen sind allerdings auch oftmals Grund von Kritik. Eine oft geführte Argumentation ist, dass nicht die Sprachmerkmale an einem Softwarefehler schuld haben, sondern der falsche Umgang mit diesen. Dem gegenüber steht, dass dies zwar richtig sei, allerdings können trotz sorgfältiger Programmierung naturgemäß Fehler unterlaufen, die gerade in Zusammenhang mit den genannten Sprachmerkmalen schwerwiegende Auswirkungen nach sich ziehen könnten.

1.4 Über alle Grenzen

In diesem Kapitel wird auf eine spezielle Klasse von Programmierfehlern eingegangen, die historisch betrachtet oftmals zu schwerwiegenden Funktionsstörungen, und auch Sicherheitsbeeinträchtigungen geführt hat. Im

Zentrum dieser Fehlerklasse steht eine unerwünschte, unzulässige Wertzuweisung eines Speicherbereichs in der Art, dass diesem Speicherbereich ein Wert zugewiesen wird, der außerhalb des vorgesehenen Wertebereichs liegt.

Dieser Fehler kann grundsätzlich mit allen Werttypen gemacht werden. Fakt ist, dass durch die beschränkte Registerbreite jeder Hardware, jeder Datentyp ein absolutes Maximum bzw. Minimum hat. Wird diese Grenze beispielsweise durch Addition überschritten, kommt es zu einem Überlauf. Je nach Interpretation des Werttyps führt dies zu einem falschen Ergebnis.

Im Folgenden werden die drei offensichtlichsten Typen derartiger Wertebereichsüberschreitungen besprochen:

1.4.1 primitive Datentypen

Auch wenn der konkrete Name in manchen Programmiersprachen anders lautet, versteht man unter primitiven Datentypen meist jene, die durch die Hardwarebeschaffenheiten direkt unterstützt werden. Dazu zählen Ganzzahlenrepräsentation mit und ohne Vorzeichen, Fließ- und Festkommawerte. Der maximale und minimale ausdrückbare absolute Wert hängt von der Anzahl der Bits ab die zur Repräsentation verwendet werden. Bei Sprachen wie C oder C++ hängt dies bei ein und demselben Datentyp von der darunterliegenden Hardware ab, bei Sprachen wie Java wird dies durch die dazwischen liegende virtuelle Hardware vorgegeben und ist somit von der CPU unabhängig. Gemein bleibt diesen Varianten jedoch, dass das absolute Maximum bzw. Minimum durch Operationen wie Addition und Subtraktion über- bzw. unterschritten werden können. Dies führt in jedem Fall zu einem falschen Ergebnis sofern dieser Umstand nicht erkannt und entsprechend behandelt wird. Bei Ganzzahlwerten spricht man dabei zum Beispiel von einem *„Integer Overflow“* bzw. *„Arithmetic Overflow“*. Viele CPUs erkennen derartige Überläufe und signalisieren dies durch setzen entsprechender Flags, die x86 Architektur sieht dafür beispielsweise die beiden Flags *„carry flag“* und *„overflow flag“* vor. Eine Nutzung dieser Information ist allerdings

nur in Sprachen wie C oder C++ möglich, die einen direkten Zugriff auf diese CPU-Flags gestatten.

Interessant erscheint in diesem Zusammenhang, dass viele populäre Programmiersprachen, darunter auch Java, keine Möglichkeit vorsehen, mit eigenen Sprachmitteln auf diese Ereignisse reagieren zu können. Die Behandlung dieser Ereignisse muss hier durch entsprechende arithmetische Überprüfungen vor der eigentlichen Operation sicher gestellt werden, dass es zu keinem fehlerhaften Ergebnis kommen kann.

Da in diesem Zusammenhang oftmals Fehler gemacht werden, die auf mangelndes Verständnis dieses Problems schließen lassen, wird an dieser Stelle eine beispielhafte Implementation eines solchen Checks für eine Addition gezeigt, zuerst eine falsche Implementierung, danach eine korrekte Überprüfung (Quellen in Pseudosyntax):

```
if( (SUMMAND1 + SUMMAND2) > Integer.MAX_VALUE)
   throw new IntegerOverflowException();
```

Fehlerhafte Überprüfung auf Überlauf

Der Ausdruck wird niemals Wahr ergeben, größer als Integer.MAX_VALUE kann ein Wert niemals werden (bei statischer Typisierung).

```
if( (Integer.MAX_VALUE – SUMMAND1) < SUMMAND2)
   throw new IntegerOverflowException();
```

Bessere Überprüfung auf Überlauf

Obwohl bereits ältere Vertreter von Programmiersprachen wie Lisp und ADA spracheigene Mittel zur Erkennung und Behandlung von Überläufen bereitstellen, hat diese durchaus sinnvolle Spracheigenschaft interessanterweise nicht Einzug in die „sichere“ Programmiersprache Java gehalten – es gibt schlicht keine `OverflowException()`, die im Fehlerfall automatisch ausgelöst wird. Im Gegensatz dazu bietet .NET ein solches Konstrukt an. Eine händische Überprüfung kann entfallen:

```
// statements_checked.cs
...
class OverFlowTest {
   static short x = 32767;   // Max short value
   static short y = 32767;
    // Using a checked expression
   public static int myMethodCh() {
      short z = 0;
      try {
        z =  checked((short)(x + y));
      }
      catch (System.OverflowException e) {
        System.Console.WriteLine(e.ToString());
      }
      return z;   // Throws the exception OverflowException
   }
   public static void Main() {
     ...
   }
}
```

OverflowException in C#, Quellcode aus msdn

1.4.2 Teilbereichstypen

In wenigen Programmiersprachen findet man Umsetzungen des Konzeptes der Teilbereichstypen (engl.: subrange types), dazu zählen unter anderen Lisp, ADA, Fortran, Modula2 und Pascal. Teilbereichstypen werden meist gemeinsam mit Datentypen wie integer, float u.Ä. zu den primitiven Datentypen gezählt. Sie ermöglichen die Einschränkung des Wertebereichs eines Datentyps. Eine Überprüfung von Variablen solcher Datentypen auf Einhaltung des Wertebereichs erfolgen dabei automatisch – also ohne programmtechnische Vorkehrungen - spätestens zur Laufzeit, in einigen Fällen jedoch schon zum Zeitpunkt des Compilierens. Im Gegensatz zu einem arithmetischen Überlauf, kann die CPU eine Verletzung der Wertgrenzen bei Teilbereichstypen nicht erkennen. Eine Überprüfung muss vom Compiler an relevanten Stellen eingefügt werden.

Der Einsatzbereich von Teilbereichstypen ist vorwiegend im Domänenmodell einer Anwendung zu suchen. Eine technische Notwendigkeit ergibt sich kaum. So ist beispielsweise in einem physikalischen Dynamikmodell die

Geschwindigkeit eines Objektes jedenfalls mit der Lichtgeschwindigkeit begrenzt (Physiker mögen dieses Beispiel verzeihen, spielen doch schon bei weit geringeren Geschwindigkeiten viele weitere Aspekte mit). Ein eigener Datentyp für die Objektgeschwindigkeit könnte somit auf Werte innerhalb der Lichtgeschwindigkeit beschränkt werden. Ähnlich dazu könnte auch die Anzahl der Personen in einem PKW zwischen Null und Fünf eingeschränkt sein.

Da, wie gesagt, die Verwendung dieses Datentyps einer entsprechenden Anwendung bedarf, findet er sich in wenigen Programmiersprachen und stellt somit eine Ausnahme dar.

Wie man erkennt, kann es auch bei Teilbereichstypen zu einer Wertebereichsverletzung kommen, was zu ähnlichen Methoden zur Überprüfung führt, wie sie auch bei primitiven Datentypen vorgestellt wurden.

Eine weitere Anwendung dieser Technik findet sich bei mehrdimensionalen Datentypen.

1.4.3 mehrdimensionale Datentypen

Mehrdimensionale Datentypen sind beispielsweise Felder aus primitiven Datentypen. Ein digitales Bild wird im Allgemeinen durch eine Menge von Werten (Pixel) die in zweidimensionaler Orientierung angeordnet sind, repräsentiert. Die Werte an sich sind z.B. repräsentativ für den Grauwert im Bereich von 0 bis 255 (siehe auch Teilbereichstypen). Eine Zeichenkette (String) ist ebenfalls ein mehrdimensionaler Datentyp. Jedes Zeichen wird, durch einen Wert codiert, hintereinander in den Speicher geschrieben.

Da der Speicher in Computern üblicherweise linear angeordnet ist, werden die einzelnen Ebenen mehrdimensionale Datentypen hintereinander in den Speicher geschrieben, bei einem Bild also z.B. Zeile für Zeile. Die Gefahr besteht nun darin, dass über einen ungültigen Zugriffsindex der Speicherbereich verlassen wird. In diesem Fall spricht man von einem *„Buffer*

overflow“. Je nachdem wo dieser Speicher angelegt ist, unterscheidet man überdies zwischen „*Stack overflow*“ und „*Heap overflow*“. Beiden Arten ist gemein, dass sie für missbräuchliche Verwendung gebraucht werden können. Ein guter Teil der existierenden Schadprogramme nutzt derartige Fehler um Speicherbereiche mit eigenem Schadcode zu überschreiben und diesen in Folge zur Ausführung zu bringen oder auch nur um Speicherbereiche auszulesen. Dies kann unter Umständen zur Kompromittierung ansonst verschlüsselter Passwörter missbraucht werden.

Um diese zu vermeiden, muss der Zugriffsindex auf die Einhaltung seiner Grenzen überprüft werden. Wie auch bei den zuvor behandelten Datentypen kann dies manuell durch Programmcode oder automatisch durch den Compiler geschehen. Im letzten Fall spricht man von automatischem „*bound checking*“.

Erfreulicherweise unterstützen viele Programmiersprachen diese Automatismen schon von Haus aus. In Java und .NET bieten die mitgelieferten Bibliotheken eine Vielzahl von so genannten Containerklassen, die einen effizienten, sicheren Umgang mit mehrdimensionalen Daten erlauben. Aber auch in C++ kann, durch die mittlerweile fast zur Sprache zugehörig empfundene Bibliothek STL, ein sicherer Umgang gewährleistet werden.

1.4.4 Zusammenfassung

Diesen drei in diesem Kapitel besprochenen Aspekten gemein ist, dass in allen Fällen ein Wert auf Einhaltung von oberer wie unterer Grenze überprüft werden sollte. Die notwendigen arithmetischen Operationen gleichen sich in allen Fällen.

Es liegt also nahe, all diese Fälle durch eine einzige Vorkehrung abzudecken. Genau diese Idee ist Grundlage der Forschung um SCPU und damit die fundamentale Basis für SILC.

SCPU will der CPU diese Wertebereichsüberprüfung in der Art beibringen, dass bei jeder Operation die beteiligten Operanden auf Einhaltung ihrer Grenzen überprüft werden. SILC befasst sich mit der Nutzung dieser Erweiterung.

1.5 Statistische Fakten

Im Folgenden wurden Softwarefehler mithilfe öffentlich zugänglicher Fehlerdatenbanken auf deren Auswirkung untersucht. Intention dieser Datenbanken ist oftmals Fehler, die zu sicherheitsrelevanten Funktionsbeeinträchtigungen führen, zu dokumentieren. Die Fehlerklassifizierung erfolgt dabei in unterschiedlichster Weise, fast immer wird die Gewichtigkeit des Fehlers – wie z.B.: niedrig, mittel, hoch – angegeben. Schon seltener findet man Metainformationen über die eingesetzte Plattform, die Programmiersprache oder den Zeitpunkt, an dem ein Fehler gefunden wird. Einen Fehlertyp im Sinne des nächstliegenden Grundes wie z.B. mangelnde Eingabedatenüberprüfung, Konfigurationsfehler oder mangelhafte Zeigerverwendung sucht man in den meisten Datenbanken vergeblich. Oft finden sich detailliertere Informationen nur als Bemerkung und sind damit schwer auswertbar. Auswertungen derartiger Datenbestände sind deshalb nur begrenzt tauglich um daraus Schlüsse und eventuell Erkenntnisse zu ziehen.

Gegenüberstellungen ähnlicher Softwarekomponenten aufgrund von Faktoren wie Fehleranzahl pro Zeitraum und Fehlergrad haben vielleicht informativen Charakter, der nebenbei viele Meinungen und Vorurteile nährt. Daraus praktischen Nutzern für die Softwareentwicklung abzuleiten ist aber leider schwer denkbar. Es soll dies nun kein weiterer Versuch sein, Software aufgrund der Plattform, der Programmiersprache oder gar anhand des Betriebssystems betreffend sicherheitsrelevanter Merkmale zu bewerten. Eine Klassifikation und Auswertung soll lediglich anhand des Fehlergrundes erfolgen.

Datenbasis der Untersuchung ist die frei zugängliche Fehlerdatenbank „National Vulnerability Database“[2,3] vom „National Institute of Standards and Technology“ (NIST). Der Datenbestand beläuft sich zum Zeitpunkt der Recherche auf ca. 26.000 Einträge, wobei im Schnitt 18 Einträge pro Tag hinzukommen. An dieser Stelle sei allerdings darauf hingewiesen, dass im Zuge der Arbeit über Monate hinweg mehrmals auf diese Daten zugegriffen wurde. Dabei fiel auf, dass sich die Daten vergangener Jahre änderten. So war z.B. die Anzahl der gemeldeten Fehler aus vergangenen Jahren beträchtlichen Änderungen unterworfen. Ob im Nachhinein Einträge verändert bzw. hinzugefügt wurden, oder dies auf einen Softwarefehler zurückzuführen ist, konnte nicht in Erfahrung gebracht werden.

Die Bewertung der Einträge nach potentieller Auswirkung erfolgt in dieser Datenbank nach der zehnstufigen Skala *„Common Vulnerability Scoring System“*[4] (CVSS-SIG). Werte zwischen null und drei werden als *„low“* klassifiziert, von vier bis sechs als *„medium“*, Werte zwischen sieben und zehn führen zu einer Klassifikation als *„high“*.

Die Datenbank unterscheidet zehn verschiedene Fehlertypen:

1. input validation error
    2. boundary condition error
    3. buffer overflow
4. access validation error
5. exceptional condition error
6. environmental error
7. configuration error
8. race condition
9. design error
10. other error

2 http://nvd.nist.gov/nvd.cfm?advancedsearch
3 http://nvd.nist.gov/statistics.cfm
4 http://www.first.org/cvss/

Folgende Grafik zeigt im zeitlichen Verlauf die Anzahl der Fehlermeldungen:

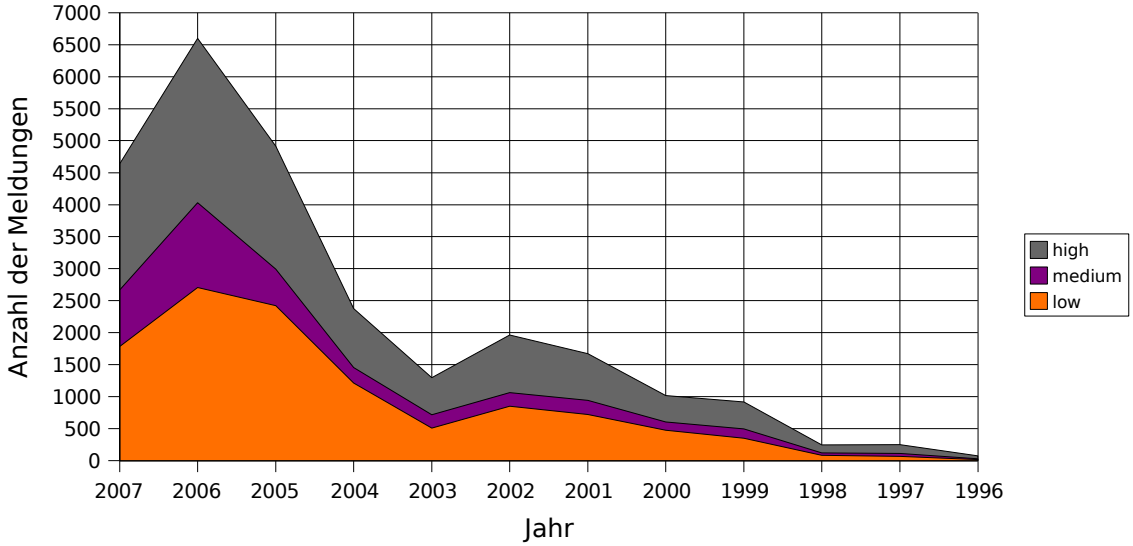

Abbildung 1-1: Gesamtzahl der Meldungen

Daraus ist ersichtlich, dass die Zahl der Meldungen beinahe stetig steigt, wobei tendenziell mehr Meldungen als „*low*“ klassifiziert werden. Natürlich kann aus dieser Auswertung nicht der Schluss gezogen werden, Software wäre im Jahr 1996 statistisch gesehen besser gewesen. Vielmehr ist die steigende Zahl der Meldungen auf erhöhte Sensibilisierung und Engagement zurückzuführen. Früher wurden schlicht weit nicht so viele Fehler gemeldet.

Folgende Auswertung zeigt den relativen Anteil der einzelnen Fehlertypen bezogen auf die Gesamtzahl in jedem Jahr:

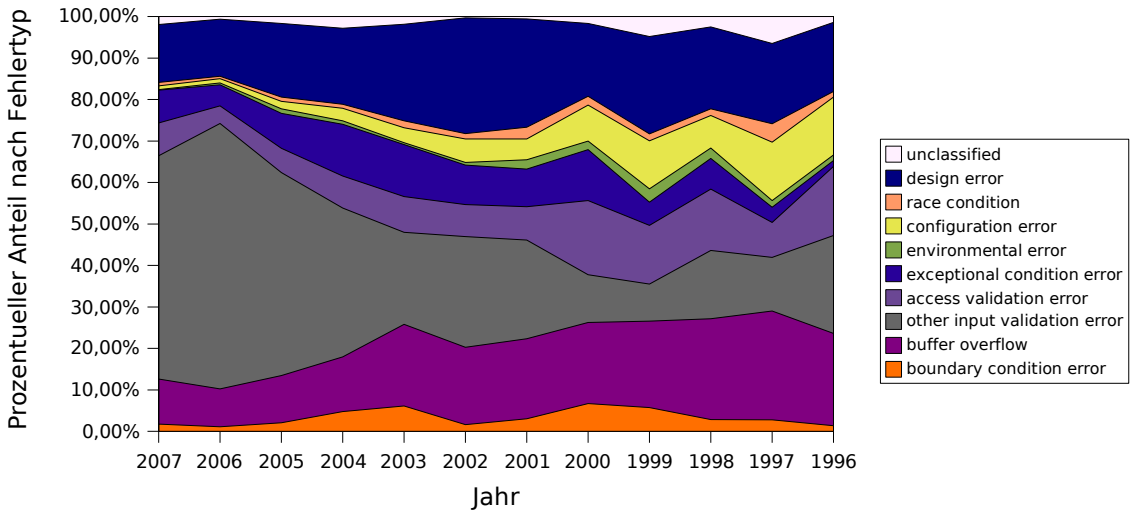

Abbildung 1-2: Prozentuelle Verteilung der Fehlertypen

Bemerkenswert ist, dass der relative Anteil an Fehlern, klassifiziert als „input validation error“ - wozu auch „buffer overflow“ und „boundary condition error“ zählen - im Betrachtungszeitraum tendenziell größer werden.

Die in dieser Arbeit behandelte Thematik, betrifft in erster Linie die Fehlertypen *„buffer overflow“* aber auch *„boundary condition error“*. Im Folgenden wird deshalb näher auf diese Fehlerklassen eingegangen.

Betrachtet man diese Fehlerklassen in Bezug auf die Klassifikation nach Fehlergewicht, ergeben sich die Zusammenhänge dargestellt in Abbildung 1-3 und Abbildung 1-4. Dabei fällt auf, dass insbesondere bei „buffer overflow“ der Anteil an Klassifizierungen mit „high“ und „medium“ unverändert hoch bei ca. 80% liegt.

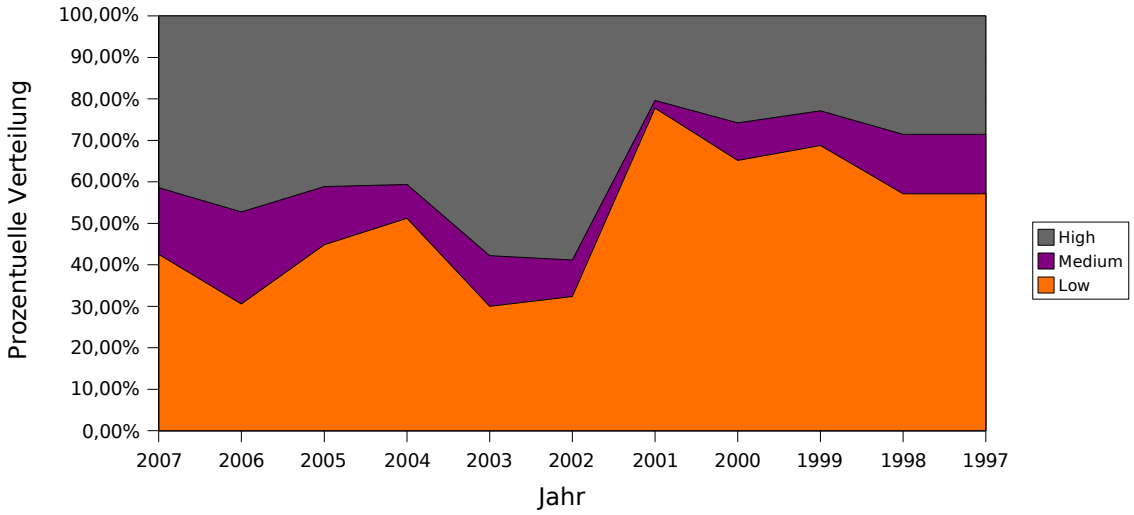

Abbildung 1-3: Verteilung boundary condition error

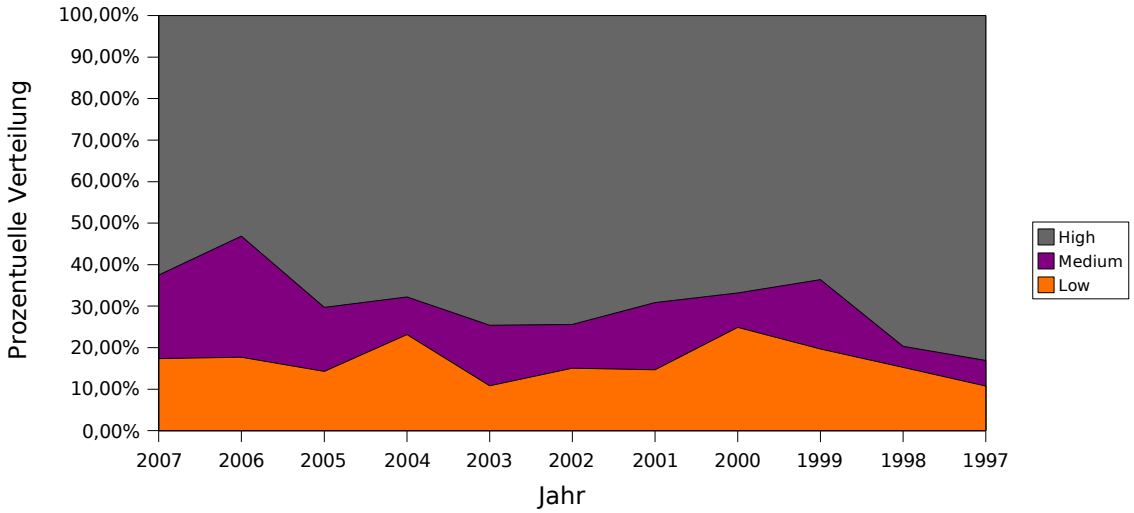

Abbildung 1-4: Verteilung buffer overflow

1.5.1 Zusammenfassung

Zusammengefasst kann also gesagt werden, das zwar der prozentuelle Anteil an „buffer overflows“ und „boundary condition errors“ gemessen an allen Fehlern in den letzten Jahren bei konstanten, relativ geringen 10% liegt. Allerdings ist die Schwere dieser Fehlertypen im Vergleich zu anderen Klassen mit beinahe 80% verhältnismäßig oft mit „high“ und „medium“ klassifiziert.

Was bedeutet dies nun für die vorliegende Arbeit?

Entgegen vieler Angaben in Literatur und vergleichbaren Arbeiten, ist der Anteil an Softwarefehlern durch Wertebereichsverletzungen relativ gering (ca. 10%). Dieser Anteil wird oftmals viel zu hoch angegeben, zumindest konnten manche Aussagen in dieser Richtung nicht durch Quellen und Daten verifiziert werden. Die Arbeiten [Shao et al., 2005] und [Lam, Chiueh, 2005] rechtfertigen ihre Bemühungen rund um die Thematik *„bound checking“* beispielsweise durch übertriebene Werte. So wird behauptet beinahe 50% aller Fehler wären auf *„buffer overflows“* zurückzuführen. Diese Übertreibungen sind einer sachlichen Herangehensweise allerdings nicht dienlich. Insbesondere da diese Übertreibung auch gar nicht notwendig wäre um ein Engagement in der Richtung zu rechtfertigen, denn:

Wie die vorangegangene Analyse vorhandener Datenbestände über Softwarefehler gezeigt hat, ist der Anteil an „buffer overflows“ gemessen an allen Fehlern relativ gering. Allerdings wird die Auswirkung ausgesprochen vieler „buffer overflows“ mit hoch bis mittel angegeben. Dies bedeutet, dass mit einem Mechanismus, der „buffer overflows“ (oder vergleichbare Fehler) verhindert, die Welt der Softwareentwicklung zwar nicht schlagartig fehlerfrei wird, allerdings könnte mit einem Schlag eine ausgesprochen schwerwiegende Fehlerklasse eliminiert werden.

2 Vergleichbare Arbeiten

Viele Hochsprachen überprüfen automatisch vor einem schreibenden oder lesenden Arrayzugriff, ob das jeweilige Element innerhalb der Arraygrenzen liegt. Diese impliziten – also nicht vom Programmierer explizit vorgesehenen – Überprüfungen schlagen sich zum Teil mit erheblichen Laufzeiteinbußen zu Buche. Besonders in zeitkritischen Applikationen, die extensiv Zugriffe auf mehrdimensionale Datenfelder nutzen, kann dieser Umstand die Verwendung von Programmiersprachen und Laufzeitumgebungen mit diesen Eigenschaften verhindern bzw. erschweren. Als plakatives Beispiel seien Bildverarbeitungsalgorithmen bei der Objektverfolgung genannt, welche harte Echtzeitbedingungen bei hohen Frameraten erfüllen müssen. Die Anzahl der Feldzugriffe bei einer typischen Filteroperation verhält sich zumindest linear zum Produkt aus Filterkerngröße und Bildgröße. Xi und Xia sprechen gar von bis zu 50% der Laufzeit einer Matrixmultiplikation, die rein auf die notwendigen Bound Checks zurückzuführen ist. [Xi, Xia, 1999]

Um diese Laufzeiteinbußen zu reduzieren und damit automatische „*bound Checks*“ auch für zugriffsintensive Anwendungen interessant zu machen, wurden viele Versuche unternommen, um dieses Laufzeitverhalten zu verbessern.

2.1 Komplexitätsreduktion durch Deaktivierung

Der wohl einfachste Ansatz ist, „*bound checks*“ zu deaktivieren. Die relativ neue Programmiersprache D, andererseits auch ältere Vertreter wie Pascal oder Fortran und viele andere mehr, erlauben das Aktivieren bzw. Deaktivieren von „*bound checks*“ über eine Compileroption zum Zeitpunkt des Compilierens. C# erlaubt das Kennzeichnen von unsicheren Bereichen (engl.: „*unsafe region*“) durch eine spezielle Notation, die unter anderem bewirkt, dass in den jeweilig gekennzeichneten Bereichen auf automatische „*bound checks*“ verzichtet wird.

Nachteil dieser Herangehensweise ist offensichtlich, dass die gewonnene Geschwindigkeitssteigerung auf Kosten der Idee von automatischem *„bound checking“* geht – der Stabilisierung und Qualitätsverbesserung von Softwarekomponenten.

In der Praxis geht man oft den Weg, *„bound checks“* während der Implementierungs- und Testphase zu aktivieren um möglichst viele fehlerhafte Zugriffe zu entdecken. Spätestens mit Auslieferung bzw. im Produktivbetrieb wird die Software allerdings durch entsprechende Optionen ohne implizite *„bound checks“* kompiliert, um eine höhere Gesamtperformanz zu erreichen.

Bezogen auf Pufferüberläufe wird der korrekte Programmablauf allerdings wiederum dem Qualitätsbewusstsein und der Aufmerksamkeit während der Codierung überlassen. Eine zweifelhafte Entscheidung, denn durch eingeschaltete *„bound checks“* während Umsetzung und Tests können fehlerhafte Zugriffe, welche durch extrinsische Einflüsse (z.B. Übergabeparameter) hervorgerufen werden, im Produktivbetrieb nicht mit Sicherheit ausgeschlossen werden. In diesen Fällen ist wiederum händische Wertebereichsüberprüfung und eine spezielle Fehlerbehandlung notwendig.

2.2 Komplexitätsreduktion durch Codeoptimierung

Alternative Ansätze versuchen bei Beibehaltung des Qualitätsgedanken die Geschwindigkeitseinbußen zu reduzieren. Die Mehrheit dieser Optimierungen setzen auf Softwareseite an und versuchen durch Eliminierung unkritischer Überprüfungen zum Zeitpunkt des Compilierens die Anzahl der notwendigen Überprüfungen zu reduzieren. Das Verständnis von unkritischen Überprüfungen differenziert allerdings stark.

2.2.1 Bounds Checking with Taint-Based Analysis

[Chuang et al., 2007] richteten ihre Implementierung auf dem Umstand aus, dass Buffer Overflows vor allem in Bezug auf mutwillige Manipulation in Form von „Stack Smaching“ und ähnlichem zu verhindern seien:

> *The optimization is based on the observation that buffer overflow attacks are launched through external inputs. Therefore, it is sufficient to bounds check only the accesses to those data structures that can possibly hold the external inputs. Also, it is sufficient to bounds check only the memory writes.*

So berechtigt diese Annahme auch scheinen mag, befriedigt dieser Ansatz nicht den generellen Anspruch Pufferüberläufe zu verhindern. Aus qualitätssichernder Perspektive erscheint es gleichwertig, ob ein Pufferüberlauf aufgrund des typischen „Off by one“ Programmierfehlers oder aufgrund mangelhafter Eingabedatenüberprüfung passiert. Beide Fälle bergen das Potential das Programmergebnis zu verfälschen, einen Programmabsturz zu provozieren oder gar Raum für zweckentfremdende Manipulationen zu öffnen.

Betrachtet man den Aspekt, dass diese Implementierung darauf abzielt, C und C++ Code mit „*bound checks*“ zu versehen, relativiert sich diese Kritik, da ohne diese Maßnahmen gar keine Überprüfung stattfinden würde. Die gezeigten Ergebnisse und Geschwindigkeitsvergleiche sind folglich immer auf eine Codevariante ohne Überprüfungen bezogen.

Technisch erfordert diese Herangehensweise, die interne Repräsentation von Zeigern um die Grenzen zu ergänzen. In diesem Fall wurde dies durch sog. „*Fat Pointer*“ gelöst. Diese zieht zwar eine Änderung des Zeigerformates mit sich, allerdings ist Zugriffsgeschwindigkeit und Initialisierungsaufwand im Vergleich zu anderen Lösungen geringer.

Für einige Programme wurden Laufzeitanalysen durchgeführt. Werden alle Zugriffe überprüft, erleidet diese Lösung einen Geschwindigkeitsreduktion um 40%. Betrachtet man nur Zugriffe die externen Einflüssen unterliegen,

reduziert sich dieser Mehraufwand auf ca. 29%, werden gar nur Netzwerkzugriffe betrachtet, verbleiben lediglich 6%.

2.2.2 Optimizing Array Bound Checks Using Flow Analysis

Rajiv Gupta beschreibt in [Rajiv Gupta, 1993] und [Rajiv Gupta, 1990] Optimierungsmöglichkeiten bei Array Bound Checks mithilfe von Datenflussanalyse. Dieser Ansatz bringt in der Praxis sehr gute Erfolge weshalb viele Implementierungen bereits derartige Optimierungen umgesetzt haben.

Die Eliminierung von Bound Checks erfolgt dabei mithilfe dreier Algorithmen:

1. Lokale Elimination

   Erfolgen innerhalb eines lokalen Blockes zwei identische Checks kann der nachfolgende Check entfallen sofern dazwischen keine Redefinition der zugrunde liegenden Variablen erfolgt.

2. Globale Elimination

   Ähnlich zur lokalen Elimination kann ein Check entfallen, wenn innerhalb eines Programmes ein identischer Check an einer vorgelagerten Stelle stattfindet, und dazwischen keine Redefinition der zugrunde liegenden Variablen erfolgt. Eine andere globale Eliminationsmöglichkeit besteht, wenn eine zuvor liegende Überprüfung die jeweilige Überprüfung bereits inkludiert.

3. Verschieben der Überprüfungen aus dem Schleifenkörper

   Erfolgt ein Check innerhalb einer Schleife, kann dieser vor den Schleifenkörper gezogen werden, sofern nur Definitionen von außerhalb des Schleifenkörpers herangezogen werden, der betreffende Check also Schleifeninvariant ist.

Die gezeigten Ergebnisse legen vollständiges *„Bound Checking“* zugrunde. Guptas erreicht mit seiner Implementierung abhängig vom Algorithmus der

Anwendung eine Reduktion von Bound Checks zwischen 42% und nahezu 100%.

2.3 Laufzeitreduktion durch spezialisierte Hardware

Ein gänzlich anderer Ansatz ist, *„Bound Checks“* durch spezielle Hardwarekomponenten zu beschleunigen. Ein *„Bound Check“* setzt sich ungeachtet der verwendeten Architektur und Programmiersprache aus zwei arithmetischen Vergleichen zusammen. Die jeweils obere und untere Grenze wird gegen den Index überprüft und das Ergebnis in Folge durch eine bedingte Sprunganweisung ausgewertet. Auf Ebene von Prozessorinstruktionen handelt sich dabei für obere und untere Grenze je nach Instruktion Set ebenfalls um einen Vergleich und einen Sprung. Obgleich die Mehrheit der Prozessorarchitekturen eine eigene Vergleichsoperation bieten, werden diese intern in den meisten Fällen durch eine arithmetische Subtraktion realisiert. CISC Architekturen verpacken Vergleich und Sprung oft auch in einer einzigen Instruktion (vgl.: Jcc Instruktion in [Intel, 1999]).

2.3.1 Intel's BOUND Instruktion

Noch weiter geht Intel mit der x86 Architektur. Mit dem Einzug der Prozessoren 80188 respektive 80186 in die x86 Prozessorfamilie (im PC-Segment 286 und folgende) ist der Maschinenbefehl BOUND Teil des Instruction Sets ([Intel, 1999]):

> *This instruction determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand).*
> *...*
> *If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled.*

Bemerkenswert erscheint jedoch der Umstand, dass eben diese Instruktion weit langsamer ist als die vier äquivalenten Einzelinstruktionen:

```
(1) cmp    reg, LowerBound
(2) jl     OutOfBounds
(3) cmp    reg, UpperBound
(4) jg     OutOfBounds
```

In Zeile 1 und 3 wird ein Wert (z.B.: Index) gegen die untere bzw. obere Grenze verglichen, Zeile 2 und 4 realisieren den Sprung in die Ausnahmebehandlung. (Quellcode aus: [Hyde, 2004])

Hyde quantifiziert diese Laufzeitdifferenz in „*The Art of Assembly Language*“ mit mit bis zu doppelter Zyklenzahl:

> *On the 80486 and Pentium/586 chips, the sequence above only requires four clock cycles assuming you can use the immediate addressing mode and the branches are not taken; the bound instruction requires 7-8 clock cycles under similar circumstances and also assuming the memory operands are in the cache.*

Leider liegt keine Aussage vor, ob dieser nicht ganz einsichtige Umstand in aktuellen Versionen nach wie vor besteht. Der „*Instruction Set Reference*“ [Intel, 2007] für die 64-Bit Varianten „*Itanium*“ ist jedenfalls zu entnehmen, dass die BOUND Instruktion im 64-Bit Modus nicht gültig ist. Lediglich in den 32-Bit Kompatibilitätsmodi steht diese Instruktion zur Verfügung, wobei hierbei zu erwähnen ist, dass der „*Itanium*“ Prozessor kein Mitglied der IA-32[5] Familie ist, sondern eine von Intel neu eingeführte Architektur mit zugehörigem Befehlssatz namens IA-64[6]. IA-64 unterstützt IA-32 auf dem Stand von Pentium III. Diese Unterstützung ist allerdings nur aus Kompatibilitätsgründen als Nebenteil des Prozessors implementiert und erreicht im IA-32 Bit Modus bei weitem nicht die Leistung aktueller IA-32 Varianten wie Core 2 und Core Quad.

All diese Eigenschaften rechtfertigen wohl, dass diese Instruktion offenbar nicht sehr beliebt ist und laut diverser Meldungen kaum Anwendung findet.

5 IA-32 ... Intel Architecture, 32-Bit
6 IA-64 ... Intel Architecture, 64-Bit

2.3.2 Cash

In [Lam, Chiueh 2005] wird ein interessanter Ansatz namens Cash[7] zur Reduktion der Laufzeiteinbußen auf Basis der x86 Architektur vorgestellt. Durch die zweckentfremdende Verwendung der Segmentationseinheit des Prozessors geschehen die eigentlichen Überprüfungen gegen obere und untere Grenze ohne zusätzliche Prozessorinstruktionen. Die Implementierung Cash basiert auf dem *„Bounds Checking GCC“* (BCC), einer GCC Erweiterung für Wertebereichsüberprüfungen in C Programmen.

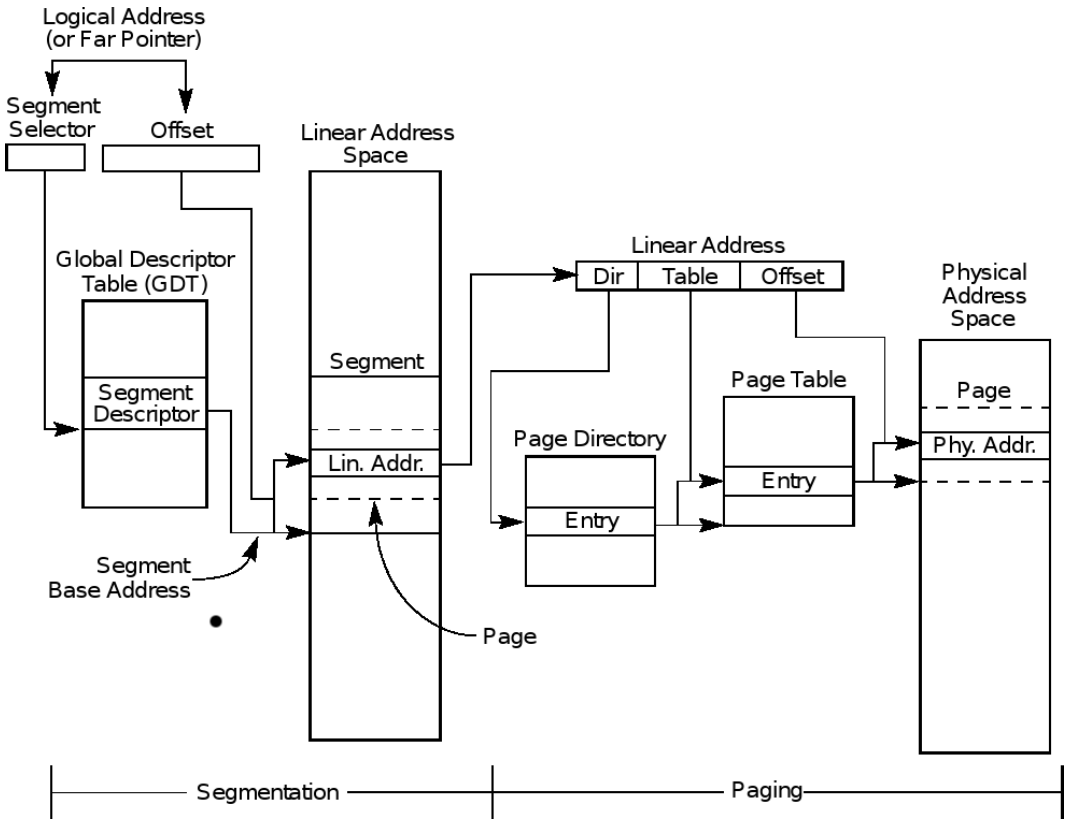

Abbildung 2-1: Speicherverwaltung in der x86 Architektur
Bildquelle: [Intel, 3A, 2007]

Intels x86 Speichermanagement bedient zwei Anforderungen an eine Speicherverwaltung. Zum einen bietet es mittels Segmentation (engl.: *„segmentation“*) einen Mechanismus an, um Code, Daten und Stack einzelner

7 **C**hecking **A**rray Bound Violation Using **S**egmentation **H**ardware

Programme zueinander zu isolieren. *„Paging“* deckt die verbleibende Anforderung ab, Prozessen einen virtuellen Adressraum zur Verfügung zu stellen.

Abbildung 2-1 zeigt das Strukturbild der Speicherverwaltungseinheit von x86 Prozessoren. Eine logische Adresse wird über den *„Segment Selector“* einem Segment innerhalb des *„Linear Address Space“* zugeordnet. Die *„Global Descriptor Table“* (GDT) verwaltet und hält dabei die Startadresse der Segmente innerhalb des *„Linear Address Space“*.

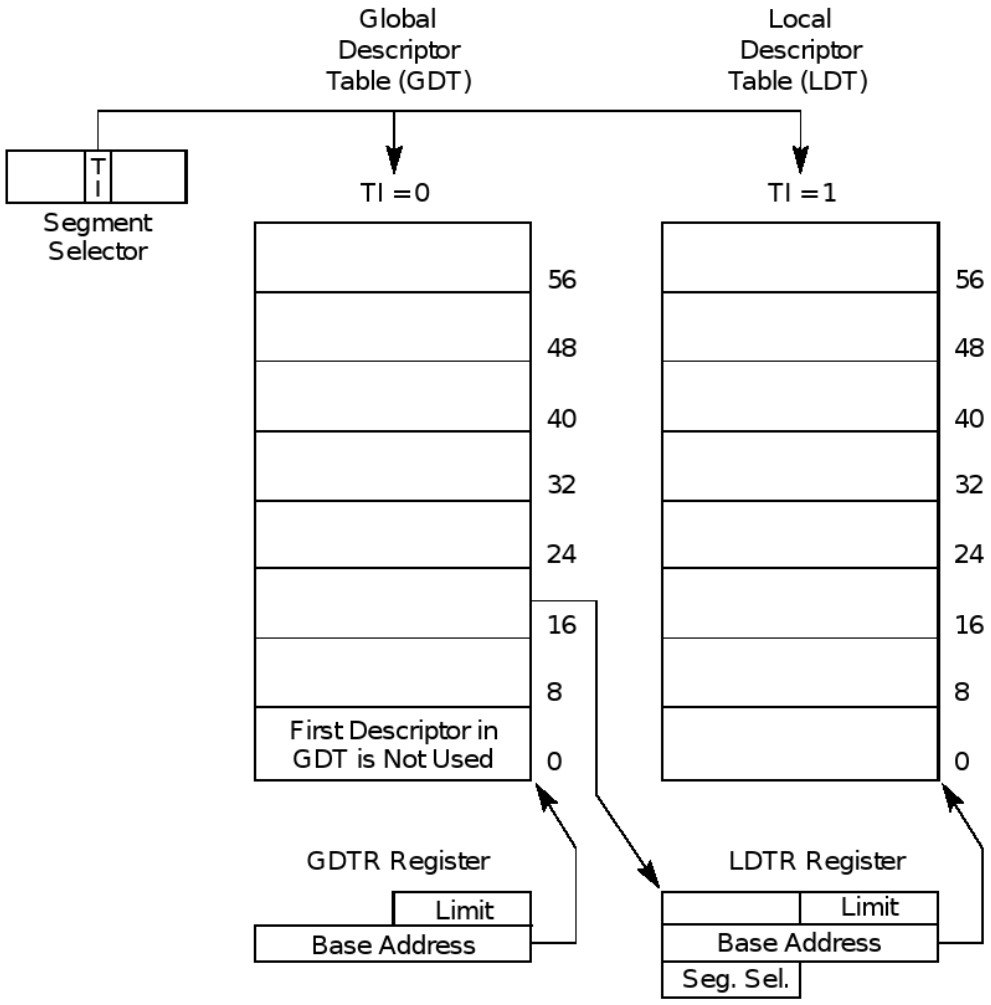

Abbildung 2-2: GDT und LDT in x86 Architekturen
Bildquelle: [Intel, 3A, 2007]

Neben der GDT kennt die Speicherverwaltung noch *„Local Descriptor Tables“* (LDT). Eine *„Descriptor Table“* ist ein Felder variabler Länge von *„Segment*

Descriptors“, hält also Metainformationen zu den Segmenten wie Basisadresse und Länge (vgl.: Abbildung 2-2). Aufgrund der Länge des „Segment Descriptors“ in der logischen Adresse von 13 bit, können bis zu 8192 Segmente in einer „*Descriptor Table*“ verwaltet werden. Jedes System muss zumindest eine GDT haben, kann optional aber mehrere LDTs verwalten. Dies kommt beispielsweise zum Einsatz wenn ein Prozess mehrere Tasks verwaltet und jedem Task ein Segment in einer „*Local Descriptor Table*“ zuordnet. Das Anlegen neuer Segmente erfolgt üblicherweise im Zuge des Programmstarts.

Dieser beschriebene Segmentationsmechanismus wird zum Überprüfen der Grenzen verwendet. Für jedes zu überprüfende Objekt wird ein eigenes Segment angefordert. Bei jedem Zugriff auf dieses Element erfolgt automatisch durch die entsprechende Hardwarestruktur eine Überprüfung auf Gültigkeit des Zugriffsindex. Da, wie zuvor beschrieben, die Anzahl der verwalteten Segmente in einer „*Descriptor Table*“ mit 8192 beschränkt ist, wäre die Anzahl der Pointer bzw. Arrays in einem Programm stark eingeschränkt. Aus diesem Grund werden in dieser Arbeit nur Arrayzugriffe der Form A[i], A++, ++A, A-- und –A innerhalb einer Schleife beachtet.

Vorteil dieser Herangehensweise ist sicherlich, dass für den eigentlichen Zugriff keine zusätzlichen Instruktionen bzw. Laufzeiteinbußen anfallen. Die einzigen zusätzlichen Schritte fallen im Zuge der Speicherallozierung in Form der Segmentzuweisung an. Entscheidender Nachteil, wie auch bei vielen Arbeiten rund um das Thema, ist, dass wiederum weit nicht alle Zugriffe abgesichert werden. In diesem Fall ist dieser Umstand jedoch weniger störend. Da Cash auf BCC basiert, werden Zugriffe, welche nicht durch Cash abgedeckt werden, durch BCC geprüft. Trotzdem erscheint diese Arbeit sehr innovativ. Setzt man den Gedanken fort, mündet dies u.U. in ergänzenden Hardwarekomponenten ähnlich der Segmentationseinheit in x86 Architekturen speziell für die Wertebereichsüberprüfung.

Auch die Resultate können sich sehen lassen: Verschiedenste Algorithmen wie die „*singular value decomposition*“ (SVD) wurden in Bezug auf Laufzeit und Codegröße mit der reinen GCC-Variante und mit der BCC compilierten

Version verglichen. Dabei schneidet Cash immer besser ab als BCC, wobei die Laufzeiteinbußen bei den getesteten Programmen zwischen 4,6% und 15,8% liegen, während sich BCC mit immerhin 40% bis 238% zu Buche schlägt.

2.3.3 Parallele Abarbeitung

Ein gänzlich anderen Ansatz wird in der Arbeit *„Low-cost, Concurrent Checking of Pointer and Array Access in C Programs“* bestritten (siehe [Patil, Fischer, 1997]). Die vorgestellte Technik namens *„guarding“* zielt darauf ab, illegale Pointer und Array Zugriffe in C Programmen zu unterbinden. Dazu werden zusätzliche Codefragmente in das Programm eingefügt, d.h. auch dieser Ansatz arbeitet auf Quellcodeebene.

Grundidee hinter dieser Arbeit ist, die notwendigen Instruktionen zur Wertebereichsüberprüfung vom eigentlichen Programmcode zu trennen. Fällt im Zuge des Programmablaufes eine Überprüfung an, wird diese nicht im selben Prozess abgearbeitet, sondern der Auftrag an einen abgespalteten Teilprozess übergeben, der die geforderten *„Bound Checks“* im Hintergrund durchführt. Diese Prozesse werden darum auch *„shadow process“* genannt. In Anbetracht dessen, dass Aufgrund des geringen Preises von CPUs vermehrt Multiprozessor- bzw. Multicoresysteme zum Einsatz kommen, erscheint diese Herangehensweise ausgesprochen zukunftsreich.

Die Geschwindigkeitsanalysen zeigen, dass dieser Ansatz durchaus konkurrenzfähig ist. Der Hauptprozess – und damit jener, der vom Benutzer wahrgenommen wird - erfährt in der Regel eine Geschwindigkeitseinbuße von <1% bis 10%. Klarerweise spielen bei dieser Implementierung auch andere Aspekte, wie Kommunikationsaufwand und Speicherbedarf des „shadow process“ eine Rolle. Da die übrigen vorgestellten Arbeiten jedoch zumeist lediglich eine prozentuelle Geschwindigkeitsreduktion angeben, existiert keine Vergleichsbasis für andere Parameter.

2.3.4 Zusätzliche Hardwarekomponenten

In der Arbeit *„Efficient Array & Pointer Bound Checking Against Buffer Overflow Attacks via Hardware/Software*" wird ein kombinierter Ansatz bestehend aus Hardwareerweiterung und Softwareoptimierung präsentiert. Ziel dieser Implementierung ist wiederum, C Programme um *„bound checking“* zu bereichern und dabei die Geschwindigkeitsreduktion zu minimieren. Auch dieser Prototyp reduziert zu diesem Zweck die kritischen, zu überprüfenden Zugriffe per Definition auf Schreiboperationen ([Shao et al., 2005]):

> *Regardless which type of buffer overflow attacks, buffers have to be overflowed for an attack to succeed. If all write operations associated with pointers and arrays are ensured to be in bounds, then overflow can not occur. Therefore, we only need to check the bounds of a pointer when its dereference is related to “write” operations for defending against buffer overflow attacks.*

Softwareseitig folgt diese Implementierung der in Kapitel 2.2.2 bereits vorgestellten Optimierungsstrategie. Durch C Quelltexttransformation werden Überprüfungen aus dem Schleifenkörper gezogen bzw. Redundante eliminiert. Im Vergleich zu den übrigen präsentierten Alternativen umfasst dieser Vorschlag jedoch eine Hardwareerweiterung in Form einer eigenen Prozessorinstruktion. Basis dieser Überlegungen ist die RISC Prozessorarchitektur DLX, welche von John L. Hennessy und David A. Patterson – den Designern der MIPS und Berkeley RISC Architektur – ersonnen wurde (siehe [PatHen, 1996]). Da diese Architektur lediglich auf Papier existiert, wurden alle Ergebnisse durch Simulation am SimpleScalar/ARM Simulator gewonnen.

Die Hardwareerweiterung wird durch die neue Prozessorinstruktion „BCK“ angesprochen. BCK übernimmt drei Argumente: die Register mit den Adressen von Wert, oberer und unterer Grenze. Zwei parallel arbeitende ALUs erledigen die arithmetische Überprüfung und speisen im Fehlerfall ein

Out_Bounds Signal, was die Abarbeitung vor Erreichen der *„Execution Stage“* beendet.

`BCK R1,R2,R3`
BCK Instruktion entspricht: if ((R1 < R2) || (R1 > R3)) Generate Out_Bounds exception signal.

Für diese spezielle Instruktion musste das ID/EX Register um einen dritten Operanden erweitert werden. Abbildung 2-3 zeigt die um eine zusätzliche ALU und um ein vergrößertes Register modifizierte Prozessorpipeline.

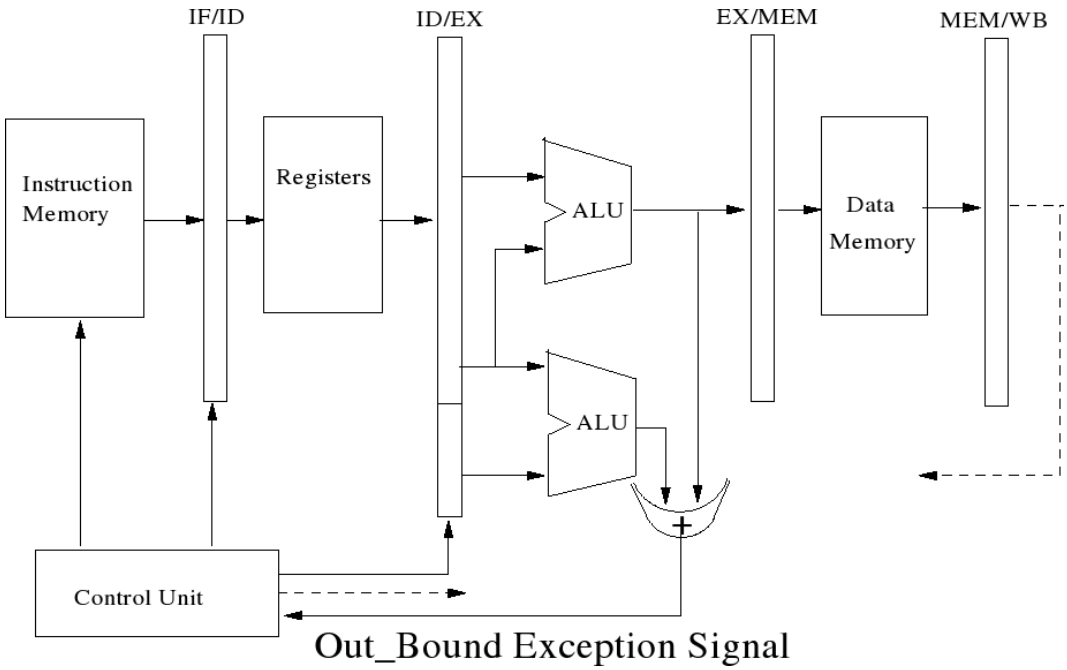

Abbildung 2-3: Modifizierte DLX Architektur
Bildquelle: [Shao et al., 2005]

Die in der Simulation gewonnen Resultate zeigen vergleichsweise gute Aussichten. Im Schnitt beträgt der Geschwindigkeitsverlust ohne Softwareoptimierung ca. 30%. Mit softwareseitigen Codeoptimierungen konnte der Wert auf lediglich 4,5 % gesenkt werden.

Diese Arbeit ist dem Projekt SCPU unter den gezeigten am ähnlichsten. Beide Vorschläge basieren auf einer RISC Architektur, die um parallel arbeitende Komparatoren erweitert wurden. Der Unterschied liegt vor allem in der Handhabung dieser Erweiterungen. SCPU sieht generell bei jeder Operation die Möglichkeit vor, eine Überprüfung der Operatoren vorzunehmen. Diese Arbeit erfordert einen expliziten Aufruf von BCK.

2.4 Zusammenfassung

Die besprochenen Arbeiten versuchen über unterschiedlichste Herangehensweise dem Performanzproblem *„Bound Check“* zu Leibe zu rücken. Zusammengefasst kann gesagt werden, dass die besten Ergebnisse durch softwareseitige Optimierungen erreicht werden können. Zwar bietet auch die Hardwareebene Raum für Optimierung, allerdings sind die möglichen Einsparungen schon rein theoretisch viel kleiner als bei Codeoptimierung. Die vier bis sechs notwendigen Prozessorinstruktionen können lediglich um die zwei Vergleichsoperationen reduziert werden, indem diese z.B. parallel abgearbeitet werden. Dies entspricht einer Reduktion um maximal 2/6. Im Gegensatz dazu kann die Anzahl der durchgeführten Überprüfungen durch geschickte Codeoptimierungen um bis zu 99% reduziert werden.

Eine Kombination dieser beiden Herangehensweisen verspricht die besten Erfolge, was auch die zuletzt gezeigte Arbeit bestätigt. Leider wurde diese Implementierung nur am Simulator durchgeführt, eine konkrete CPU wäre sicherlich der Idee dienlich.

3 Common Language Infrastructure

In diesem Kapitel wird die „Common Language Infrastructure“ (CLI) näher betrachtet. Nach einem Überblick über die historische Entwicklung wird das Konzept näher erläutert und konkrete Implementierungen vorgestellt. Abschließend werden die Unterschiede zu ähnlichen Konzepten - allen voran Java – hervorgehoben.

CLI in der heutigen Form entsprang aus einer Zusammenarbeit von Microsoft, Hewlett-Packard und Intel. CLI ist eine Systemspezifikation, die Programmiersprachenunabhängige und Plattformunabhängige Systeme ermöglichen soll. An dieser Stelle sei hervorgehoben, dass es sich bei CLI um keine Implementierung, sondern wie gesagt, nur um eine Spezifikation handelt.

3.1 Historische Entwicklung

Nachstehend wird ein kurzer Einblick in die historische Entwicklung der Common Language Infrastucture gegeben (siehe [CLIWP, 2007]). Dieser Überblick bezieht sich lediglich auf die CLI Spezifikation. Unabhängig von dieser Entwicklung wurden im Laufe der Zeit eine mehr oder weniger große Anzahl von Implementierungen der Spezifikation geboren und liegen heute in unterschiedlichster Vollständigkeit sowohl als kommerzielle Produkte wie auch Public Domain Software vor. An späterer Stelle wird auf die bekanntesten dieser Umsetzungen noch genauer eingegangen.

- August 2000: Microsoft, HP und Intel reichen die CLI-Spezifikation bei ECMA zur Standardisierung ein.
- Dezember 2001: Die Spezifikation wird unter dem Namen ECMA-335 standardisiert.
- Dezember 2001: Dieselbe Spezifikation wird bei ISO/IEC zur Standardisierung vorgelegt.

- Dezember 2002: Abänderung des Standards zu ECMA-335 2nd edition um Konformität zum ISO/IEC Standard zu wahren.
- April 2003: ISO/IEC standardisieren ECMA-335 2nd edition: ISO/IEC 23271 und ISO/IEC 23272.
- Juni 2005: Erweiterung des Standards um Generics und standardisierte Debuginformationen zu ECMA-335 3nd edition.
- Juni 2006: ECMA-335 4nd edition wird von ECMA standardisiert.

3.2 Architekturüberblick

Die CLI Spezifikation deckt in mehreren separaten Teilen viele Belange rund um die Entwicklung und Ausführung von Applikationen ab. Beginnend mit einer Beschreibung der Gesamtarchitektur, über das verwendete Typensystem, die Sprachspezifikation für sprachübergreifende Nutzung bis hin zu Klassenbibliotheken und einem einheitlichen Debuggingformat umfasst die CLI Spezifikation alles, um Kompatible Module, Programmiersprachen, Compiler, Werkzeuge und ganze CLI Implementationen entwickeln zu können.

Die jeweils aktuelle Version der Spezifikation nach ECMA kann über die Webseite[8] frei bezogen werden. Die vorliegende vierte Version (siehe [ECMA-355, 2006]) erweitert die Vorgängerversion um Kompatibilität zum CLI Standard nach ISO/IEC 23271:2006 (siehe [ISO/IEC 23271,2006]).

Im Detail umfasst das Dokument sechs Hauptteile, „Partitions“ genannt (aus [ECMA-355, 2006]):

- Partition I: *„Concepts and Architecture“*

  Neben der Beschreibung des Gesamtkonzeptes findet man hier die Spezifikation zu den zentralen CLI Komponenten:

  - dem „Common Type System“ (CTS)
  - dem „Virtual Execution System“ (VES)

8 http://www.ecma-international.org/publications/standards/Ecma-335.htm

    - und der „Common Language Spezification“ (CLS)

- Partition II: *„Metadata Definition and Semantics“*
  Beschreibt Inhalt, interne Struktur und Semantik der Metadaten. Metadaten ergänzen CIL-Quellcode um Zusatzinformation die zur Laufzeit ausgewertet werden oder auch lediglich informativen Charakter wie Angaben zum Hersteller haben.

- Partition III: *„CIL Instruction Se*t“
  Spezifiziert alle gültigen Instruktionen die vom VES implementiert werden.

- Partition IV: „Profiles and Libraries“
  Gibt einen Einblick in die spezifizierten Softwarebibliotheken die jede CLI Implementierung enthalten sollte.

- Partition V: *„Debug Interchange Format“*
  Dieser Abschnitt beschreibt das Format für Debuginformationen (CILDB). Damit soll eine definierte Möglichkeit zum Austausch von Debuginformationen zwischen CIL Erzeuger und CIL Produzent ermöglicht werden.

- Partition VI: *„Annexes“*
  Im Anhang finden sich Quellcodebeispiele, Guidlines zum Umgang mit Bibliotheken und Programmiervorschläge um typische Fehler zu vermeiden.

Der Hauptteil der Spezifikation liegt in den ersten vier *„Partitions“* die rund dreiviertel des Umfangs ausmachen. Im Folgenden wird auf die wesentlichen Merkmale näher eingegangen.

3.2.1 Concepts and Architecture

Die beachtenswerte grundlegende Idee der CLI ist, eine von Programmiersprachen unabhängige Umgebung zu schaffen. Dies manifestiert sich darin, dass in einem Zwischenschritt – ähnlich zu Java – ein hardwareunabhängiger Assemblycode, in CIL, erzeugt wird. Aus allen unterstützten Programmiersprachen soll ein Aufruf von Objektmethoden und

Funktionen möglich sein, unabhängig davon in welcher dieser unterstützten Programmiersprachen diese codiert wurden.

Für die Ausführung des Assembycodes auf der darunter liegenden Hardware ist die virtuelle Laufzeitumgebung verantwortlich. Diese stellt damit die einzige Schnittstelle zur CPU[9] und Peripherie dar. Damit steht grundsätzlich einer Plattformunabhängigkeit nichts im Wege, was mittlerweile zu einer beachtlichen Anzahl von unterstützten Plattformen geführt hat.

Die CLI definiert nicht, ob und wann das VES die Assemblies in Maschinencode compilieren muss. So spricht die Spezifikation grundsätzlich nicht gegen eine Abarbeitung durch einen Interpreter. In den bekannten Implementierungen wird allerdings durchwegs angestrebt, einen *„Just in time“* (JIT) Compiler einzusetzen. Diese Technik übersetzt Assemblies im Allgemeinen erst bei erstmaliger Verwendung in Maschineninstruktionen und hält diese Form in einem Zwischenspeicher für weiteren Gebrauch vor. Dies stellt einen guten Kompromiss dar, der die plattform- und sprachneutrale Eigenschaft mit vertretbaren Compilierzeiten verbindet.

Abbildung 3-1 zeigt einen Überblick über die CLI. Anzumerken ist in dieser Abbildung, dass anstelle des laut Spezifikation gültigen Begriffes VES die Bezeichnung *„Common Language Runtime“* (CLR) verwendet wird. CLR bezeichnet die kommerzielle Implementierung des VES von Microsoft im Rahmen der .NET Plattform. Diese Ungenauigkeit findet sich allerdings oftmals in Literatur und Artikeln, und das, obwohl gleichzeitig die Standardisierung der CLI durch ECMA bzw. ISO hervorgehoben wird.

9 CPU ... **C**entral **P**rocessing **U**nit

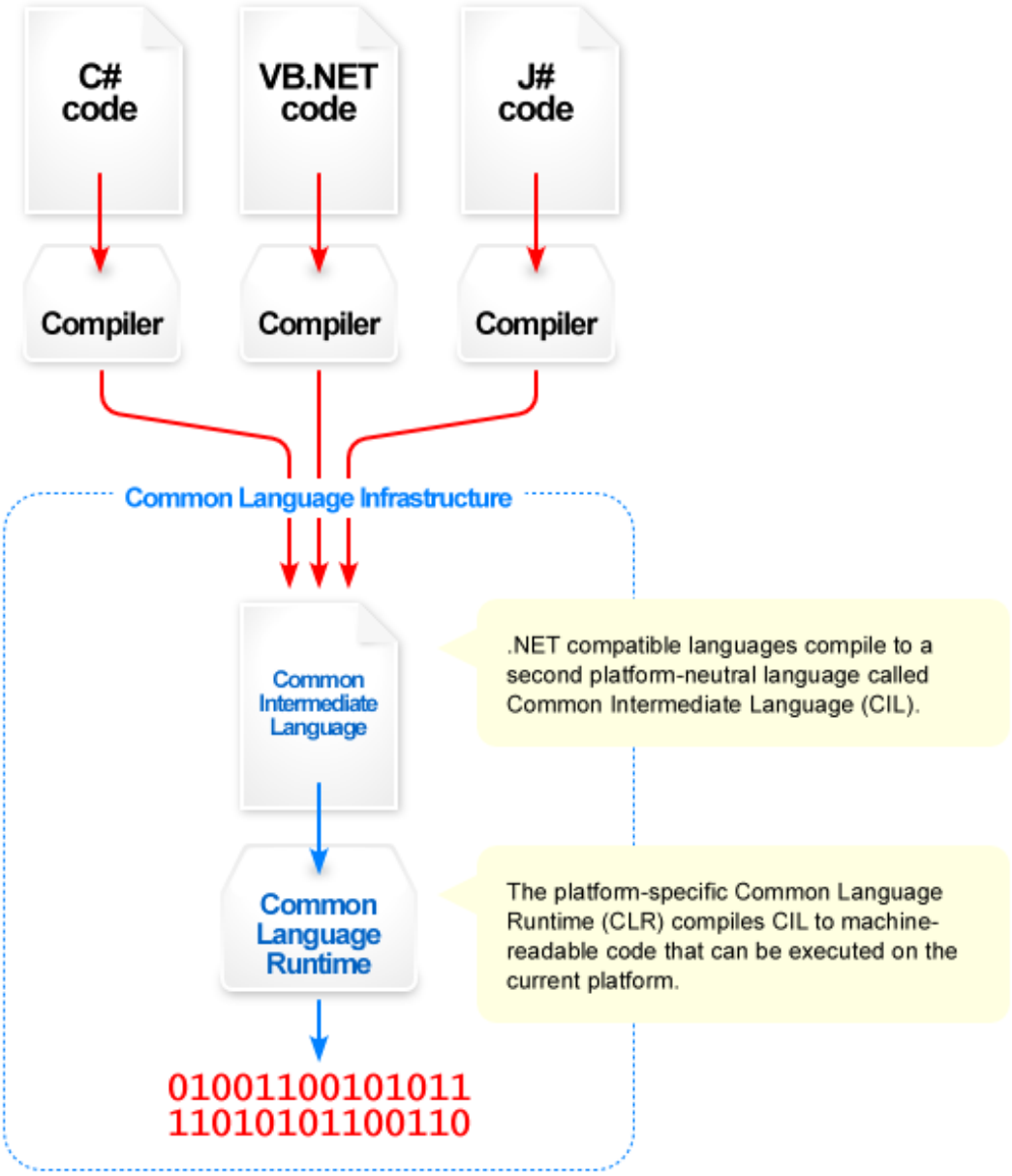

Abbildung 3-1: Überblick Common Language Infrastructure
Bildquelle: [WP, 2007]

3.2.2 Common Type System

Das *„Common Type System“* (CTS) spezifiziert Typen und Typoperationen mit dem Ziel Typsysteme möglichst vieler Programmiersprachen zu unterstützen. Abbildung 3-2 zeigt eine Übersicht über das CTS. Grundlegend sind zwei Klassen von Typen spezifiziert, Werttypen („Value Types“) und Referenztypen („Reference Types“). Werttypen, in der Literatur auch unter dem Namen

primitive Datentypen bekannt, decken grundlegende Datentypen wie Integer- oder Floatwerte ab. Referenztypen umfassen komplexe Datentypen wie Zeiger, Objekte oder Strings.

Mit diesem CTS soll die CLI sowohl objektorientierte, funktionale wie auch prozedurale Programmiersprachen unterstützen.

Folgende Tabelle listet alle integralen Datentypen, diese werden direkt vom VES unterstützt (siehe [ECMA-355, 2006]):

Name im CIL Assembler	CLS Typ	Name in CLI Bibliothek	Beschreibung
`bool`	Ja	`System.Boolean`	true/false value
`char`	Ja	`System.Char`	Unicode 16-bit char
`object`	Ja	`System.Object`	Object or boxed value type
`string`	Ja	`System.String`	Unicode string
`float32`	Ja	`System.Single`	IEC 60559:1989 32-bit float
`float64`	Ja	`System.Double`	IEC 60559:1989 64-bit float
`int8`	Nein	`System.SByte`	Signed 8-bit integer
`int16`	Ja	`System.Int16`	Signed 16-bit integer
`int32`	Ja	`System.Int32`	Signed 32-bit integer
`int64`	Ja	`System.Int64`	Signed 64-bit integer
`native int`	Ja	`System.IntPtr`	Signed integer, native size
`native unsigned int`	Nein	`System.UIntPtr`	Unsigned integer, native size
`typederef`	Nein	`System.TypedReference`	Pointer plus exact type
`unsigned int8`	Ja	`System.Byte`	Unsigned 8-bit integer
`unsigned int16`	Nein	`System.UInt16`	Unsigned 16-bit integer
`unsigned int32`	Nein	`System.UInt32`	Unsigned 32-bit integer
`unsigned int64`	Nein	`System.UInt64`	Unsigned 64-bit integer

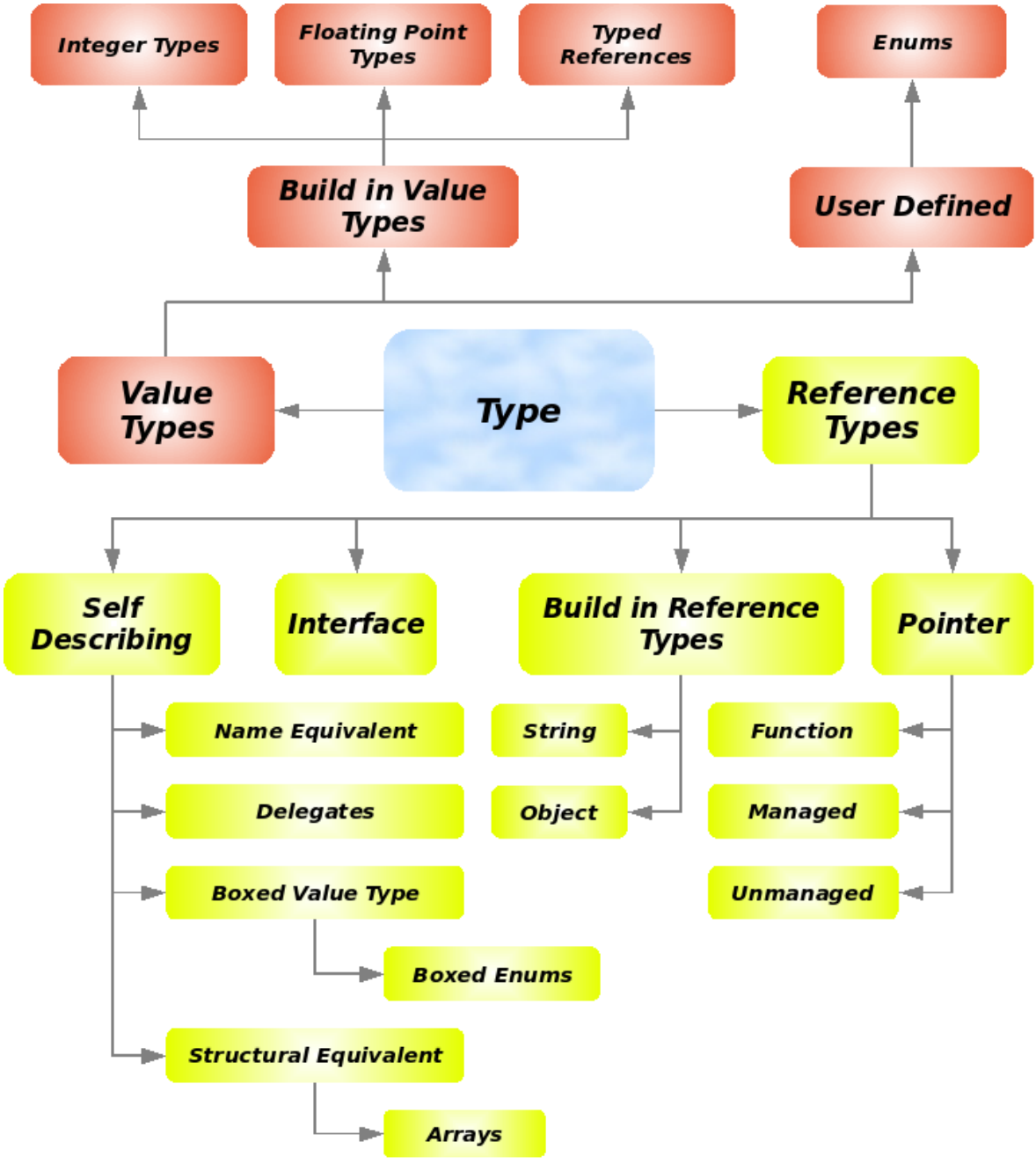

Abbildung 3-2: Überblick Common Type System
Bild nach [ECMA-355, 2006]

3.2.3 Common Language Specification

Die *„Common Language Specification“* (CLS) umfasst eine Menge von Regeln um die Programmiersprachenineroperabilität zu gewährleisten. Damit betrifft die CLS vor allem generierte CIL Assemblies. Diese Regeln haben Auswirkung auf unterschiedliche Bereiche. Aus diesem Grund ergänzt die

CLS jede angegebene Regel um die Angabe, welche Bereiche davon berührt sind. Hierzu werden drei Sichtweisen auf die CLS eingeführt:

- CLS framework
  Das CLS framework bezeichnet CIL-Softwarebibliotheken, die von allen unterstützten Programmiersprachen verwendet werden können.
- CLS consumer
  Ein CLS consumer bezeichnet Programmiersprachen und Werkzeuge, die das CLS framework verwenden können, allerdings nicht selbst produzieren können.
- CLS extender
  Als CLS extender werden Programmiersprachen und Werkzeuge bezeichnet, die CLS framework verwenden und erweitern kann.

Insgesamt werden in der vorliegenden Version 48 Regeln definiert, eine Zusammenfassung aller CLS Regeln ist in [ECMA-355, 2006] S74 ff. angeführt.

3.2.4 Virtual Execution System

Das „Virtual Execution System“ (VES) implementiert das CTS Modell. Es ist verantwortlich für das Laden und Ausführen von Programmen, die für die CLI geschrieben wurden. Das VES ist als virtuelle Maschine zu verstehen, wobei diese – wie auch in Java – als Stack Machine spezifiziert ist. Dies hat für konkrete Implementierungen den entscheidenden Vorteil, dass diese relativ einfach auf verschiedenste Hardwarestrukturen wie Registermaschinen oder Akkumulatormaschinen umsetzbar sind.

Die Struktur einer Stack Machine zeichnet sich durch den Umstand aus, dass der Operator den Operanden folgt. Um beispielsweise zwei Integer zu addieren, werden zuerst die beiden Werte auf den Stack geschrieben, um in Folge mittels der Operation `add` diese beiden Werte zu addieren, und wiederum auf den Stack zu schreiben. Diese Vorgehensweise ist auch von Taschenrechnern mit RPN[10], also Postfixnotation, bekannt.

10 RPN oder auch UPN ... **R**everse **P**olish **N**otation

Folgendes Codebeispiel zeigt die Addition zweier Werte, die als Parameter einer Funktion übergeben werden, in CIL-Syntax:

```
...                 # Funktionskopf
ldarg.0             # erstes Argument auf den Stack laden
ldarg.1             # zweites Argument auf den Stack laden
add                 # Zwei obersten Werte vom Stack addieren
ret                 # obersten Wert vom Stack zurückgeben
```

Addition zweier Werte in CIL-Syntax

Die Spezifikation des VES regelt folglich alles, um *„CIL Instruction Set“* ausführen zu können. Dies umfasst Ausnahmebehandlung, Kontrollfluss und unterstützte Datentypen.

3.2.5 Common Language Specification

Die Common Language Specification (CLS) ist als Übereinkunft zwischen Programmiersprachendesigner und Bibliotheksdesigner zu verstehen. Sie spezifiziert das minimale Set an Typen des CTS und deren Verwendung.

3.2.6 Metadata Definition and Semantics

Partition II der ECMA Spezifikation widmet sich der Semantik und Struktur der Metadaten bzw. generell der Dateistruktur eines CIL Programmes.

Inhalt einer CIL Datei sind neben dem eigentlichen Programmcode auch Informationen wie Abhängigkeiten zu anderen Bibliotheken, Typinformationen und Versions bzw. Autorinformationen. Abbildung 3-3 zeigt den schematischen Aufbau einer CIL Datei.

Ein kurzes Codebeispiel illustriert den internen Aufbau eines CIL Programmes.

```
.assembly extern mscorlib {}
.assembly hello {}
.method static public void main() cil managed
{
    .entrypoint
    .maxstack 1
    ldstr "Hello world!"
    call void [mscorlib]System.Console::WriteLine(class
                                                   System.String)
   ret
}
```

Hello world in CIL

Aufgabe des kurzen Programmes ist, die Zeile „Hello world!“ auf der Konsole auszugeben. Dies geschieht durch den Aufruf der statischen Methode `WriteLine` der Bibliotheksklasse `System.Console`. Zeile eins gibt die Abhängigkeit zur externen Bibliothek mscorlib an, in welcher sich die verwendete Funktion findet. Zeile zwei spezifiziert den Namen des Assemblies „`hello`“. Die Deklaration `.method` definiert die globale Methode main als managed CIL Code. `.entrypoint` zeigt an, dass die Methode `main()` der Einstiegspunkt des Programmes ist. Die Anweisung .maxstack legt fest, wie groß der Stack für dieses Assembly maximal werden kann.

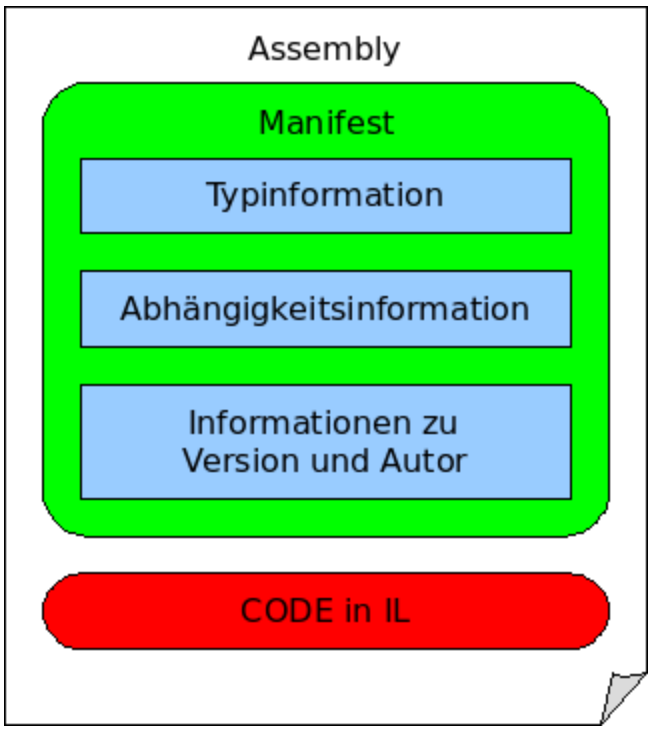

Abbildung 3-3: Aufbau CIL Assembly

Die eigentliche Programmfunktionalität ist lediglich durch die drei Instruktionen `ldstr`, `call` und `ret` implementiert. Die Instruktion `ldstr` gibt die Zeichenkette „`Hello world!`“ auf den Stack. call ruft die externe Methode `System.Console::WriteLine` mit einem String als Argument auf. Die Anweisung `ret` beendet die Methode `main` und damit die Programmausführung.

3.3 Implementierungen

Im Folgenden wird kurz auf die derzeit verfügbaren, relevanten Implementierungen der CLI Spezifikation eingegangen.

Microsoft hat federführend an der Spezifikation der CLI mitgewirkt. Es erscheint nicht weiter verwunderlich, dass aus diesem Hause gleich mehrere Implementierungen dieser Spezifikation stammen.

3.3.1 Microsoft .NET Framework

Mit Version 1.0 begann Microsoft im Jahre 2000 die kommerzielle Variante für Windows PC-Systeme unter dem Namen .NET Framework zu vertrieben. Mittlerweile ist .NET mit Version 3.0 als Teil jedes aktuellen Windowsbetriebssystems ein vollwertiger Ersatz für das vormals für Windows bereitgestellte Component Object Model (COM).

Einige konkurrierende Implementierungen orientieren sich trotz vorliegendem Standard in Bezug auf Funktionalität an den einzelnen Versionen dieser Implementierung von Microsoft. Um an späterer Stelle den „Rückstand“ vergleichbarer Produkte besser einschätzen zu können, wird an dieser Stelle ein Überblick über bisherige und geplante Versionen von Microsofts -NET Framework gegeben (aus [NETWP, 2007]):

- 2002: .NET V1.0
- 2003: .NET V1.1
- 2005: .NET V2.0

- 2006: .NET V3.0
- 2007: .NET V3.5 geplant
- 2009: .NET V4.0 geplant

Neben den standardisierten Eigenschaften wurde das .NET Framework in diesem Zuge um viele Komponenten erweitert. Diese proprietären Erweiterungen sind oftmals Grund für Kritik, da damit der offene Gedanke der Standardisierung verloren gänge. Ungeachtet dieser Kritik bietet Microsoft mit den zugehörigen Entwicklungsumgebungen wie VisualStudio eine umfangreiche Plattform für Softwareentwickler. Nachstehend werden die wesentlichen Erweiterungen kurz erläutert (aus [NETWP, 2007]).

- Windows Presentation Foundation (WPF) zur Erstellung von komplexen, leistungsfähigen grafischen Oberflächen
- Windows Communication Foundation (WCF) zum einfachen Nachrichtenaustausch zwischen Prozessen
- Windows Workflow Foundation (WWF)
- Windows CardSpace (WCS) zur sicheren Personalisierung von Anwendungen

Abbildung 3-4 zeigt ein Strukturbild der derzeit aktuellen Version 3.0, neben den genannten Erweiterungen wird oftmals auch die Entwicklungsumgebung VisualStudio .NET zum Gesamtkonzept hinzu gezählt.

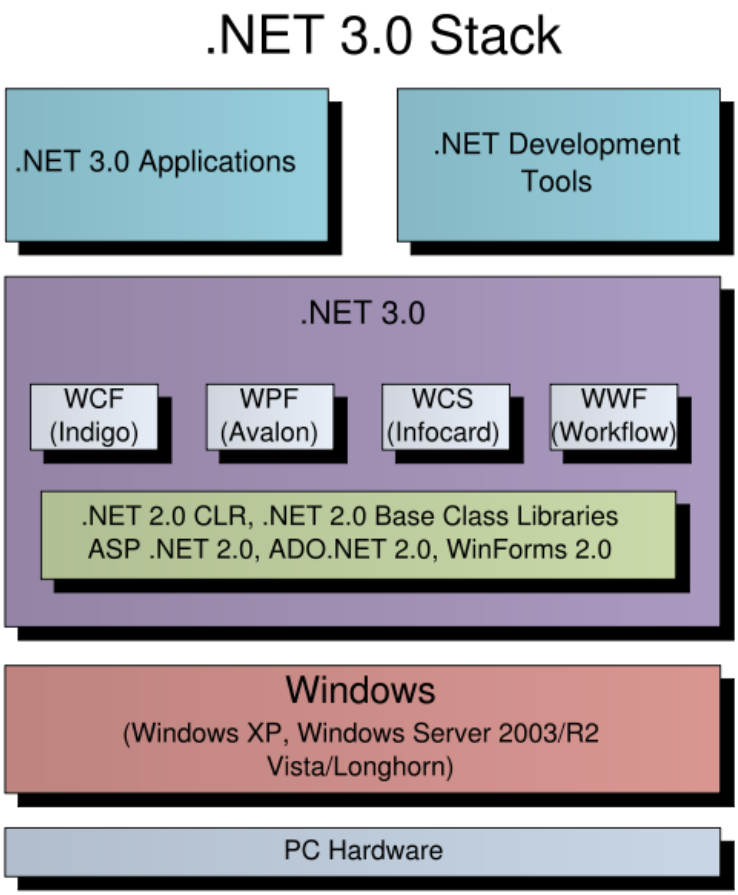

Abbildung 3-4: .NET Framework
Bildquelle: [NETWP, 2007]

3.3.2 Alternative Implementierungen von Microsoft

Microsoft bietet zusätzlich zwei weitere Implementierungen der CLI an:

- Shared Source Common Language Infrastructure (vormals Rotor)
  ist eine offene Implementierung für Windows und FreeBSD. Sie soll vor allem Schulungszwecken und als Referenzimplementierung des ECMA Standards dienen, ein Einsatz für kommerzielle Zwecke ist aus Lizenzgründen[11] nicht möglich.
- .NET Compact Framework (-NET CF)
  ist eine spezielle Version des -NET Frameworks für mobile Endgeräte wie PDAs und Mobiltelefone.

11 http://www.microsoft.com/resources/sharedsource/licensingbasics/sharedsourcelicenses.mspx

3.3.3 Mono

Selbsterklärtes Ziel von Mono ist, eine .NET-kompatible Laufzeitumgebung auf Basis der ECMA Standards 334 und 335 für Linux, Solaris, Mac OS, Windows und Unix zu bieten [MONO, 2007]. Dies inkludiert neben den im Standard vorgeschriebenen Funktionalitäten, bestmögliche Unterstützung für Microsofts Erweiterungen.

Mono wurde im Jahr 2001 als eigenständiges Open Source Projekt gestartet. Starke Unterstützung erfuhr Mono anfangs durch die Firma Ximian, welche im Jahr 2004 von Novell gekauft wurde. Gerade durch diese potente Firma im Hintergrund erfährt Mono eine stete Weiterentwicklung und Pflege. Unter den Alternativen zu Microsofts Lösung gehört Mono sicherlich zu den weit verbreiteten und beliebten.

Die derzeitig aktuelle Version Mono 1.2 unterstützt die .NET API in Version 1.1 vollständig und Teile von .NET 2.0. Den Entwicklungsplänen[12] ist zu entnehmen, dass vollständige .NET 2.0 Unterstützung erst mit Version 2.2 geplant ist. Microsofts Erweiterungen in den Folgeversionen 3.0 bzw. 3.5 werden in separaten Mono-Projekten bearbeitet und fließen vorerst noch nicht in den Releaseplan von Mono ein.

Gegen einen Einsatz von Mono in diesem Projekt hat vorwiegend der Umstand gesprochen, dass derzeit keine Unterstützung für SPARC/Linux existiert. Da Mono stark auf eine Laufzeitumgebung mit JIT setzt, und der vorhandene CIL-Interpreter nicht mehr aktualisiert und unterstützt wird, erschien eine Portierung schwieriger als beim gewählten Produkt portable.NET.

3.3.4 portable.NET

In diesem Projekt wurde die CLI Implementierung des DotGNU Projekts[13] mit Namen portable.NET eingesetzt. Wie auch bei Mono hat sich auch dieses

12 http://www.mono-project.com/Mono_Project_Roadmap
13 http://www.dotgnu.org/

Projekt einer zu Microsoft .NET kompatiblen Implementierung gewidmet. Portable.NET zeichnet sich aber vor allem dadurch aus, dass bereits viele Architekturen unterstützt werden. Neben x86, PPC, ARM und einigen weiteren Architekturen wird auch SPARC als unterstützte Plattform genannt. Diese Nennung bezieht sich allerdings nur auf Erfahrungen mit Solaris auf SPARC, derzeit finden sich keine Hinweise auf andere Portierungen von portable.NET auf Linux@SPARC.

Das VES (in diesem Kontext auch „Runtime Engine“) von Portable.NET ist eine genauere Betrachtung wert. Im Gegensatz zu den anderen Umsetzungen konvertiert portable.NET CIL Bytecode in einen einfacheren Befehlssatz, „Converted Virtual Machine“ (CVM) genannt.

> *Because interpreting CIL bytecode directly is quite inefficient, we take a different approach. We first convert the CIL bytecode into a simpler instruction set for what we call the Converted Virtual Machine (CVM). The simpler CVM instructions are then executed using a high-performance interpreter.*
>
> *...*
> *The CVM approach gives us many of the benefits of a Just-In-Time compiler (JIT), in that the opcodes can be tailored to handle system differences (e.g. 32-bit vs 64-bit CPU's). At the same time, the engine's source code is highly portable to new platforms [MONO, 2007].*

Der Zwischenschritt der Umwandlung auf CVM erfolgt auf Methodenebene, d.h. es wird nicht das gesamte auszuführende Programm auf ein mal in diesen Befehlssatz übersetzt, sondern nur die aktuell benötigten Methoden.

Das VES von portable.NET, die Laufzeitumgebung „`ilrun`“ arbeitet CLI Programme in fünf Schritten ab:

1. loading
   Die CLI Applikation wird geladen und in ein interne Repräsentation gebracht.

2. verification
   Dieser Schritt kommt erst zu tragen, wenn eine Methode aufgerufen wird. Der geforderte CIL Bytecode für die Methode wird analysiert und auf Korrektheit überprüft.
3. coding
   Dieser Schritt überführt den IL Bytecode in andere (effizientere) Formate wie CVM oder nativen Maschinencode (derzeit nur CVM). CVM Code wird für spätere Verwendung in einem Methodencache gespeichert.
4. CVM interpreter
   CVM Code wird durch einen Interpreter ausgeführt.
5. class library support
   Bibliotheksfunktionen wie Strings, Files, Sockets oder Threads werden eingebunden.

Grundidee hinter dieser Vorgehensweise ist, eine Basis für unterschiedliche virtuelle Maschinen zu bieten. Ähnlich zur GCC[14] soll es möglich sein, Backends für mehrere Bytecode Systeme einzubinden. Derzeit ist allerdings „nur“ ein Backend für die CLI, definiert durch ECMA-335[15], implementiert. Zusätzliche Backends für die „Java Virtual Machine“ (JVM) und für „Perl 6 Parrot“ sind geplant (siehe [dotgnu, 2007]).

Abgesehen von dieser, im Vergleich zu anderen CLI Implementierungen, grundlegend verschiedenen Programmabarbeitung im VES, kann auch der CIL Compiler mit einigen Besonderheiten aufwarten. Der portable.NET Compiler `cscc` ist ein modularer Compiler der Unterstützung für eine Vielzahl von Programmiersprachen bieten will. Zur Zeit werden front-ends für C# laut ECMA C# Language Specification ECMA-334[16] und für ANSI-C angeboten, Unterstützung für Java und VB.NET sind in Arbeit (siehe [dotgnu, 2007]).

14 GCC ... **G**NU **C**ompiler **C**ollection
15 http://www.ecma-international.org/publications/standards/Ecma-335.htm
16 http://www.ecma-international.org/publications/standards/Ecma-334.htm

3.4 Vergleich mit Java

In Zusammenhang mit der CLI bzw. mit .NET von Microsoft wird oftmals der Vergleich mit Java bemüht. Erste Vorversionen von Java wurde im Auftrag von Sun Microsystems im Jahr 1992 entwickelt. Ziel war eine plattformunabhängige Laufzeitumgebung samt einfacher Programmiersprache zu schaffen. Öffentliche Aufmerksamkeit wurde jedoch erst mit einer Version im Jahre 1995 erreicht. Mit der Integration der Plattform in den Internetbrowser Netscape erlebte Java einen weiteren Aufschwung. Im Jahr 1996 wurde Version 1.0 von Sun offiziell vorgestellt. Es folgten Erweiterungen in vielerlei Hinsicht. Sowohl die Programmiersprache Java wurde beispielsweise um Generics bereichert, als auch die Laufzeitumgebung samt Systembibliotheken erfuhren Erweiterungen ([Abrams, 1998]).

Auch Sun hatte Bestrebungen ihre Java Plattform standardisieren zu lassen. Im Jahr 1997 wurden konkrete Bemühungen unternommen, jedoch hat sowohl ECMA wie auch ISO/IEC diese Anträge verworfen. Es konnte nicht plausibel dargelegt werden, dass die Motivation zum Standardisierungsprozess technischen Ursprung hat, vielmehr wäre eine bessere Marktposition damit angestrebt worden([Egyedi, 2001]). Vielleicht auch um in diesen Aspekten gegen die erfolgreiche CLI bestehen zu können wurden Kernkomponenten von „Java Platform“ im Jahr 2007 von Sun unter die GPL gestellt(siehe [Sun, 2007]).

Diese Gegenüberstellung hat technisch gesehen durchaus Berechtigung, Beide Plattformen verwenden einen virtuellen Maschinenbefehlssatz, in den Quellcodeprogramme transferiert werden.

Ein Vergleich mit Java ist jedoch nur im korrekten Kontext zulässig. *„Java Technology“* umfasst zum einen die Programmiersprachen Java, zum anderen auch die zugehörige Laufzeitumgebung *„Java Platform“*. Das Pendant zu CLI ist hierbei in der *„Java Platform“* zu finden. Die Programmiersprache Java kann bestenfalls mit der Programmiersprache C# verglichen werden. Im Weiteren wird jedenfalls mit Java die Laufzeitumgebung von Sun gemeint.

Oftmals wird in diversen Abhandlungen rund um den Vergleich von Java mit CLI diese Unterscheidung zu wenig hervorgehoben bzw. zum Teil sogar unterschlagen.

Bezieht man sich auf einschlägige Medien und Literatur, kann der propagierte Unterschied folgend subsumiert werden:

> Java hat sich zum Ziel gesteckt, plattformunabhängige Anwendungen zu ermöglichen.
>
> Für die CIL steht die Programmiersprachenunabhängigkeit im Vordergrund.

Diese beiden Aussagen können in Anbetracht der Entwicklung bestenfalls als einführende Marketingargumente angesehen werden. Sowohl für Java existieren beispielsweise mit JRuby[17] und Jython[18] zwei Compiler, die aus den fremden Programmiersprachen Ruby bzw. Python, Java-VM tauglichen Bytecode erzeugen. Als auch von der CLI gibt es mittlerweile eine kleine Anzahl von bewährten Implementierungen, die auf eine Vielzahl von Hardwareplattformen portiert wurde.

17 http://jruby.codehaus.org/
18 http://www.jython.org/

4 Implementierung

In diesem Kapitel wird auf die konkreten Implementierungen im Zuge dieser Diplomarbeit eingegangen.

Ein Teil der Umsetzung fällt auf die Portierung des portable.NET Interpreters auf die Zielplattform SnapGear-Linux laufend auf einer Sparc-Architektur. Darüber hinaus wurden Modifikationen am portierten Interpreter vorgenommen. Diese zielen darauf ab, die zur Laufzeit im Zuge von Feldzugriffen anfallenden *bound checks,* mithilfe der hardwareunterstützen Wertebereichsüberprüfung auf Basis der SCPU Implementierung auszuführen. Zur Verifikation des Systemverhaltens und zur Gewinnung von Messdaten für eine Laufzeitevaluierung wurden beispielhaft Anwendungen in C# Syntax programmiert. Diese machen exzessiv Gebrauch von ein- und mehrdimensionalen Feldzugriffen und damit von den getätigten Interpretermodifikationen.

Um eine möglichst einfache Reproduzierbarkeit des Testaufbaus zu gewährleisten, wird in diesem Kapitel versucht, möglichst detailliert alle wichtigen Schritte zur Installation, Inbetriebnahme und Bedienung der diskutierten Hard- und Softwareumgebung zu beschrieben. Um den Umfang dieser Anleitung aber nicht zu sprengen, werden grundsätzliche Basiskenntnisse im Umgang mit Linuxsystemen - speziell mit der verwendeten Distribution Ubuntu[19] - und den damit üblicherweise verwendeten Werkzeugen wie der Paketverwaltung apt[20] vorausgesetzt.

Zuerst werden alle verwendeten Hardwarekomponenten hinsichtlich ihrer Funktion, wie auch deren Interaktion mit anderen Komponenten beschrieben. Im Anschluss wird die Synthetisierung des Leon2 in der modifizierten Version SCPU sowie die Programmierung des FPGA[21] grob erklärt. Eine umfangreichere Diskussion mit detaillierter Beschreibung zur Harwaresynthese und zu den getätigten SCPU spezifischen Modifikationen ist

19 http://www.ubuntu.com/
20 apt ... **A**dvanced **P**ackaging **T**ool
21 FPGA ... **F**ield **P**rogrammable **G**ate **A**rray

in [Grasser, 2007] beschrieben. Der darauf folgende Abschnitt behandelt die Installation und den Umgang mit SnapGear – der eingesetzten Linuxdistribution - für die Zielplattform. Um schlussendlich auf der Zielplattform die Demonstrationsprogramme ausführen zu können und damit Testergebnisse zu gewinnen, wird anschließend die Integration von portable.NET in SnapGear beschrieben.

4.1 Arbeitsumgebung

Unter Arbeitsumgebung ist in diesem Zusammenhang der Komplex aller beteiligten, grundsätzlich unabhängigen, Einzelkomponenten zu verstehen. Dieser Abschnitt soll nebenbei die grundlegensten Voraussetzungen zur Reproduktion des Testaufbaus aufzeigen.

Die verwendete Arbeitsumgebung setzt sich aus den folgenden drei Hauptkomponenten zusammen:

- die Zielplattform
- ein Windows PC zur Synthetisierung und Programmierung des FPGA Prozessors
- ein Linuxsystem zur Erstellung des lauffähigen Zielsystems und zur Kommunikation mit dem laufenden Zielsystem

Abbildung 4-2 zeigt einen schematischen Überblick über die eingesetzten Hauptkomponenten inklusive aller verwendeten Kommunikations-schnittstellen. Hierbei ist allerdings einzuräumen, dass der gezeigte Aufbau keinesfalls der einzig mögliche ist. Klarerweise sind insbesondere die PC-Systeme flexibel zu konfektionieren. So ist es leicht möglich, die beiden verwendeten PCs durch einen zu ersetzen, sofern dieser über zwei COM Schnittstellen verfügt. Dieser einzige Entwicklungsrechner könnte dabei sowohl unter Windows wie auch Linux betrieben werden. Eine Virtualisierungslösung mit beiden gezeigten Systemen auf einem PC wäre ebenfalls denkbar.

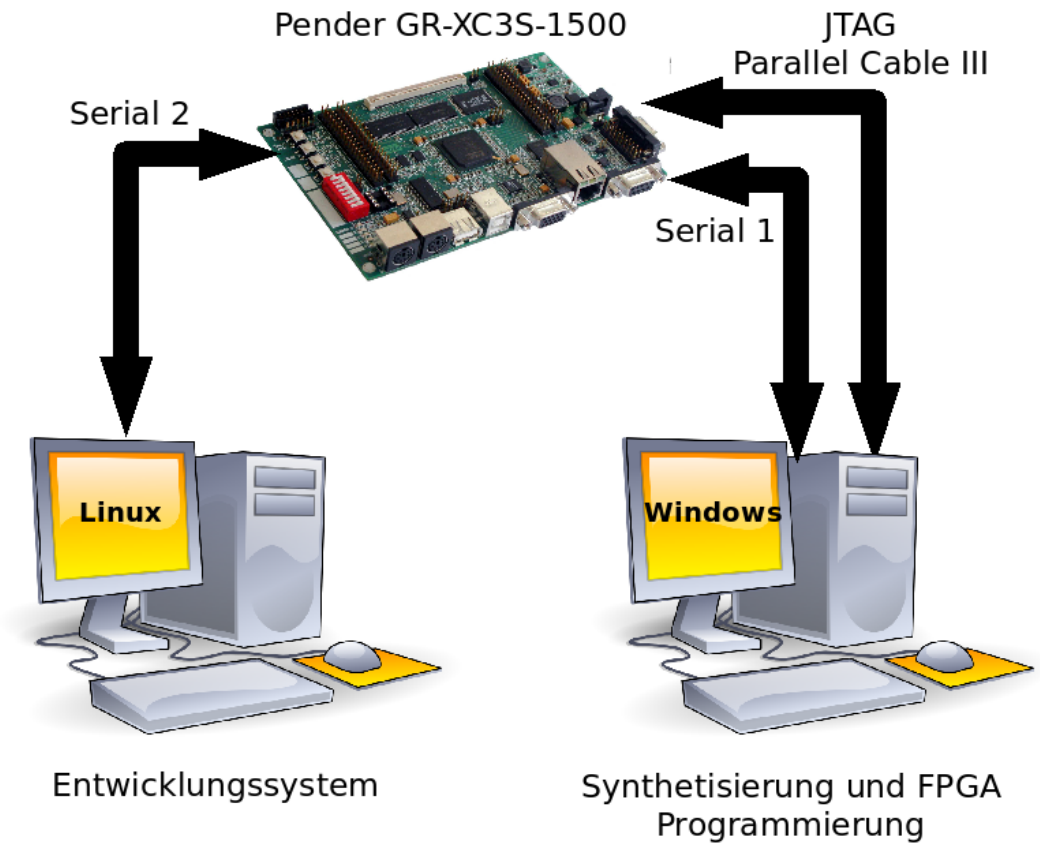

Abbildung 4-1: Überblick Arbeitsumgebung

Der verwendete Aufbau ist mitunter zu einem großen Teil durch die historische Entwicklung des SCPU-Projektes bedingt. Die dabei entwickelte, auf mehrere Rechner verteilte, *toolchain* war alleine durch die Trennung von Hardwareentwicklung und Softwareentwicklung auf mehrere Arbeitsstationen und Personen naturgemäß vorgegeben. Die Wahl zweier verschiedener Betriebssysteme war ebenso ungeplant, jedoch auch nicht weiter störend. Zur Hardwaresynthetisierung wird lizenzbedingt ein Windowsprodukt verwendet. Die damit erzeugten Dateien zur FPGA Programmierung (siehe auch beiliegende CD) können allerdings, sofern unverändert verwendbar, auch ohne dieses Softwarepaket eingesetzt werden.

Zur Konfiguration und Compilierung von SnapGear erschien ein Linuxsystem mit konsistenter Paketverwaltung und umfangreicheren Boardmitteln hingegen geeigneter. Die beiliegende CD enthält alle Dateien zur Reproduktion des Versuchsaufbaus. Eingesetzte Software, sowie

urheberrechtlich geschützte Werke sind mitunter aus Speicherplatzgründen nicht Teil der CD-ROM.

4.1.1 Pender GR-XC3S-1500

Die eingesetzte Hardwareplattform GR-XC3S-1500 der Firma Pender[22] versteht sich als preiswertes Entwicklungssystem für die Evaluierung von Leon2 und Leon3/GRLIB Prozessorsystemen [Pender, 2006]. Abbildung 4-3 zeigt das unkonfigurierte, nicht erweiterte Board im Auslieferungszustand.

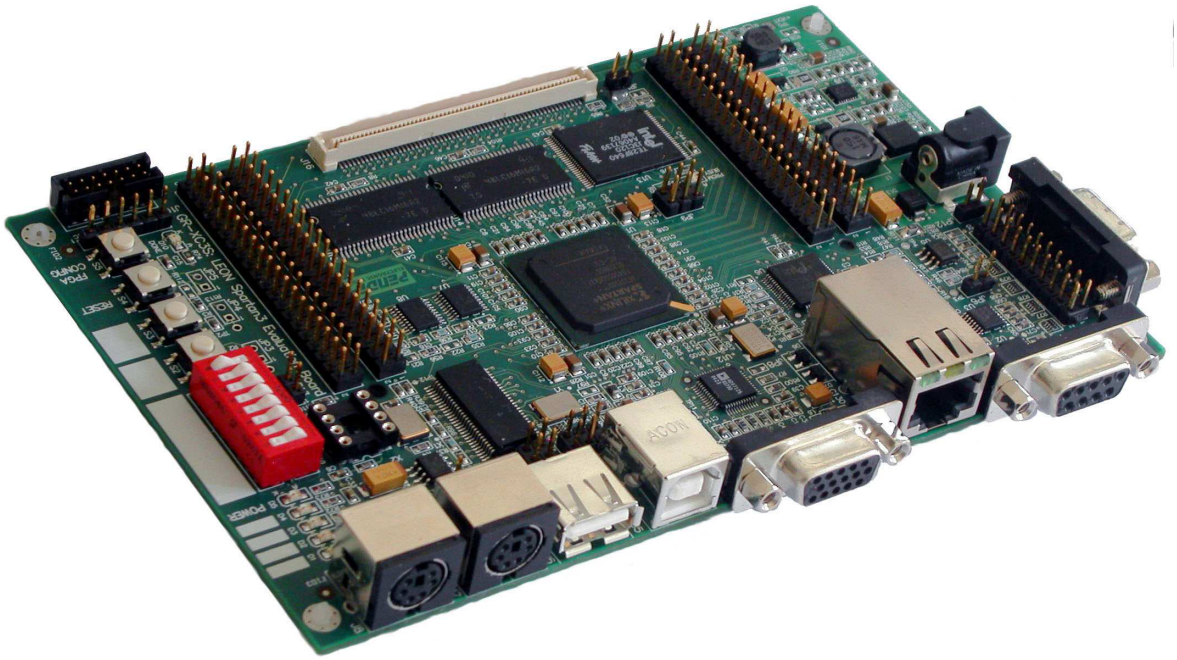

Abbildung 4-3: Pender GR-XC3S-1500
Bildquelle: [Pender, 2006]

Das Entwicklungsboard ist Produkt einer Zusammenarbeit von Gaisler Research und der schweizer Firma Pender Electronic Design GmbH. Kern der Plattform ist die XC3S1500-4FG456 FPGA Einheit der Xilinx Spartan3 Familie mit 1.5 Millionen programmierbaren Gattern.

Zwei Flash PROMs mit 1 x 4 Mbit und 1 x 1 Mbit Speicherkapazität dienen dem nicht flüchtigen Abspeichern der FPGA Konfiguration. Das Board verfügt überdies über 64 Mbit (8 Mbyte) FLASH PROM und 512 Mbit (64 Mbyte) PC-

22 http://www.pender.ch/

133 SDRAM Speicher. Über eine Speichererweiterungsschnittstelle („memory expansion connector“) kann das Board mit zusätzlichem SRAM Speicher aufgerüstet werden.

Maus und Tastatur können über die standardisierten PS2 Anschlüsse betrieben werden. Zwei SUB-D9 Schnittstellen ermöglichen die Kommunikation mit Fremdsystemen, wobei eine dieser für die DSU[23] reserviert ist. Die Verbleibende kann für Applikationen verwendet werden. Ein VGA kompatibler Ausgang ermöglicht Ausgaben an einen direkt angeschlossenen Monitor. Eine Intel LXT971A Ethernet Einheit ermöglicht mit der zugehörigen 10/100 Mbit/s RJ 45 Schnittstelle Anwendungen mit Netzwerkkommunikation. Ein Cypress CY7C68000 USB 2.0 bietet mit USB-A wie auch USB-B Anschlüssen flexible Nutzung von USB-Verbindungen von sowohl Host-zu-Host wie auch Peripheriegeräten. Über den 120 Pin Erweiterungssteckplatz können mit entsprechenden Adaptern „logic analyzer“ wie auch Speichererweiterungen betrieben werden. Die JTAG-Schnittstelle[24] unterstützt sowohl Parallel Cable III (Flying leads) wie auch Parallel cable IV (2x7pin 2mm header) für die Programmierung des FPGAs (siehe [Pender, 2006]). Abbildung 4-3 zeigt das Blockschaltbild des Entwicklungsboardes mit den wichtigsten Komponenten.

Die Entscheidung, im Zuge des SCPU Projektes auf dieses Entwicklungsboard umzusteigen, kann als durchaus gelungen bewertet werden. Im Gegensatz zum in der Projekteingangsphase eingesetzten Entwicklungsboard XESS XSV800[25] mit Virtex FPGA und lediglich 2 MByte Speicher, verfügt das Pender Board über wesentlich mehr programmierbare Gatter und auch mehr Onboardspeicher.

23 DSU ... **D**ebug **S**upport **U**nit
24 JTAG .. **J**oint **T**est **A**ction **G**roup
25 http://www.xess.com/

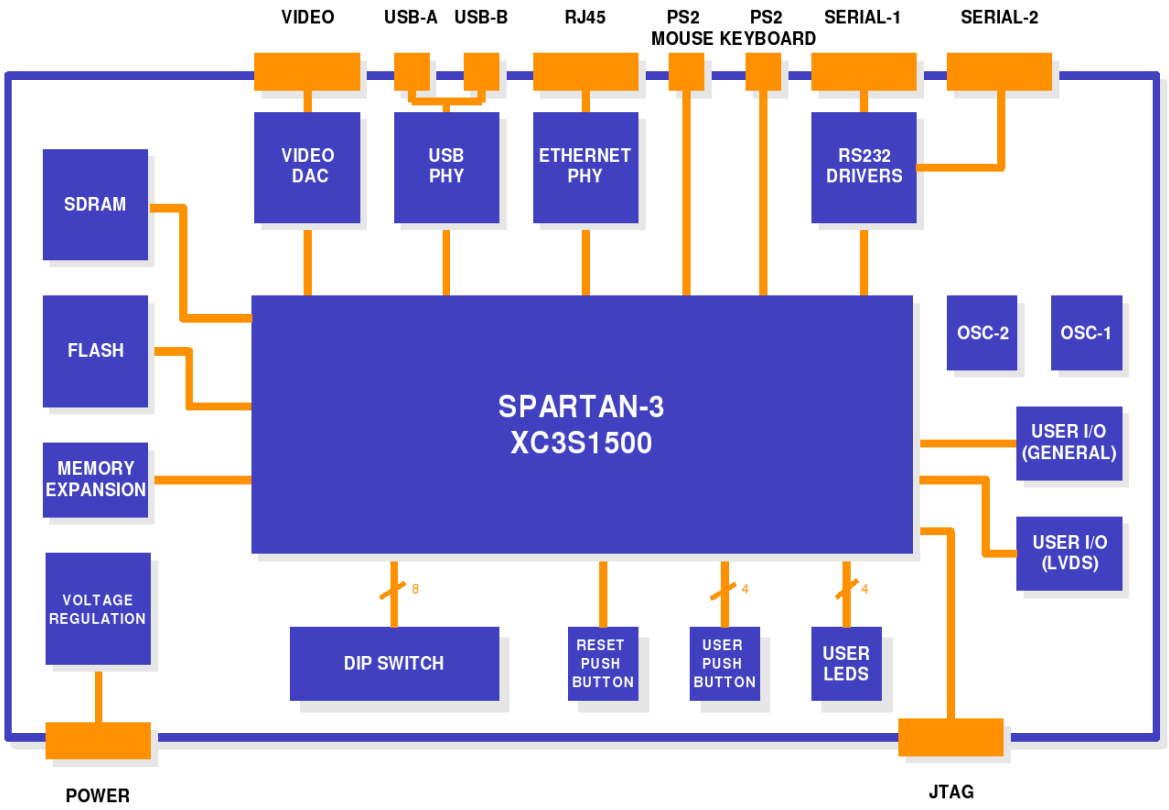

Abbildung 4-4: Blockschaltbild Pender GR-XC3S-1500
Bildquelle: [Pender, 2006]

Die damit einhergehenden Möglichkeiten bezüglich der eingesetzten Betriebssysteme und darauf laufenden Softwareapplikationen sind alleine aufgrund des größeren Speichers ungleich höher. Die Unterstützung von Seiten Gaisler Research für das Pender Board ist beispielhaft. So ist ein guter Teil der technischen Dokumentation, welche von Gaisler für Leon2 und Leon3 angeboten wird, auf dieses Board zugeschnitten. Auch Gaisler Research selbst empfiehlt dieses Board für die Entwicklung von Leon Systemen. Dieser Empfehlung dürften offenbar viele Entwicklungsteams Folge leisten, so finden sich im einschlägigen leon_sparc Forum bei Yahoo-Groups[26] eine Vielzahl von Themen rund um die Leon-Entwicklung in Zusammenhang mit diesem Entwicklungsboard.

26 http://tech.groups.yahoo.com/group/leon_sparc/

4.1.2 Windows Synthetisierungsplattform

Zur Synthetisierung des Leon2 Kerns und anschließender Programmierung des FPGAs wurde ein Windowssystem auf handelsüblicher PC-Hardware verwendet. Um die Geduld während Kompilation und Synthetisierung des VHDL-Codes nicht all zu sehr zu strapazieren, empfiehlt es sich, bei Speicher und Prozessortakt nicht zu sparen. Der in diesem Projekt verwendete Windows-PC kann mit ca. 2 GHz Prozessortakt und 1024 MByte Hauptspeicher die Anforderungen an die Verwendete „Place&Route“ Software von Xilinx[27] voll erfüllen.

Die Programmierung des FPGAs kann grundsätzlich mit einer Vielzahl von Programmen erfolgen. In diesem Projekt wurde Xilinx ISE 8.1i verwendet. Dieses Programmpaket vermag alle Xilinx FPGAs zu programmieren, womit auch das verwendete Entwicklungsboard Unterstützung erfährt. Eine korrekte Verbindung des Entwicklungsboardes vorausgesetzt, ist Impact in der Lage, die angeschlossene Hardware über die JTAG Verbindung auszulesen und zu erkennen. Dies erleichtert die Handhabe der Entwicklungshardware erheblich, ein manuelles Konfigurieren der Hardwareparameterm, wie beim zuvor verwendeten Xess-Board, ist nicht notwendig.

Für die korrekte Verbindung der Entwicklungshardware zwecks FPGA Programmierung mit dem Windows-PC stehen mehrere Möglichkeiten offen. Da nicht alle dieser Optionen eine Kommunikation mit Leon2 erlauben, wurde in diesem Projekt auf eine Verbindung mittels parallelem Kabel zurückgegriffen. Diese Variante funktioniert laut Manuals mit allen Gaisler Produkten, was die Entscheidung unterstreicht. Nachteilig wirkt dabei die etwas niedrigere Übertragungsrate beim Programmieren des Prozessors. In Anbetracht dessen, dass dieses nicht allzu häufig von Nöten ist und im Vergleich zur übrigen Entwicklungszeit eher als gering angesehen werden kann, ist dies hinsichtlich des Vorteils, auch mit anderen Prozessoren kommunizieren zu können, vernachlässigbar.

27 http://www.xilinx.com/

Hardwareseitig wird also für eine Inbetriebnahme lediglich das mit dem Entwicklungsboard mitgelieferte parallele Schnittstellenkabel benötigt (Parallel Cable III). Auf Seite des Entwicklungsboardes ist dieses mit der JTAG Schnittstelle zu verbinden, auf PC-Seite wird eine vorhandene, freie parallele Schnittstelle benötigt.

Um nach Programmierung des FPGAs mit dem aktivierten Prozessor kommunizieren zu können, wurde in diesem Projekt auf Kommunikation mittels seriellem Schnittstellenkabel zurückgegriffen. Im Lieferumfang der Pender-Hardware ist ein passendes Kabel inkludiert. Dieses ist auf Seite des Entwicklungsboards mit der ersten Schnittstelle (dcom0) zu verbinden, auf PC-Seite ist dieses mit einer freien COM-Schnittstelle zu verbinden.

Softwareseitig sind auf dem Windows-PC für den Testaufbau zwei Programmprodukte zu installieren:

Die Installation des Xilinx-Paketes erfolgt durch ein Windows-typisches Installationsprogramm, die benötigte Lizenz lag im vorliegenden Projekt am Institut zur Verwendung auf.

Für das Aufspielen des Betriebssystems wird GRMON[28] von Gaisler Research benötigt. Für die Bedürfnisse beim SCPU Projekt reichte die Evaluierungsversion (`grmon-eval`). Die Installation erfolgt erfreulich einfach durch simples Entpacken des heruntergeladenen Softwarepaketes. Für eine einfachere Handhabe empfiehlt es sich, das eigentliche Programm `grmon-eval` in den Systempfad aufzunehmen. Später kann so `grmon-eval` direkt im Verzeichnis mit dem Betriebssystemimage gestartet werden. Dies erspart die Angabe des vollen Pfades zu dem zu ladenden Image.

4.1.3 Linux Entwicklungssystem

Die zweite PC-Plattform dient dem Erstellen der Betriebssystemabbilder und der Kommunikation mit dem laufenden Zielsystem. Grundsätzlich könnte die Betriebssystemerstellung und auch die Kommunikation über den Windows-PC

28 http://www.gaisler.com/

erfolgen. Dies erfordert jedoch eine vergleichsweise umständlich zu konfigurierende Cygwin-Umgebung[29] zur Konfiguration und Kompilation des Betriebssystems SnapGear. Überdies benötigt man zur Kommunikation mit dem laufenden Zielsystem eine weitere COM Schnittstelle. Da, wie so oft bei unspezialiserter Standard PC Hardware, auf dem Windows-PC nur eine serielle Schnittstelle zur Verfügung steht, wäre der Einbau einer zusätzlichen Einschubkarte notwendig gewesen. In Anbetracht dieser Umstände, erschien es einfacher einen zweiten PC zur Verfügung zu stellen. Ausgestattet mit einem Pentium IV bei 2 GHz Prozessortakt und lediglich 256 MByte Hauptspeicher, kommt es sehr gelegen, dass Linux - selbst mit grafischer Oberfläche - recht sparsam konfiguriert werden kann.

Im Testaufbau kommt die Linuxdistribution Ubuntu in der Version 6.10 zum Einsatz. Um dem Umfang dieser Arbeit nicht zu sprengen, wird auf die Installation und Konfiguration des Basissystems nicht näher eingegangen. Im Weiteren wird ein fertig installiertes System vorausgesetzt, wobei die verwendete Distribution, insbesondere aber die verwendete Version keine maßgebliche Rolle spielen sollte.

Hardwareseitig wird für eine Reproduktion der Testergebnisse lediglich eine freie serielle Schnittstelle benötigt. Diese ist durch ein entsprechendes serielles Schnittstellenkabel mit der ersten seriellen Schnittstelle am Entwicklungsboard zu verbinden.

Softwareseitig wird für die Kommunikation ein Terminalemulationsprogramm benötigt. Im Testaufbau wurde dazu minicom[30] verwendet. Die Installation gestaltet sich dank verfügbarer Ubuntu-Pakete denkbar einfach:

```
$ sudo apt-get install minicom
```

Installation der Terminalemulation minicom

Die Installation aller weiteren benötigten Softwarepakete wird an gegebener Stelle beschrieben.

29 http://www.cygwin.com/
30 http://alioth.debian.org/projects/minicom/

4.2 Leon2

Der Leon Prozessorkern ist ein synthetisierbarer 32 Bit Prozessorkern auf Basis der SparcV8[31] Prozessorarchitektur. Mittlerweile wurden mehrere Versionen von Leon entwickelt. Leon1 ist eine Entwicklung der „European Space Agency“ (ESA) aus dem Jahre 2000 mit der der Grundstein der Leon Prozessorfamilie gelegt wurde. Leon2 wurde im Auftrag der ESA entwickelt, was zur Gründung von Gaisler Research im Jahre 2003 führte. Leon2 verfügt wie sein Vorgänger über eine fünfstufige Prozessorpipeline. Der Nachfolger Leon3 wurde als komplette Neuentwicklung konzipiert und wiederum von Gaisler Research im Jahre 2004 umgesetzt. Leon3 ist Teil der umfangreichen „IP Core“ Bibliothek mit dem Namen GRLIB. Er verfügt im Gegensatz zu Leon2 über eine tiefere Prozessorpipeline und Unterstützung für Multiprozessorsysteme. Leon3 wird überdies auch in einer fehlertoleranten Version angeboten [Gaisler, 2003].

Alle Prozessorkerne werden in Form von VHDL-Quellcode zum herunterladen angeboten. Der Quellcode ist Großteils unter GPL[32] gestellt, was die Verwendung und Adaptierung der Prozessorkerne in der Forschung vor allem Aufgrund fehlender Lizenzzahlungen erleichtert. Für Leon3 existiert allerdings auch eine kommerzielle Version, diese Umfasst neben den Funktionalitäten der GPL-Version zusätzliche Treiber, beispielsweise für USB Unterstützung.

Im diskutierten Projekt wurde Leon2 verwendet, dies ist vor allem durch die bei Projektstart bessere Unterstützung für Leon2 durch Gaisler Research begründet.

Aus heutiger Sicht verliert dieser Umstand immer mehr an Bedeutung, so änderte sich diese Tatsache im Zuge des Projektfortschrittes sogar ins Gegenteil. Mit Jahresbeginn 2007 war eine deutliche Prioritätsverschiebung seitens Gaisler Research in Richtung Leon3 spürbar. So wurden alle

31 http://www.sparc.org/
32 GPL ... **G**eneral **P**ublic **L**icense, siehe: http://www.gnu.de/documents/gpl.de.html

aktualisierten Handbücher für Leon3 ausgelegt. Überdies scheint der Testaufwand für den Leon2 deutlich gesunken zu sein.

Eine aktueller Beitrag vom 26.07.2007 von Jiri Gaisler im leon_sparc Forum unterstreicht diesen Umstand:

> *„I have said this many times before: leon2 is no longer maintained or supported. It is very likely that the test software will fail when compiled with newer gcc versions and/or different compile flags than it was designed for.“*

An einigen Stellen des Projektes war ein Vorankommen nur nach Rückfrage bzw. Fehlerbericht über die einschlägigen Foren möglich. Erfreulicherweise waren aber die Reaktionen des Entwicklungsteams um Jiri Gaisler durchwegs positiv. In allen Fällen, die an späterer Stelle noch näher beschrieben werden, wurde kompetente Hilfe in Form von Quellcodepatches binnen weniger Werktage bereit gestellt.

Da der verwendete Leon2 Prozessorkern der SparcV8 Prozessorarchitektur-spezifikation genügt, wird im Folgenden kurz auf diese eingegangen:

4.2.1 SPARC V8

SPARC steht für **S**calable **P**rocessor **Arc**hitecture und ist eine RISC[33] Mikroprozessor *„instruction set architecture“*. Ursprünglich von Sun Microsystems im Jahre 1985 aus den RISC I & II Designs der Berkeley Universität in Kalifornien entwickelt, findet sie sich heute in vielen Prozessoren, allen voran in jenen von Sun und Motorola [Sparc, 1991].

Die Entstehungsgeschichte der Sparc Architektur zieht sich mittlerweile über drei Revisionen. Die erste Version – SparcV7 - wurde 1986 als 32 Bit Architektur publiziert. 1990 folgte die erweiterte Sparc Architektur namens SparcV8. 1993 folgte die 64 Bit Variante SparcV9 deren aktuellen Vertreter unter dem Namen UltraSparc zu finden sind. SparcV8 wurde schließlich 1994

33 RISC ... **R**educed **I**nstruction **S**et **C**omputer

als 32 Bit Mikroprozessorarchitektur unter dem Kürzel IEEE 1754-1994 von IEEE standardisiert.

Im Folgenden wird ein grober Überblick über die Hauptmerkmale der SparcV8 Architektur gegeben [Sparc, 1991]:

- linearer 32 Bit breiter Adressraum
- einfaches, uniformes Befehlsformat, alle Befehle sind 32 Bit breit und in 32 Bit Schritten im Speicher angeordnet. Es gibt lediglich drei unterschiedliche Befehlsformate, wobei opcode und Registeradressen immer in selber Reihenfolge vorkommen
- wenige Adressierungsmodi: eine Speicheradresse wird durch „Register + Register“ oder „Register + Konstante“ angegeben
- ein großes Registerfile, bestehend aus 8 globalen Registern und 2 bis 32 „24 register windows“
- ein eigenes floating point Registerfile.

Besondere Beachtung kann den *„register windows“* geschenkt werden. Neben den acht globalen Registern hat jede Instruktion Zugriff auf Register innerhalb eines *„register windows“* korrespondierend zum *„current window pointer“* (CWP). Abbildung 4-5 zeigt den strukturellen Aufbau und die Anordnung dieser *„register windows“*.

Jedes dieser Fenster besteht aus insgesamt drei mal acht Registern. Jeweils acht Register dienen als Eingaberegister, lokale Register und Ausgaberegister. Die acht Eingaberegister eines Fensters überlappen sich mit den acht Ausgaberegister des vorhergehenden Fensters. Analog dazu überlappen sich die acht Ausgaberegister mit den acht Eingaberegister des folgenden Fensters.

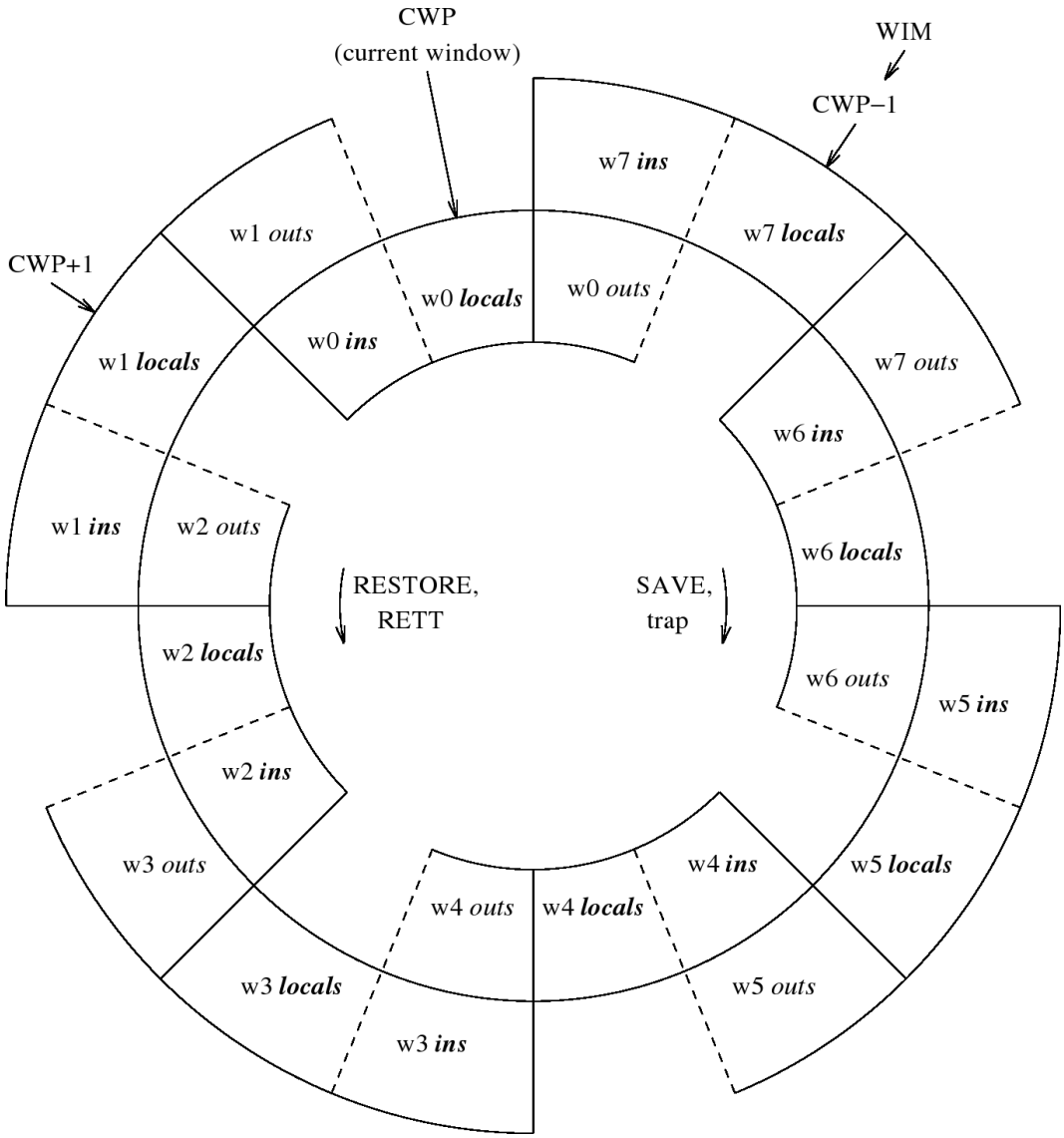

Abbildung 4-5: Sparc V8 register windows
Bildquelle: [Sparc, 1991]

Diese Art der Registerverwaltung hat vor allem den Vorteil, eine Parameterübergabe möglichst performant realisieren zu können. Durch simples De- bzw. Inkrementieren des CWP kann – in gewissen Grenzen - ohne abspeichern bzw. laden der Registerinhalte ein Contextswitch vollzogen werden. Die Parameterübergabe findet dabei über die sich überlappenden In- bzw. Outregister statt. Mitunter deshalb findet man derartige Registerverwaltung auch in anderen Prozessoren wie dem AMD 29000 oder dem Intel i960 [AMD, 1995], [Intel, 1998].

Die SparcV8 Architektur erlaubt eine vergleichsweise freie Konfiguration dieser *„register windows“*. Je nach Implementierung können 2 bis 32 dieser sich überlappenden Registerfenster verwendet werden. Dies entspricht einer Gesamtanzahl von 40 respektive 520 Registern [Sparc, 1991].

Entgegen dieser Angabe in der Dokumentation, wurde SCPU für den ersten Prototyp mit nur einem Fenster konfiguriert. Die Vorteile dieser Technik bleiben damit zwar ungenutzt, hinsichtlich einer einfacheren Fehlersuche, vor allem bei der portable.NET Portierung, erscheint dies allerdings tolerabel.

4.2.2 SCPU

Secure CPU (SCPU) ist eine auf Leon2 und damit auf SparcV8 basierende Hardwarearchitektur, die es sich zum Ziel setzt, *„bound checks“* oder ähnliche Vergleiche durch Hardwareunterstützung abzuarbeiten.

Das Projekt SCPU wurde an der TU Graz am Institut für Technische Informatik unter der Leitung von Ao.Univ.-Prof. Eugen Brenner geführt. Nach theoretischer Aufarbeitung der Thematik und Simulation des Konzeptes wurde beginnend mit 2006 unter der technischen Leitung von Dr. Michael Grasser SCPU bis zur heutigen Version entwickelt. Grasser beschreibt in [Grasser, 2007] Intention und Werdegang des Projektes. Einleitend wird der Grundgedanke folgend beschrieben:

> *Die Grundidee der Forschung war jedes Prozessorregister mit zwei zusätzlichen Registern zu erweitern, welche den jeweils niedrigst- und höchstzulässigen Wert enthalten. Aufgrund dieser Daten können sowohl Wert- als auch Referenztypen erfolgreich auf die Einhaltung der Grenzen überprüft werden. Diese Methode der Überprüfung wird im Englischen „Bound Checking“ genannt.*

Hauptaugenmerk der Anwendung dieser Prozessorerweiterung wird in SCPU auf sicherheitsrelevante Belange gelegt. Damit reiht sich diese Arbeit zu vielen vergleichbaren Aufarbeitungen des Sicherheitsaspektes im Kontext von „Bound Checks“ (siehe auch Kapitel 2 Vergleichbare Arbeiten). SCPU ist

allerdings im Gegensatz zu vielen anderen Implementierungen stark auf eine Hardwareerweiterung ausgerichtet.

Dies unterstreicht den Gedanken, dass Wertebereichsüberprüfungen als integraler Bestandteil von „sicherer“, moderner Software gesehen werden sollten. Unter anderem soll durch eine derartige Herangehensweise impliziten Sicherheitsüberprüfungen Vorschub geleistet werden. Zusammengefasst ergibt sich für die damit angeregte Überlegung:

Um den negativ wirkenden Argumenten „Geschwindigkeitseinbuße“ und „Quellcodegröße“ entgegen zu treten, wird die Hardware für die Aufgabe Wertebereichsüberprüfung optimiert. Damit ausgestattete Systeme sollen durch das Diktat des konsequenten, unumgänglichen Einsatz dieser Hardware optimalerweise nur mit passender Software betrieben werden können und damit sicherer im Bezug auf *„Buffer Overflows“* oder ähnlichen Auswirkungen sein.

Der im Zuge der Projektarbeit anfallende Implementierungsaufwand zur Modifikation der Hardware wurde auf insgesamt vier Projektmitarbeiter aufgeteilt. Konzeption und Design wurde im Beisein aller Beteiligter durchgeführt. Dadurch konnte vor allem die Schnittstelle zur Software überlegt gestaltet werden.

Die Modifikationen am Leon2 Prozessor betreffen zwei Bereiche. Eine zusätzliche Komparatoreinheit mit erweitertem Registerfile übernimmt die eigentliche Wertebereichsüberprüfung. Modifikationen am Trap System lösen im Fehlerfall ein entsprechendes Signal aus.

Die im Vergleich zu den gezeigten ähnlichen Arbeiten weitreichenste Modifikation betrifft Erweiterungen im Registerfile. Grundidee ist, jedes Register mit zwei weiteren Registern gleicher Breite zu versehen. Diese Register sollen obere und untere Grenze halten. Dieser Hardwareaufwand wird insbesondere dadurch gerechtfertigt, dass damit keine Register für die Wertebereichsüberprüfung belegt werden, lediglich der eigentliche Wert (z.B. Adresse) belegt ein „normales“ Register. Dies liegt aber in der Natur der Sache, schließlich will damit gearbeitet werden.

Eine weitere Idee war, die Vergleichseinheit grundsätzlich jeder Operation zur Verfügung zu stellen. So soll es möglich sein, auch eine Addition „normaler“ Integerwerte überprüfen zu können, ohne explizite Instruktionen dafür vorzusehen. Abbildung 4-6 zeigt die modifizierte Prozessorpipeline von SCPU.

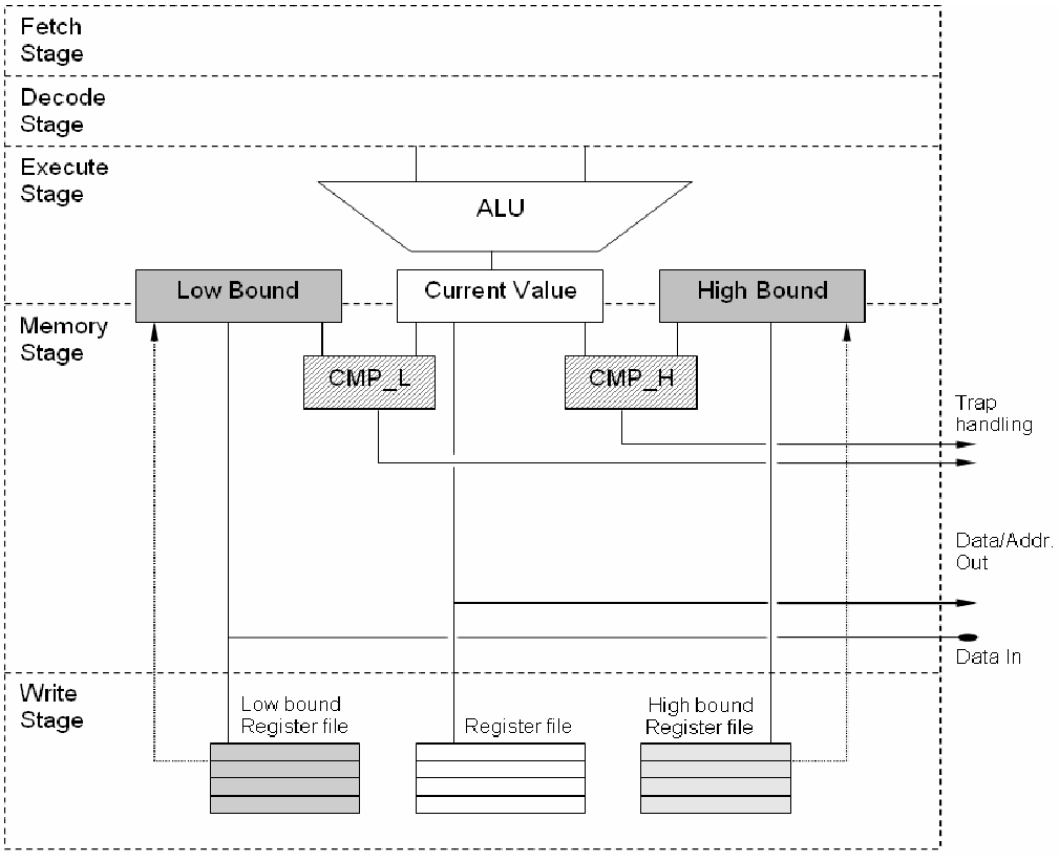

Abbildung 4-6: SCPU Pipeline
Bildquelle: [Grasser, 2007]

Um diese Erweiterungen benutzen zu können, wurde auch das *„instruction set“* von Leon2 um zwei Instruktionen erweitert. `LLB` steht für *„load low bound“* und dient dem Laden der unteren Grenze zu einem Register. Analog dazu dient `LHB` dem Laden der oberen Grenze zu einem Register. Das Befehlsformat wurde an bestehende Ladeinstruktionen angelehnt. Die Befehle nehmen jeweils zwei Parameter, das Zielregister (bzw. Zwillingsregister für obere oder untere Grenze) und das Quellregister, welches die Adresse mit oberer bzw. unterer Grenze hält.

Wurden einmal mittels dieser Befehle obere und untere Grenze für ein Register gesetzt, kommt in Folge die Vergleichseinheit zum Tragen. Diese arbeitet parallel zur normalen Ausführungspipeline. In dem vorliegenden Prototyp wurde diese Vergleichseinheit allerdings nur bei einer normalen Ladeoperation aktiviert. Wenn versucht wird, durch eine Ladeoperation einen Wert aus dem Speicher in ein Register zu laden, wird im Vorfeld die Speicheradresse auf Gültigkeit überprüft. Genau genommen entspricht dieses Verhalten nicht streng der Vorgabe. Das Zielregister bei einem Ladebefehl hält in Folge den Speicherwert. Die Grenzen beziehen sich allerdings nicht auf diesen Wert, sondern auf die Adresse, von der zu laden ist. Auf eine Ausdehnung des Konzeptes auf andere Instruktionen wurde für diesen Prototyp verzichtet.

Im Fehlerfall wird über den modifizierten Traphandler ein Signal gesendet. Aus heutiger Sicht ungünstig erscheint die Entscheidung, dafür ein bestehendes Signal zu überschreiben. Wird eine Grenzverletzung festgestellt, wird an das Betriebssystem das Signal SIGILL, welches ansonst eine nicht korrekte Instruktion signalisiert, gesendet. Dieser Umstand trägt mit Schuld daran, dass die Implementierung nicht die erwarteten Geschwindigkeitsvorteile bringt. An späterer Stelle wird noch näher auf diese Eigenschaft eingegangen.

Im Folgenden wird davon ausgegangen, dass die SCPU Version des Leon2 Quellcodes, also mit projektspezifischen Quellcodeänderungen, verwendet wird. Der Quellcode ist auf der beiliegenden CD zu finden.

4.2.3 Konfiguration und Synthetisierung von Leon2

In diesem Abschnitt wird die Konfiguration und Synthetisierung von Leon2 für das verwendete Entwicklungsboard beschrieben. Dieser Schritt erfordert naturgemäß eine installierte Synthetisierungssoftware. Unterstützt wird Leon2 von den Synthesetools Synopsys DC, Xilinx ISE und Synpicity Simplify. In diesem Projekt wurde Xilinx ISE 8.1i verwendet.

Die Konfiguration und Synthetisierung kann in einer Linux- wie auch in einer Cygwin Windows Umgebung erfolgen. In Anbetracht des Umstandes, dass in diesem Projekt eine Windowsplattform verwendet wurde, wird auch hier nur diese Variante beschrieben. Detailliertere Informationen können dem Manual zu Leon2 (siehe [Gaisler, 2004]) sowie der Projektdokumentation zu SCPU (siehe [Grasser, 2007]) entnommen werden.

Leon2 kann über vergleichsweise einfache Konfigurationsdateien für eine Vielzahl von Entwicklungshardware konfiguriert werden. Für einige populäre Boards, darunter auch das verwendete Pender GR-XC3S-1500, stellt Gaisler als Teil des Leon2 Quellcodes solche Dateien bereit.

Ungeachtet dessen ist der Zeitaufwand für Konfiguration und vor allem Compilation nicht unerheblich. Das Hardwareentwicklungsteam musste auf dem verwendeten Windows-PC mit Compilezeiten von bis zu einer Stunde leben. Wenn also keine triftigen Gründe bestehen, Leon2 selbst zu compilieren, kann dieser Schritt übersprungen werden und eine vorcompilierte Version der beiliegenden CD entnommen werden.

Zur Konfiguration von Leon2 bestehen zwei Möglichkeiten. Zum einen kann durch Abänderung der Konfigurationsdateivorlagen und anschließendem Aufruf von `make` der Konfigurationsprozess angestoßen werden. Die zweite Alternative besteht in der Konfiguration mittels grafischer Oberfläche. Da für eine Reproduktion der Ergebnisse keine Änderungen an Leon2 notwendig ist, wird an dieser Stelle nur die erste Möglichkeit beschrieben.

Um die Konfiguration für das verwendete Board GR-XC3S-1500 vorzunehmen, wird das Programm `make` benötigt. Dieses kann einfach über die Installationsroutine von Cygwin[34] installiert werden. Die Vorlagen der Konfigurationsdateien für verschiedenste Entwicklungsboards sind im Verzeichnis *leon2-1.0.32-xst/boards* der Leon2 Quellcodedistribution zu finden. Über den Parameter `BOARD` kann die zu verwendende Hardware angegeben werden. Der Parameterwert entspricht dabei dem Namen des Unterverzeichnisses von *leon2-1.0.32-xst/boards*, in dem die jeweilige

34 http://www.cygwin.com/

Konfigurationsdateien vorliegen. Ohne weitere Parameterangaben wird in diesem Verzeichnis eine Datei namens `config-default` erwartet. Um andere Konfigurationen zu verwenden, besteht die Möglichkeit über den Parameter `CONFIG` einen anderen Dateinamen zu übergeben.

Der eigentliche Aufruf zum Anstoßen der Konfiguration von Leon2 für das verwendete Board in der benutzten Konfiguration lautet folglich:

```
$ cd <LEON2_SOURCE>
$ make config BOARD=gr-xc3s1500
```

Konfiguration von LEON2

Mit erfolgreicher Abarbeitung dieses Schrittes wird eine VHDL Konfigurationsdatei im Verzeichnis `leon2-1.0.32-xst/leon` mit dem Namen `device.vhd` erzeugt.

Im Anschluss an die Konfiguration ist in ähnlicher Weise die Synthetisierung durchzuführen. Um die Synthetisierung mit der zuvor erzeugten Hardwarekonfiguration zu starten bedient man sich wiederum des Programmes `make`. Zur einfachen Kontrolle des Synthesevorganges empfiehlt es sich, die Ausgabe in eine Datei umzulenken. Diese kann damit einfach nach etwaigen Fehlern durchsucht werden. Wiederum im Hauptverzeichnis des Leon2 Quellcodes wird folgender Befehl abgesetzt:

```
$ make fpga > synthesis.log
```

Synthetisierung von Leon2

Um Konfiguration und Synthetisierung in einem Schritt auszuführen, besteht die Möglichkeit dem Programm `make` beide Ziele (engl. target) anzugeben:

```
$ make config fpga BOARD=gr-xc3s1500 > all.log
```

Konfiguration und Synthetisierung von Leon2

Nach erfolgreicher Abarbeitung von Konfiguration und Synthese stehen im Verzeichnis *leon2-1.0.32-xst/boards/gr-xc3s1500* entsprechende Dateien zur Programmierung des FPGAs bereit. Die Namen der Dateien entsprechen dem Muster: `leon*.II` bzw. `leon*.msk`.

4.2.4 Programmieren des FPGAs

In diesem Abschnitt wird die Programmierung des FPGAs beschrieben. Für diesen Schritt wird eine korrekte physikalische Verbindung des Windows-PCs mit dem Entwicklungsboard vorausgesetzt. Wie in Abschnitt „4.1.2 Windows Synthetisierungsplattform„ schon erwähnt, wird für die Programmierung die über „Parallel Cable III“ verbundene JTAG Schnittstelle verwendet. Abbildung 4-7 zeigt das verbundene Entwicklungsboard im Versuchsaufbau.

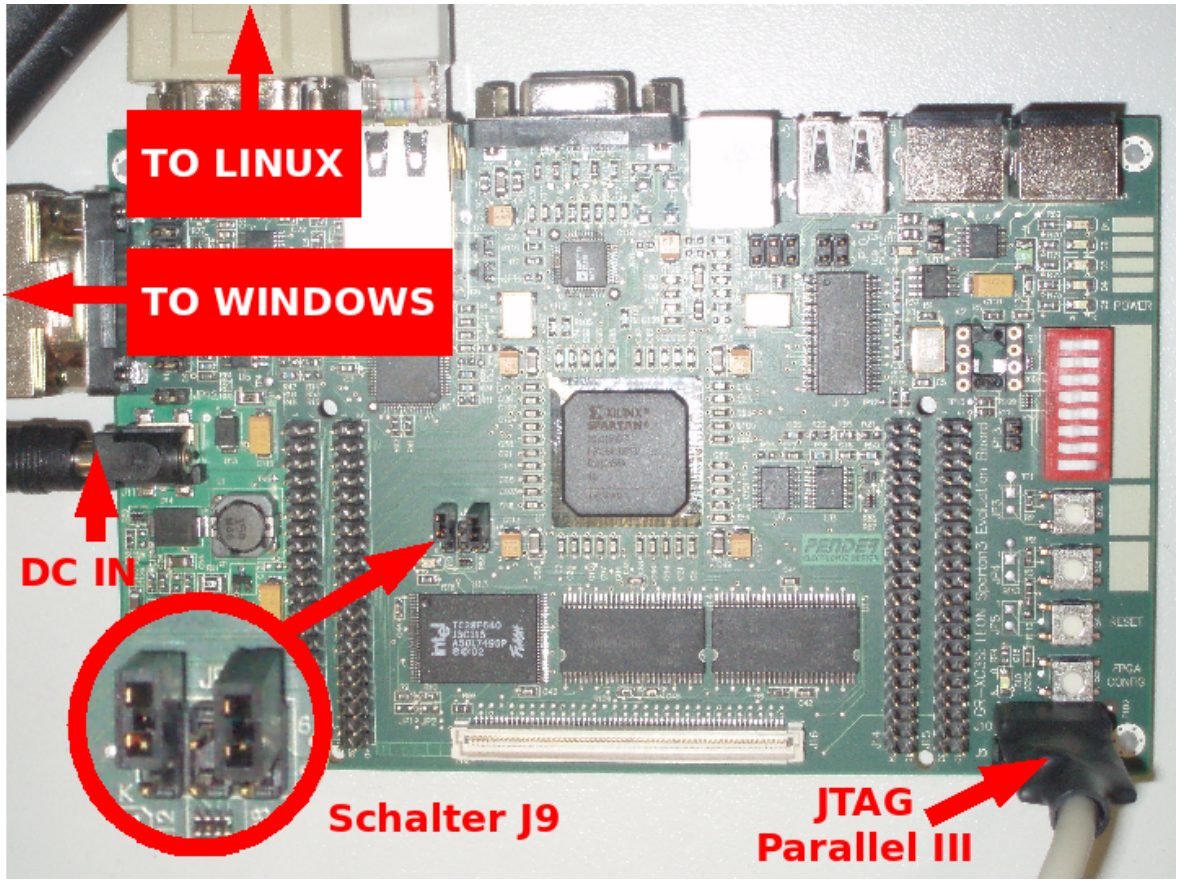

Abbildung 4-7: Pender GR-XC3S-1500 im Versuchsaufbau

Eine erste Verifikation der Verbindung kann erst nach korrekter Konfiguration der FPGA-Programmiervariante erfolgen. Die Programmierung kann grundsätzlich auf zweierlei Arten durchgeführt werden. Für eine erstmalige Programmierung des FPGAs, oder nach Änderung und erneuter Synthetisierung, geschieht diese über die JTAG Schnittstelle. Diese Methode wird im Handbuch (siehe [Pender, 2006]) auch *„JTAG Mode Configuration“* genannt. Wurde das Entwicklungsboard bereits einmal programmiert, kann diese FPGA Konfiguration bei Neustart oder Reset auch automatisch aus den vorhandenen FLASH Speichern geladen werden. Diese Variante wird im Handbuch „Master Serial Mode Configuration“ genannt.

Die jeweils verwendete Variante wird über Setzen der Schalter benannt mit J9 bestimmt. Wird Pin 1-2 und 5-6 mittels *„Jumper“* gesetzt, ist *„JTAG Mode Configuration“* aktiviert, andernfalls ist *„Master Serial Mode Configuration“* aktiv.

Für ein erstmaliges Programmieren ist die Entwicklungshardware also in *„JTAG Mode Configuration“* zu versetzen. Abbildung 4-7 zeigt im Detail „Schalter J9“ die entsprechenden Pins 1-2 und 5-6 geöffnet, für den gewünschten Betriebsmodus müssen diese geschlossen werden.

Nach erfolgter Konfiguration des Schalters ist das Entwicklungsboard über das mitgelieferte Netzgerät mit Spannung zu versorgen. Ein Status-LED am Board zeigt eine funktionierende Stromversorgung an.

Nun kann die Verbindung verifiziert werden und im Erfolgsfall die Programmierung des FGPAs vorgenommen werden. Dazu wird das Programm Xilinx Impact verwendet. Bei Programmstart kann die Aufforderung - ein vorhandenes Projekt zu öffnen - geschlossen werden. Durch Klick auf *„Boundary Scan“* im linken *„iMPACT Modes“*-Fenster kann in der oberen Befehlsleiste das Icon mit der Funktion *„scan JTAG chain“* betätigt werden. Dieser Vorgang durchsucht automatisch mögliche Anschlüsse nach bekannter Hardware. Wurde das Board korrekt angeschlossen, wird es als Pender GR-XC3S-1500 erkannt und die drei Hauptkomponenten werden symbolisch am Bildschirm dargestellt: zwei FLASH Speicher benannt mit xcf04s und xcf01s

und der FPGA mit Namen xc3s1500. Abbildung 4-8 zeigt die erkannten Module.

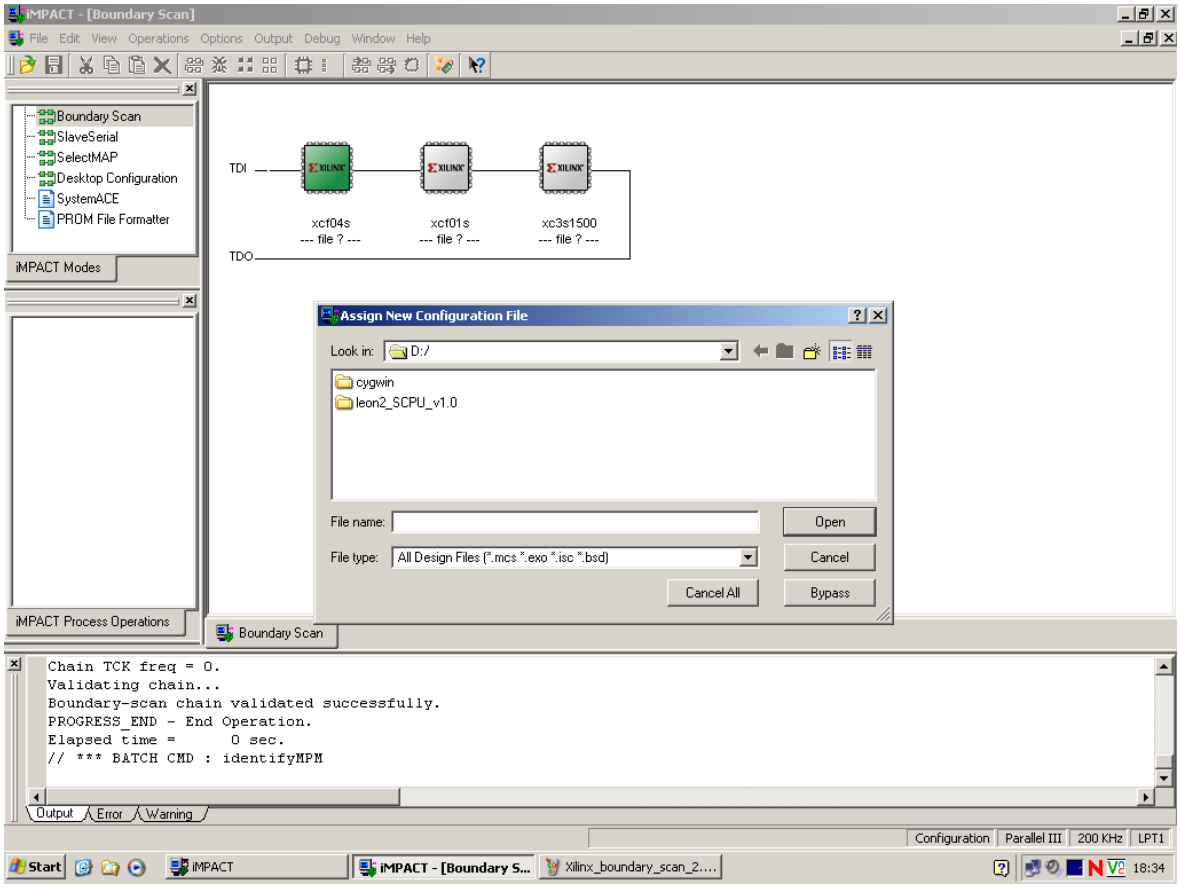

Abbildung 4-8: Xilinx iMPACT nach JTAG Initialisierung

Nun werden die im Syntheseschritt erstellten VHDL Konfigurationsdateien den jeweiligen Modulen zugeordnet. iMPACT fordert den Benutzer in drei aufeinander folgenden Dialogen automatisch auf, die Dateien anzugeben. Auf der beiliegenden CD-ROM finden sich die entsprechenden Dateien im Verzeichnis /leon2_SCPU_v1.0/boards/gr-xc3s1500/ mit dem Namensschema leon_xst_SCPU_*.*. Die Zuordnung muss in richtiger Reihenfolge erfolgen:

1. Modul: xcf104s -> Datei: leon_xst_SCPU_0.mcs
2. Modul: xcf101s -> Datei: leon_xst_SCPU_1.mcs
3. Modul: xc3s1500 -> Datei: leon_xst_SCPU.bit

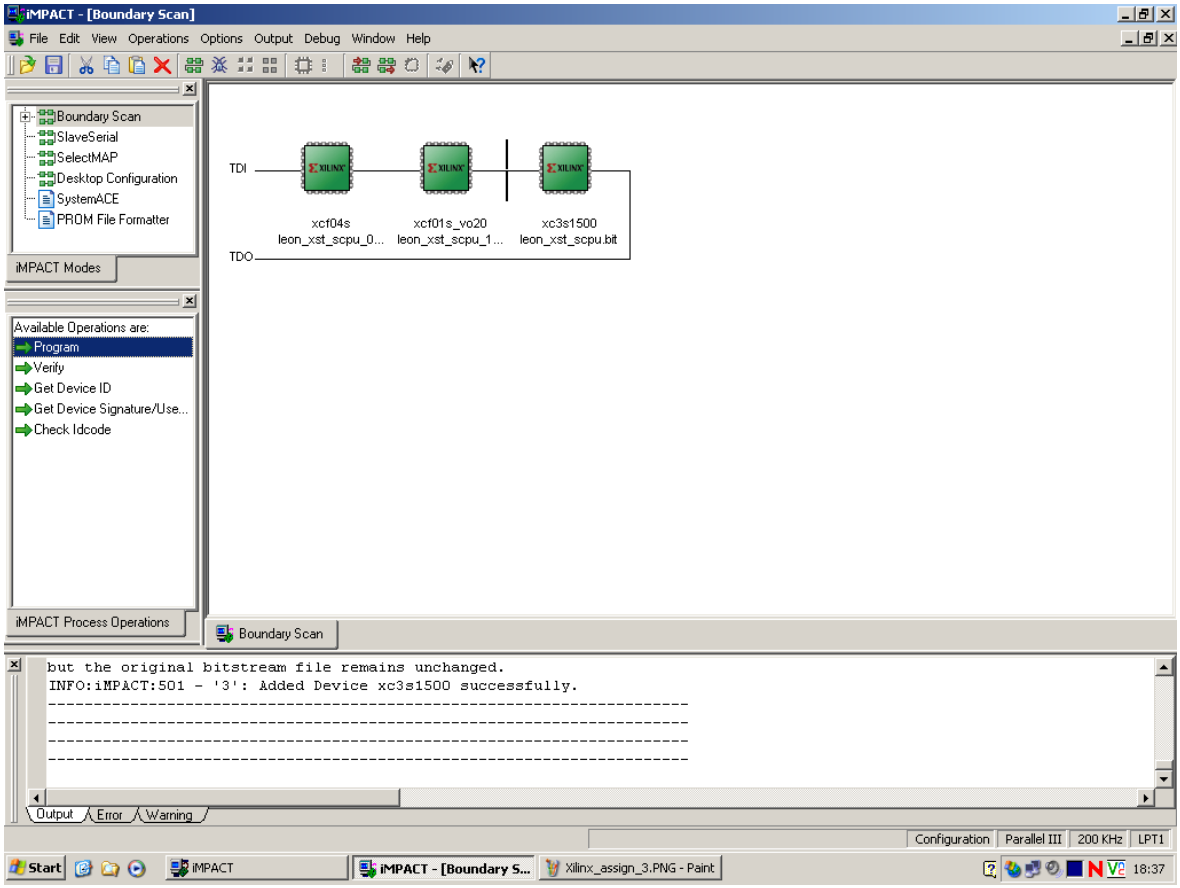

Abbildung 4-9: Xilinx iMPACT FPGA Programmierung

Nach erfolgter Zuordnung kann mit der Programmierung fortgefahren werden. Dazu müssen alle drei Module durch Halten der [Strg]-Taste und Anklicken der einzelnen Module ausgewählt werden. Abbildung 4-9 zeigt die selektierten Module vor Programmierung. Im linken Menü *„iMPACT Process Operations“* wird durch Betätigung von *„Program“* die Programmierung gestartet. Durch Auswahl von *„Erase Befor Programming“* wird sichergestellt, dass alle Speicherbereiche zuvor gelöscht werden. Der eigentliche Schreibvorgang geht verhältnismäßig schnell von statten, ein Fortschrittsbalken zeigt den Verlauf der Programmierung. Zusätzlich wird JTAG Aktivität auch am Board direkt durch blinkende Leuchtdioden angezeigt.

Um die erfolgreiche Programmierung zu verifizieren, wird das Programm `grmon` benötigt. Die Evaluierungsversion `grmon-eval` ist für die Bedürfnisse

des Projektes ausreichend und kann über die Webseite[35] von Gaisler Research bezogen werden. Die Installation von `grmon` besteht im Entpacken des Programmpakets in ein beliebiges Verzeichnis. Um aus der Windowskonsole `cmd` einfach auf `grmon` zugreifen zu können, empfiehlt es sich das Windowsverzeichnis des Programmpakets in den Pfad aufzunehmen.

Vor Verifikation der erfolgreichen Programmierung, ist sicherzustellen, dass das Entwicklungsboard mittels seriellem Schnittstellenkabel an den Windows-PC angeschlossen ist. In Abbildung 4-7 ist der entsprechende Anschluss mit *„TO WINDOWS“* markiert.

Laut Handbuch (siehe [Gaisler GRMON, 2007]) muss `grmon` mit dem Parameter `-nb` gestartet werden. Diese Option ermöglicht SnapGear Linux bei MMU[36] Systemen Traps zu empfangen. Andernfalls würde `grmon` die Ausführung beispielsweise beim ersten *„page fault“* stoppen.

Folgender Auszug zeigt die Bildschirmausgabe von `grmon-eval -nb`. `grmon` versucht dabei in der Standardeinstellung eine Kommunikation zum Entwicklungsboard über die erste serielle Schnittstelle `COM1` herzustellen. Soll dazu eine andere Schnittstelle verwendet werden, kann dies über den Parameter `-uart` angegeben werden. Der Parameterwert hängt dabei vom verwendeten Betriebssystem ab. Das Programm `grmon` verfügt über keine Hilfe. An dieser Stelle sei deshalb auf das Handbuch zu `grmon` verwiesen. Kapitel 3 beschreibt alle Programmoptionen für unterschiedliche Anwendung (siehe [Gaisler GRMON, 2007]).

Da `grmon` auch für die Kommunikation mit Leon3 Prozessoren herangezogen werden kann, wird nach Leon3-spezifischen Eigenschaften gesucht. Diese Suche schlägt korrekterweise fehl und `grmon` schaltet automatisch in den Leon2 Kompatibilitätsmodus. Die erkannte Prozessortaktfrequenz sollte jener entsprechen, die in der Leon2 Konfiguration angegeben wurde. Nach erfolgreicher Erstinitialisierung werden erkannte Prozessorkomponenten gelistet. `grmon` mündet nach erfolgreicher Initialisierung in einer eigenen

35 http://www.gaisler.com/
36 MMU – memory management unit

Konsole, aus der zu späterem Zeitpunkt die Betriebssystemabbilder geladen werden.

```
C:\>grmon-eval -nb
 GRMON LEON debug monitor v1.1.19a (evaluation version)
 Copyright (C) 2004,2005 Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to support@gaisler.com

 This evaluation version will expire on 28/6/2007
try open device //./com1
###opened device //./com1

 GRLIB plug&play not found, switching to LEON2 legacy mode

 initialising ............
 detected frequency:  50 Mhz

 Component                            Vendor
 LEON2 Memory Controller              European Space Agency
 LEON2 AHB Status & Failing Addr      European Space Agency
 LEON2 SPARC V8 processor             European Space Agency
 LEON2 Write Protection               European Space Agency
 LEON2 Configuration register         European Space Agency
 LEON2 Timer Unit                     European Space Agency
 LEON2 UART                           European Space Agency
 LEON2 UART                           European Space Agency
 LEON2 Interrupt Ctrl                 European Space Agency
 LEON2 I/O port                       European Space Agency
 AHB Debug UART                       Gaisler Research
 LEON2 Debug Support Unit             Gaisler Research
 Use command 'info sys' to print a detailed report of attached
 cores

grlib>
```

Verifikation der FPGA Programmierung mittels grmon-eval

Für einen ersten Test können durch Aufruf von `info sys` Prozessorspezifika ausgelesen werden. Alle inkludierten Prozessorteile werden in der getätigten Konfiguration detailliert gelistet. Dies umfasst den Prozessorkern selbst, Leons DSU und auch eventuell konfigurierte Netzwerkhardware inklusive der vergeben MAC Adresse.

Für die SCPU-Version sieht die Ausgabe von `info sys` wie folgt aus:

```
grlib> info sys
00.04:00f   European Space Agency  LEON2 Memory Controller (ver0)

   ahb: 00000000 - 20000000
   ahb: 20000000 - 40000000
   ahb: 40000000 - 80000000
   apb: 80000000 - 80000010
   8-bit prom @ 0x00000000
   32-bit sdram: 1 * 64 Mbyte @ 0x40000000, col 9, cas2, ref7.8us

01.04:017  European Space Agency  LEON2 AHB Status & Failing
           Addr (ver 0)
   apb: 8000000c - 80000014
02.04:002  European Space Agency  LEON2 SPARC V8 processor (ver 0)

   apb: 80000014 - 80000018
03.04:018  European Space Agency  LEON2 Write Protection (ver 0)
   apb: 8000001c - 80000020
04.04:008  European Space Agency  LEON2 Configuration register
           (ver0)
   apb: 80000024 - 80000028
   val: e877bf00
05.04:006  European Space Agency  LEON2 Timer Unit (ver 0)
   apb: 80000040 - 80000070
06.04:007  European Space Agency  LEON2 UART (ver 0)
   apb: 80000070 - 80000080
   baud rate 38400
07.04:007  European Space Agency  LEON2 UART (ver 0)
   apb: 80000080 - 80000090
   baud rate 38400
08.04:005  European Space Agency  LEON2 Interrupt Ctrl (ver 0)
   apb: 80000090 - 800000a0
09.04:009  European Space Agency  LEON2 I/O port (ver 0)
   apb: 800000a0 - 800000ac
0a.01:007  Gaisler Research  AHB Debug UART (ver 0)
   apb: 800000c0 - 800000d0
   baud rate 115200, ahb frequency 50.00
0b.01:002  Gaisler Research  LEON2 Debug Support Unit (ver 0)
   ahb: 90000000 - a0000000
   trace buffer 512 lines, stack pointer 0x43fffff0
   CPU#0 win 8, hwbp 2, V8 mul/div, srmmu, lddel 1
   icache 1 * 8 kbyte, 32 byte/line
   dcache 1 * 8 kbyte, 32 byte/line
grlib>
```

grmon Ausgabe von info sys

Zu einem späteren Zeitpunkt werden diese Informationen noch benötigt. Bei der Konfiguration des Betriebssystems SnapGear muss beispielsweise die konfigurierte Übertragungsrate der seriellen Schnittstellen bekannt sein. In diesem Projekt beträgt die `baud rate` für beide `UARTs` jeweils 38400.

4.3 Vorbereitung von portable.NET

Bevor die Linuxdistribution SnapGear konfiguriert und kompiliert werden kann, wird in diesem Abschnitt die eingesetzte Laufzeitumgebung portable.NET behandelt. Ein Überblick über die Plattform wurde schon in Kapitel 3.3.4 gegeben, an dieser Stelle werden die getätigten Modifikationen beschrieben und der abgeänderte Quelltext für eine Einbindung in SnapGear vorbereitet. Im darauf folgenden Abschnitt wird im Zuge der Konfiguration von SnapGear die tatsächliche Integration von portable.NET getätigt.

Folgende Anleitung gleicht in weiten Teilen Kapitel 4.5.3 auf Seite 94, in dem die Installation von portable.NET am Linuxrechner beschrieben ist. Um eine bestmögliche Reproduzierbarkeit zu gewährleisten, werden auch an dieser Stelle alle notwendigen Angaben gemacht.

Anstelle aktuelle Quellcodebasis aus dem CVS[37] Repository von gnu.org wird zur Reproduktion die abgeänderte Quellcodeversion von der beiliegenden CD benötigt. In der Installationsdokumentation[38] von portable.NET ist die Konfiguration grob beschrieben, hier wird zwecks einfacher Reproduzierbarkeit eine detailliertere Installationsanleitung gegeben.

Bedingung für die Konfiguration von portable.NET ist das Programm `treecc`[39] welches eigentlich nicht Teil der portable.NET Laufzeitumgebung ist, sondern ein Hilfsmittel zur Entwicklung von Compilern und anderen Programmen die extensiv Syntaxbäume verarbeiten. `treecc` ist nur während der Compilierung von `pnet` von Nöten. Die Installation von `treecc` gestaltet sich vergleichsweise einfach durch Einspielen des entsprechenden Ubuntu Paketes mittels `apt-get`:

```
$ sudo apt-get install treecc
```

Installation von treecc

37 CVS ... **C**oncurrent **V**ersions **S**ystem
38 http://dotgnu.org/pnet-install.html
39 http://www.southern-storm.com.au/treecc.html

Nach erfolgreicher Installation steht das Programm `treecc` im Pfad zur Verfügung. In den nachfolgenden Schritten wird dieses von den Konfigurationsprogrammen von `pnet` benötigt.

Da portable.NET für das Zielsystem compiliert werden soll, muss nun der an die SPARC Architektur angepasste gcc installiert werden. Dieser Arbeitsschritt wird auch von SnapGear benötigt, weshalb dieser Schritt in Kapitel 4.4.1 auf Seite 81 detailliert beschrieben wird. Im Folgenden wird davon ausgegangen, dass dieser Schritt vollzogen wurde.

Zur Übersetzung von `pnet` sind überdies einige wenige Systemvoraussetzungen zu schaffen:

```
$ sudo apt-get install build-essentials bison flex automake
                       autoconf
```

Installation der Übersetzungswerkzeuge für pnet@SILC

Nach erfolgreicher Installation obiger Programme kann die modifizierte Version von portable.NET von der CD kopiert werden. Um eine klare Trennung zwischen portable.NET für das Entwicklungssystem und der Variante für SILC zu schaffen, empfiehlt es sich zuvor eine entsprechende Verzeichnisstruktur zu schaffen:

```
$ mkdir portable.NET.SILC
$ cd portable.NET.SILC
$ tar -xzvf /cdrom/src/portable.net.SILC.tar.gz .
```

portable.NET für SILC kopieren

Zur Abarbeitung der Demonstrationsprogramme wird lediglich der Interpreter `pnet` benötigt. Der Hilfe des `./configure` Skriptes ist die Verwendung der relevanten Konfigurationsparameter zu entnehmen:

```
$ ./configure –help
`configure' configures this package to adapt to many kinds of
systems.
Usage: ./configure [OPTION]... [VAR=VALUE]...
...
Installation directories:
--prefix=PREFIX   install architecture-independent files in PREFIX
...
System types:
  --build=BUILD     configure for building on BUILD [guessed]
  --host=HOST       cross-compile to build programs to run on
                      HOST [BUILD]
  --target=TARGET   configure for building compilers for
                      TARGET [HOST]
Optional Features:
...
--disable-tools   Remove the developer tools from the build
...
Optional Packages:
...
--without-libffi  disable libffi support
...
--without-libgc   disable libgc support
```

Hilfe zu ./configure für pnet@SILC

Hinsichtlich der eingeschränkten Software- und Hardwarevoraussetzungen am Zielsystem SCPU wird pnet weitestgehend einfach und klein gehalten. Dies wird über obig beschriebene Konfigurationsoptionen `--disable-tools`, `--without-libffi` und `--without-libgc` erreicht. Überdies werden angaben zum Zielsystem gemacht.

```
$ ./configure --prefix=~/pnet_SCPU --build=i686-pc-linux-gnu
        --host=sparc-linux --target=sparc-linux
        --disable-tools --without-libffi --without-libgc
```

Aufruf von ./configure für pnet@SILC

Die unter Linux obligatorischen Folgebefehle `make` und `make install` werden vorerst nicht ausgeführt, Ziel war lediglich portable.NET für den Compileprozess gemeinsam mit SnapGear vorzubereiten.

4.3.1 Modifikationen

Die getätigten Modifikationen an portable.NET betreffen zum einen Anpassungen an den Makefiles zur nahtlosen Integration in SnapGear, zum anderen natürlich die Codemodifikationen um die SCPU spezifischen Erweiterungen zu nutzen.

Alle einzelnen Codemodifikationen im Detail zu beschreiben, würde den Umfang dieser Arbeit sprengen. Aus diesem Grund wird hier nur auf die wesentlichen Punkte eingegangen.

Alle wesentlichen Änderungen wurden in der so genannten „*engine*“ durchgeführt. Wie in dem vorangegangenen Kapitel über portable.NET beschrieben, interpretiert diese Laufzeitumgebung nicht direkt CIL sondern CVM. Dieser Teil des Interpreters wird „*engine*“ genannt. In diesem, auf wenige Dateien verteilte, Quellcode finden sich u.A. Interpretationsregeln für Ladeoperationen, d.h. der eigentliche Arbeitscode, der beim Laden eines Speicherbereichs ausgeführt wird. Wird in einem CIL Programm beispielsweise ein Arrayzugriff getätigt, erfolgt in der „*engine*“ die eigentliche Abarbeitung dieser Operation inklusive der notwendigen Wertebereichsüberprüfungen. Dies bedeutet auch, dass auf CIL Ebene keine Operationen zum „*bound checking*“ enthalten sind, diese werden vom Interpreter an geeigneter Stelle ausgeführt.

Portable.NET ist in C geschrieben was die hardwarenahe Anpassung überhaupt erst ermöglicht. Da keine Assembly Befehle für LLB und LHB existieren, wurden diese speziellen SCPU Instruktionen über den Umweg „*inline assembly*“ eingebaut. „*inline assembly*“ ist eine Möglichkeit Assemblycode direkt in C Code einzubetten. Damit stehen alle Assemblybefehle der Zielarchitektur zur Verfügung, somit auch der Befehl `.long`. `.long` erlaubt das Bitweise angeben einer Prozessorinstruktion, also nicht als Assemblybefehl. So konnten die unbekannten Befehle aufgerufen werden. Das folgende Beispiel sind Teil eines Testprogramms zur Verifikation von SCPU, die beiden `.long` Befehle stehen für `LLB` und `LHB`:

```
// insert load instruction for high and low bound
  __asm__ ("ld %0, %%g4\n\t"
           "ld %1, %%g5\n\t"
           ".long (3<<30)|(8<<25)| (8<<19)|(4<<14)|(0)\n\t"
   /* 3...FMT3, 8... output register 1,  8...opCode for LLB,
      4...global register 4, 0...no offset*/
           ".long (3<<30)|(8<<25)|(24<<19)|(5<<14)|(0)"
    /* 3...FMT3, 8... output register 1, 24...opCode for LHB,
      5...global register 4, 0...no offset*/
           :/* no output registers */
           :"g"(array_start_pointer), "g"(array_end_pointer)
           :"memory"
          );
```

Beispielhafte Inlnie Assembly Anweisung

Für eine nachfolgende Leseoperation, die wieder in reinem C codiert ist, stellt dieser Block die notwendige Funktionalität um obere und untere Grenze in die entsprechenden Zwillingsregister (in diesem Fall zu R8) zu schreiben.

Um im Fehlerfall auf den Trap reagieren zu können, muss das vom Betriebssystem empfangene Signal aufgefangen und verarbeitet werden. Diese Funktionalität wurde über einen eigene Signalhandler-Funktion realisiert. Diese wird bei Start der Laufzeitumgebung definiert und an gegebener Stelle aktiviert. Da, wie bereits erwähnt, leider ein bestehendes Signal (SIGILL) überschrieben wurde und vom Traphandler gesendet wird, muss für eine ordnungsgemäße Funktionalität der *Signalhandler* immer wieder zurückgesetzt werden, um ein etwaiges echtes Auftreten dieses Signals nicht zu verpassen. In den Testläufen wurde allerdings auf dieses korrekte Vorgehen teilweise verzichtet um mögliche Geschwindigkeitssteigerungen ausloten zu können.

Die eigentliche Verarbeitung des Signals zu einer *„Exception“* im Sinne des CIL Programmes geschieht allerdings nicht im *Signalhandler*. Dieser sollte tunlichst kurz gehalten werden um das Betriebssystem nicht zu lange in diesem Kontext zu halten. Ein alsbaldiges terminieren dieser Funktionen wird von vielen Seiten stark empfohlen um schwer auffindbare Probleme zu vermeiden.

Anstelle einer direkten Abarbeitung im *Signalhandler* wird in diesem nur eine globale Variable gesetzt, die einen Wertüberlauf signalisiert. Dieser Wert wurde in Folge überprüft und im Fehlerfall der Ausnahmecode angesprungen.

Auch diese Vorgehensweise ist einer performanten Laufzeitumgebung nicht dienlich, schließlich galt es in diesem Projekt, eben arithmetische Vergleiche einzusparen.

Eine weitere Änderung ist in den Makefiles nötig. Den Hilfedateien von SnapGear kann entnommen werden, dass für eigene Programme spezielle Ziele für `make` eingerichtet werden müssen. Nach Compilation von SnapGear wird automatisch das `make`-Ziel `romfs` aufgerufen. Mit diesem Schritt werden alle compilierten Programme, das Betriebssystem selbst und Bibliotheken in eine einzige Datei gepackt, die, auf den Leon2 geladen, zur Ausführung gebracht werden können.

Im Folgenden wird davon ausgegangen, dass der modifizierte Quellcode von portable.NET von der CD verwendet wird.

4.4 SnapGear

Gaisler bietet ergänzend zu den Prozessorkernen Leon2 und Leon3 das Betriebssystem SnapGear inklusive Leon-spezifischer Modifikationen auf deren Internetseite[40] zum Herunterladen an.

SnapGear[41] ist eine Linuxdistribution speziell für eingebettete Systeme und bietet Unterstützung für eine Vielzahl von Plattformen unterschiedlichster Ausprägung. SnapGear kann mit Linuxkernel der Version 2.4 wie auch mit Version 2.6 konfiguriert werden, was sowohl den Einsatz auf Systemen ohne MMU wie auch mit MMU zulässt. SnapGear bietet überdies verschiedene libc-Bibliotheken zur Auswahl an. Die kompakte Bibliothek uClibc erleichtern den Einsatz auf speicherarmen Systemen, die Standardbibliothek glibc bietet bestmögliche Kompatibilität zu bestehenden Linuxapplikationen. Durch eine

40 http://www.gaisler.com/
41 http://www.snapgear.org/

durchgehende, homogenen Entwicklungskette (engl.: toolchain) basierend auf der Compilerzusammenstellung gcc ist die Konfiguration und Kompilation (cross compilation) auf herkömmlichen Linuxsystemen bequem möglich.

Die Leon-Version von SnapGear erfährt von Gaisler Research in unregelmäßigen Abständen Korrekturen und Anpassungen an aktualisierte Linuxkernelversionen. Im Zuge des Projektes wurden beginnend mit SnapGear 0.27 bis zu Version 0.33 gearbeitet. Wie bereits erwähnt, sank im Zuge des Projektverlaufes die Unterstützung des Leon2 Prozessors zugunsten Leon3. Dies machte sich insbesondere bei aktualisierten SnapGear-Versionen bemerkbar. Während sich Stabilität und Qualität der einzelnen Versionen im Projekteinsatz bis Version 0.29 allem Anschein nach verbesserte, war ein stabiler Betrieb ab 0.31 ohne weiteres nicht mehr möglich. Da ab 0.31 keine für diesen Projekteinsatz relevanten Verbesserungen eingeflossen sind, wurde schließlich diese Version verwendet.

4.4.1 Installation von sparc-linux GCC

Für das Compilieren von SnapGear wird eine für SPARC Systeme adaptierte Variante der GCC in Version 3.2.2 verwendet. Diese ist über die Webseite von Gaisler Research zu beziehen und liegt als gepacktes vorcompiliertes Programmpaket vor.

Die Installation am Linux Entwicklungssystem erfolgt laut Handbuch (siehe [Gaisler, 2007]) durch Entpacken dieses Paketes in das Verzeichnis `/opt/sparc-linux/`.

```
$ sudo mkdir /opt
$ sudo cp sparc-linux-1.0.0.tar.bz2 /opt/
$ cd /opt
$ sudo tar -xjvf sparc-linux-1.0.0.tar.bz2
```

Entpacken des SPARC Compilers

Anschließend müssen die entpackten Programme in die `$PATH` Umgebungsvariable aufgenommen werden. Für die *„bash-shell“* geschieht dies durch den Aufruf von:

```
$ EXPORT PATH=$PATH:/opt/sparc-linux/bin
```

SPARC Compiler in die Systemumgebung aufnehmen

Dabei ist zu Beachten, dass dieser Aufruf nur für die aktuelle Sitzung gilt und für jede neue Shell-Instanz erneut ausgeführt werden muss. Natürlich kann man zwecks Bequemlichkeit die `$PATH` Variable auch permanent erweitern. Dies geschieht durch einen entsprechenden Eintrag in die Datei `~/.profile`.

Damit ist dieser Teil der Installation abgeschlossen und kann durch Aufruf des Programmes `sparc-linux-gcc` aus einem beliebigen Verzeichnis verifiziert werden.

```
$ sparc-linux-gcc —version
Copyright (C) 2002 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR
A PARTICULAR PURPOSE.
```

Verifikation der SPARC-linux Installation

4.4.2 Vorbereiten von SnapGear V0.31

Die von Gaisler Research angebotene Originalversion 0.31 beinhaltet zwei für Leon2 relevante Fehler. Diese kamen im Zuge des Projektes zu Tage und führten zu einem Stopp der Entwicklung. Nur durch die tatkräftige Unterstützung durch Gaisler Research, die binnen weniger Werktage nach Fehlermeldung über das Forum, insgesamt zwei Patches zur Verfügung stellten, konnte diese Version stabil verwendet werden.

Die beiliegende CD beinhaltet die verwendete Version SnapGear V0.31 in unmodifizierter Version. Vor Konfiguration und Compilieren von SnapGear

müssen folglich beide Patches eingespielt werden, die sich ebenfalls auf der CD befinden. SnapGear kann in ein beliebiges Verzeichnis entpackt werden, in diesem Beispiel wird SnapGear in das Benutzerverzeichnis kopiert und dort bearbeitet.

```
$ cd ~
$ mkdir linux
$ cd linux
$ cp /cdrom/src/snapgear-1.0.31.tar.bz2 .
$ tar -xjvf snapgear-1.0.31.tar.bz2
```

Entpacken von SnapGear

Zum Einspielen der beiden Patchdateien sind diese ebenfalls in die Verzeichnisstruktur zu kopieren und mittels des Programmes `patch` einzuspielen.

```
$ cd ~/linux/snapgear-1.0.31/
$ cp /cdrom/src/leon2mmu_*.patch .
$ patch -p < leon2mmu_serial.patch
$ patch -p < leon2mmu_initramfs.patch
```

Einspielen der SnapGear Patches

4.4.3 Integration von portable.NET

SnapGear bietet einen dokumentierten Weg um eigene Programme in die Distribution aufzunehmen. Dieser Schritt erfordert mehrere Arbeitsgänge und ist in der Datei `Adding-User-Apps-HOWTO` im Verzeichnis `Documentation` beschrieben.

1. Erweitern der Datei `user/Makefile` um die neuen Programme
2. Hilfeeintrag für Konfigurationsskripte in `config/Configure.help` einfügen
3. Menüeintrag für die Konfiguration in `config/config.in` anlegen

Auf der CD finden sich diese Dateien in korrekter, abgeänderter Version.

4.4.4 Konfiguration von SnapGear

In diesem Abschnitt wird die Konfiguration von SnapGear beschrieben. Es wird hier die grafische Konfigurationsvariante anhand von Bildschirmfotos gezeigt. Für eine eins-zu-eins Reproduktion müssen alle Einstellungen wie abgebildet getroffen werden.

Im Hauptverzeichnis von SnapGear wird die Konfiguration mittels `make xconfig` gestartet:

`$ make xconfig`
Start der SnapGear Konfiguration

Die folgenden zwei Dialoge in Abbildung 4-10 und Abbildung 4-11dienen der Angabe von prozessortypischen Merkmalen:

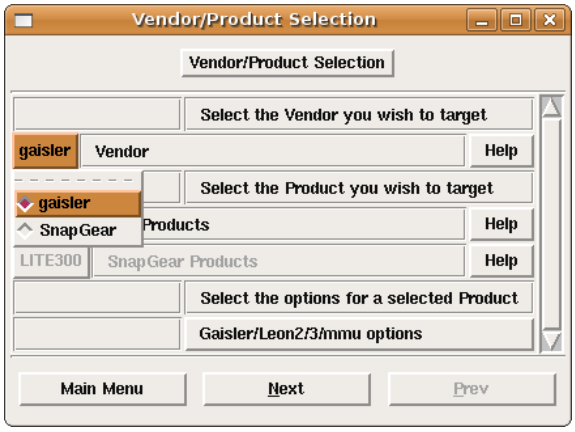

Abbildung 4-10: Herstellerauswahl

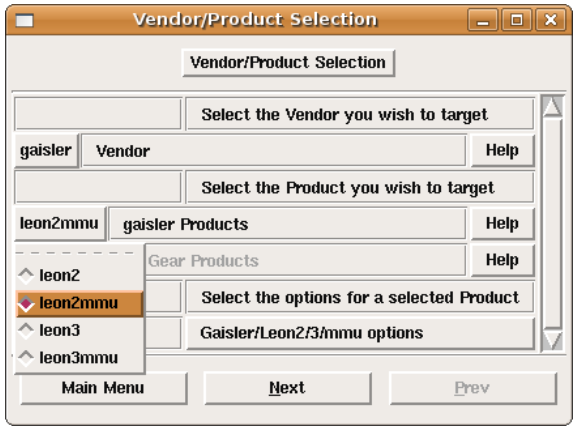

Abbildung 4-11: Prozessorkernauswahl

Als Hersteller wird `gaisler` angegeben, die verwendete Prozessorvariante wird mit `leon2mmu` spezifiziert. Bestätigung mit `Next` führt automatisch zum nächsten Dialog, der Leon2 Konfiguration.

In diesem Dialog sind Angaben zu Prozessorspezifika und zum Speicher zu tätigen (siehe Abbildung 4-12 und Abbildung 4-13).

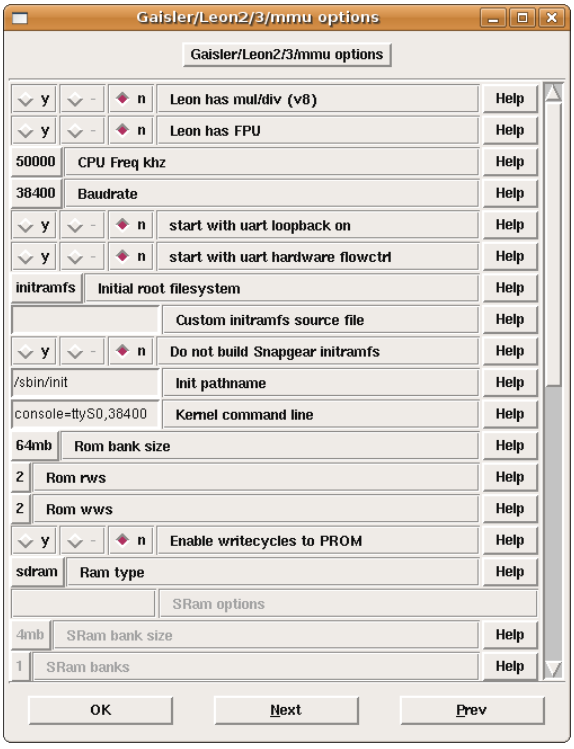

Abbildung 4-12: Leon2 Konfiguration 1

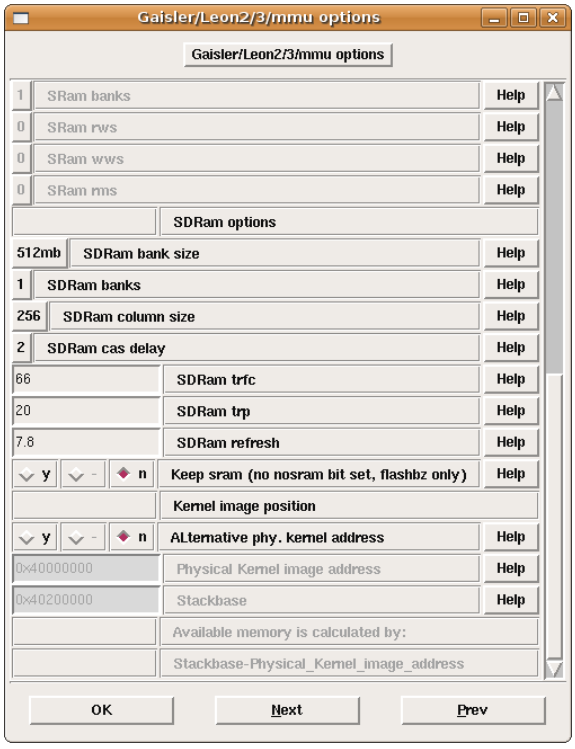

Abbildung 4-13: Leon2 Konfiguration 2

Nach Bestätigung durch `Next` folgen Konfigurationsangaben zu Linux und ob Benutzerprogramme konfiguriert werden wollen (siehe Abbildung 4-14):

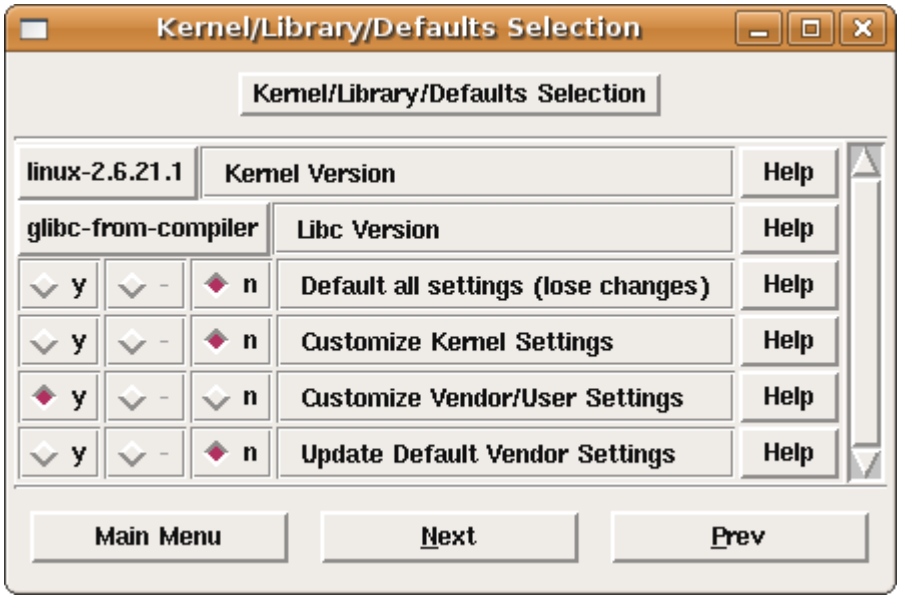

Abbildung 4-14: Kernel und Bibliotheksauswahl

Nach Abspeichern und Beenden dieser Konfiguration wird man automatisch zur Konfiguration der Benutzerprogramme weitergeleitet (siehe Abbildung 4-15). Portable.NET wurde in die Sparte *„Miscellaneous Applications“* eingeordnet.

Abbildung 4-15: Konfiguration der Benutzerprogramme

Am Ende der Programmliste findet sich der Eintrag für portable.NET. Soll dieses in SnapGear eingebunden werden, ist die nebenstehende Box zu auf „y“ zu setzen (siehe).

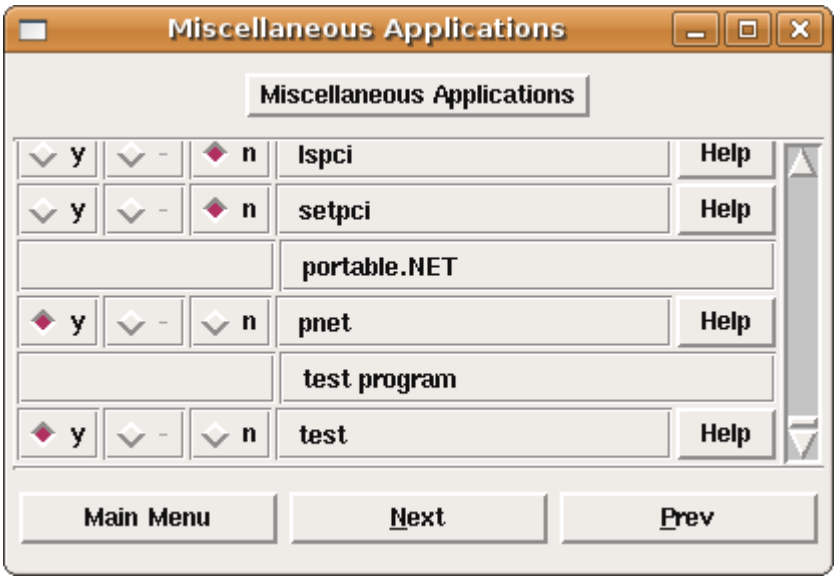

Abbildung 4-16: Konfiguration von portable.NET

4.4.5 Compilieren von SnapGear

Danach kann auch dieser Konfiguration durch „`Save and Exit`“ abgeschlossen werden und die Kompilation gestartet werden.

```
$ make clean; make
```

Compilieren von SnapGear

Das Compilieren von SnapGear nimmt je nach Prozessorleistung einige Zeit in Anspruch. Nach erfolgreichem Abschluss wurden im Unterverzeichnis `images` verschiedenste Dateien mit Betriebssystemabbildern erzeugt:

```
$ ls -1
image
image.dis
image.dsu
image.dsu.dis
image.dsu.x
image.flashbz
image.flashbz.dis
progs
```

Liste der erzeugten SnapGear Images

Für das spätere Laden mittels `grmon` wird lediglich die Datei `image.dsu` benötigt. Eine vorcompilierte Version befindet sich auf der beigelegten CD.

4.5 Demonstrationsprogramme

Zur Demonstration der Laufzeitumgebung wurden zwei Beispielprogramme in der Programmiersprache C# implementiert.

Die Laufzeit eines Programmes innerhalb von portable.NET hängt nicht ausschließlich von der Laufzeit des eigentlichen implementierten Algorithmus ab, sondern auch von anderen Faktoren wie Initialisierung der Laufzeitumgebung, Parsen des CIL-Assemblies, Konvertieren in CVM und letztlich der eigentlichen Programmausführung.

Das erste Beispielprogramm welches innerhalb einer Schleife Zugriffe auf zufällige Feldelemente eines eindimensionalen Feldes vornimmt, soll sicherstellen, dass aus den gewonnenen Messergebnissen etwaige Compileroptimierungen und Laufzeitumgebungsverluste extrahiert werden können (siehe Listing: random_access.cs S.88). Das Programm `random_access.cs` soll also vorwiegend dazu dienen einen einfacheren Bezug des Laufzeitverhaltens zuvor genannter Faktoren herzustellen.

```
public class RandomAccess {

public static void Main(string[] argv) {

    int repeat;
    int array_size = 100;
    int index;
    int temp;
    int[] array = new int[array_size];
    if(argv.Length != 1) { print_help(); return; }

    try {
      repeat = int.Parse(argv[0]);
    }
    catch(Exception e) {
      print_help();
      return;
    }

    Random number_generator = new Random();
    for(index = 0; index < array_size; index++) {
      array[index] = number_generator.Next(0, 100);
    }
    DateTime startTime = DateTime.Now;
    for(int i = 0; i < repeat; i++) {
      index = number_generator.Next(0, array_size – 1);
      temp = temp + array[index];
    }
    DateTime stopTime = DateTime.Now;
    TimeSpan elapsedTime = stopTime – startTime;
    Console.WriteLine ("Time in milliseconds:" +
                       elapsedTime.TotalMilliseconds);
}

public static void print_help() {
    Console.WriteLine ("Usage: ilrun random_access.exe [size]");
}
}
```

Listing: random_access.cs

Das zweite Demonstrationsprogramm `matrix_multiplication.cs` (siehe Listing: matrix_multiplication.cs S.90) implementiert die Multiplikation zweier quadratischer Matrizen welche zuvor mit Werten initialisiert werden. Dieses Programm soll vor allem das Verhalten der Implementation im Umgang mit mehrdimensionalen Feldern verifizieren.

Beide Programme erwarten jeweils einen einzigen Parameter - der die Größe der verwendeten Datenstruktur bestimmt und damit die Laufzeit der jeweiligen Algorithmen bestimmt. Im Falle der Matrixmultiplikation also die Größe der beiden zu multiplizierenden quadratischen Matrizen. Im Beispiel mit zufälligen Wertzuweisungen wird die Anzahl der gewünschten Algorithmuswiederholungen damit bestimmt. Dies ist vor allem für eine Messwertgewinnung vorteilhaft, da einfach unterschiedlicher Algorithmusaufwand erzeugt werden kann.

Das Laufzeitverhalten beider Programme skaliert jedenfalls nicht linear mit der Anzahl der Wiederholungen bzw. der Größe der Datenstrukturen. Die Messwerte zeigen gar, dass nicht einmal ein streng monotones Wachstumsverhalten gegeben ist.

Ohne genauer die Hintergründe dazu hinterfragt zu haben, liegt die Vermutung nahe, dass dies wohl zum Teil auf die Initialisierungsphase der Laufzeitumgebung zurückzuführen sein könnte. Diese scheint jedenfalls ungeachtet des ausgeführten Programmes nahezu konstant zu sein, womit sie abhängig von der Gesamtlaufzeit des ausgeführten Algorithmus natürlich immer einen unterschiedlichen Anteil an der Laufzeit des gesamten Prozesses einnimmt.

```
public class MatrixMultiplication {

public static void Main(string[] argv) {
    int array_size;
    int repeat = 1;
    int[,] array_1 = new int[array_size, array_size];
    int[,] array_2 = new int[array_size, array_size];
    int[,] result_array = new int[array_size, array_size];

    if(argv.Length != 1) {
      print_help();
      return;
   }
    try {
      array_size = int.Parse(argv[0]);
   }
    catch(Exception e) {
      print_help();
      return;
   }

    // initialize arrays
    for(int row = 0; row < array_size; row++) {
      for(int col = 0; col < array_size; col++) {
        array_1[row, col] = 5;
        array_2[row, col] = 10;
     }
   }
    // do the multiplication [repeat] times
    DateTime startTime = DateTime.Now;
    for(int a = 0; a < repeat; a++) {
      for(int i = 0; i < array_size; i++) {
        for(int j = 0; j < array_size; j++) {
          for(int k = 0; k < array_size; k++) {
            result_array[i,j] =  result_array[i,j] +
                                 array_1[i,k] * array_2[k,j];
         }
       }
     }
   }
    DateTime stopTime = DateTime.Now;
    TimeSpan elapsedTime = stopTime - startTime;
    Console.WriteLine ("Time in milliseconds: " +
                       elapsedTime.TotalMilliseconds);
}

public static void print_help()
{
    Console.WriteLine ("Usage: ilrun matrix_multiplication.exe
                       [size]");
}
}
```

Listing: matrix_multiplication.cs

4.5.1 Installation des portable.NET Compilers

Die beiden in C# Syntax vorliegenden Beispielprogramme `random_access.cs` und `matrix_multiplication.cs` müssen vor Interpretation durch den portable.NET Interpreter auf SCUP in CIL-Bytecode[42] übersetzt werden. Üblicherweise wird für CIL-Bytecode die Dateierweiterung `.exe` verwendet. Die Einhaltung dieser Konvention hat zwar keine funktionalen Auswirkungen, erscheint allerdings angesichts einer transparenten Entwicklung vorteilhaft. Ziel dieses Schrittes ist also zwei kompilierte Dateien `random_access.exe` bzw. `matrix_multiplication.exe` zu erzeugen.

Wie in Kapitel 3 Common Language Infrastructure auf S.31 beschrieben, kann Quellcode aus den unterstützen Programmiersprachen grundsätzlich mit jedem Compiler in CIL übersetzt werden, der Standardkonformen CIL-Code erzeugen kann. Der portable.NET Interpreter `ilrun` kann infolge das erzeugte Programm ausführen.

Dies hat den Vorteil, dass Zielplattform und Entwicklungsplattform nicht zwingend identisch sein müssen. CIL-Compiler und CIL-Interpreter können also auf gänzlich unterschiedlichen Systemen installiert sein. Dieser Umstand kommt SILC insofern entgegen, als es nie Ziel des Projektes war die gesamte portable.NET Umgebung auf snapgear zu portieren. Die Vorgangsweise bei der Übersetzung der C# Programme hängt also gänzlich vom verwendeten Entwicklungssystem ab.

Bei der vorliegenden Entwicklung wurde ausschließlich ein System basierend auf der Linuxdistribution Ubuntu verwendet. In der verwendeten Version 6.10 bietet Ubuntu mit den voneinander unabhängigen Paketen `pnet-compiler`, `pnet-interpreter` und `pnet-assemblies` die Möglichkeit, diese Freiheit entsprechend auszukosten. Da die Interpretation der Demonstrationsprogramme am Zielsystem SCPU erfolgen soll, reicht es, grundsätzlich nur den Compiler von portable.NET zu installieren. Die benötigten Pakete liegen jedenfalls im Repository „`universe`". Dieses muss

42 CIL ... Common Intermediate Language

bei Vorliegen einer unberührten Standardinstallation allerdings noch zu den Paketquellverzeichnissen hinzugefügt werden (siehe dazu [Fischer, 2007]). Die Installation des benötigten Compilers kann in Folge bequem über grafische Installationshilfsmittel wie „synaptic“[43] erfolgen, oder klassisch durch bemühen einer Shell und Verwendung des Programmes `apt-get`:

```
$ sudo apt-get update
...
$ sudo apt-get install pnet-compiler
```

Das erste Kommando aktualisiert das Paketverzeichnis, mit dem zweiten Kommando wird der portable.NET Compiler installiert.

Nach erfolgreicher Installation ist das Programm `cscc`[44] über den Pfad verfügbar und ausführbar. Ein Aufruf von cscc mit dem Parameter `--version` zeigt die installierte Version:

```
$ cscc --version
cscc version 0.7.4
```

Versionsinformation des cscc Compilers von portable.NET

Alternativ kann `cscc` natürlich auch direkt aus dem aktuellen Quellcode compiliert werden[45] oder auch ein anderer C# Compiler wie `mcs`[46] des mono Projektes verwendet werden. Trotzdem sowohl die C# Compiler `cscc` wie auch `mcs` funktional äquivalenten CIL Bytecode erzeugen, unterscheidet sich ein und dasselbe Programm mit verschiedenen C# Compilern übersetzt sowohl in Größe wie auch in Laufzeit. Einfache Messungen bei Verwendung des Interpreters `ilrun` am Entwicklungssystem mit zwei Varianten von `random_access.exe` haben ergeben, dass `cscc` kompakteren Code erzeugt, der überdies auch schneller abgearbeitet wird. Ein zeilenweiser Vergleich nach Disassemblierung mittels des Disassemblers des portable.NET Projektes

43 http://www.nongnu.org/synaptic/
44 http://dotgnu.org/pnet.html
45 http://dotgnu.org/pnet-install.html
46 http://www.mono-project.com/CSharp_Compiler

`ildasm`[47] hat gezeigt, dass sich die Unterschiede allerdings vorwiegend auf Codekommentare und Stackverwendung reduzieren.

Eine Verallgemeinerung dieser Beobachtung hinsichtlich Codegröße und Geschwindigkeit ist Aufgrund der kleinen Messdatenbasis allerdings sicherlich nicht möglich und auch nicht gewollt.

4.5.2 Compilieren der *.cs Dateien

Die Erzeugung des CIL Bytecodes erfolgt angelehnt an andere Compiler wie die C und C++ Compiler von GCC durch einfachen Aufruf von `cscc:`

```
$ cscc -o <PROGRAMMNAME.EXE> <BYTECODEDATEI.CS>
```

Erzeugen einer CIL Bytecodedatei mittels cscc

Die Hilfe gibt Auskunft über die Verwendung von `cscc`:

```
$ cscc –help
Usage: cscc [options] file ...
Options:
  -o <file>                Place the output into <file>
...
```

Verwendung von cscc

Zum Compilieren der Demonstrationsprogramme sind folglich zwei Aufrufe des Compilers zwecks Einfachheit im Verzeichnis des C# Quellcodes notwendig:

```
$ cd <QUELLCODEVERZEICHNIS>
$ cscc -o random_access.exe random_access.cs
$ cscc -o matrix_multiplication.exe matrix_multiplication.cs
```

Compilieren der Demonstrationsprogramme

47 siehe auch http://www.southern-storm.com.au/docs/pnettools_12.html

Im aktuellen Verzeichnis werden mit erfolgreicher Compilierung die zwei CIL Bytecodedateien erzeugt:

```
$ ls -1
matrix_multiplication.cs
matrix_multiplication.exe
random_access.cs
random_access.exe
```

Liste der compilierten Demonstrationsprogramme

4.5.3 Verifizieren der Demonstrationsprogramme

Da das Einspielen der erzeugten Demonstrationsprogramme in snapgear und damit auf SCPU eine Reihe von zum Teil langwierigen Arbeitsschritten erfordert, bietet sich eine Verifikation der korrekten Funktionalität der erzeugten Programme im Vorfeld am Entwicklungssystem an. Dies erfordert die Installation der Laufzeitumgebung `ilrun` auch am Entwicklungssystem.

Wie in „Versionsinformation des cscc Compilers von portable.NET„ auf Seite 92 ersichtlich, bietet Ubuntu in der verwendeten Version 6.10 nur Pakete von portable.NET in der Version 0.7.4 an. Da am Zielsystem allerdings aus technischen Gründen die aktuelle Quellcodebasis in Version 0.8 verwendet wurde (siehe Kapitel S.), liegt es – um etwaige Seiteneffekte zu vermeiden – nahe, zur Verifikation der Demonstrationsprogramme ebenfalls diese Version zu verwenden.

Dazu muss die aktuelle Quellcodebasis aus dem CVS[48] Repository von gnu.org bezogen werden und mittels der Entwicklungsumgebung des Entwicklungssystems compiliert werden. In der Installationsdokumentation[49] von portable.NET ist dieser Vorgang grob beschrieben, an dieser Stelle wird zwecks einfacher Reproduzierbarkeit eine detailliertere Installationsanleitung gegeben.

48 CVS ... **C**oncurrent **V**ersions **S**ystem
49 http://dotgnu.org/pnet-install.html

Die Installation von portable.NET aus den Quellen für die gewünschten Testzwecke erfordert drei Schritte:

1. Installation von `treecc`
2. Vorbereiten der Entwicklungstoolchain
3. Compilieren von `pnet`

Das Programm `treecc`[50] ist eigentlich nicht Teil der portable.NET Laufzeitumgebung, sondern ein Hilfsmittel zur Entwicklung von Compilern und anderen Programmen die extensiv Syntaxbäume verarbeiten. `treecc` ist nur während der Compilierung von `pnet` von Nöten.

Obgleich auch `treecc` über das Repository heruntergeladen wird, ist eine händische Konfiguration, Compilation und Installation nicht notwendig. Die von Ubuntu 6.10 angebotene Version 0.3.8 unterscheidet sich offenbar nicht maßgeblich von der aktuellen CVS Version 0.3.10, alle von `treecc` betroffenen Schritte ließen sich auch mit der Paketversion durchführen. Die Installation von `treecc` gestaltet sich also vergleichsweise einfach durch Einspielen des entsprechenden Ubuntu Paketes mittels `apt-get`:

```
$ sudo apt-get install treecc
```

Installation von treecc

Nach erfolgreicher Installation steht das Programm `treecc` im Pfad zur Verfügung. In den nachfolgenden Schritten wird dieses von den Konfigurationsprogrammen von `pnet` benötigt.

Zur Übersetzung von `pnet` sind einige wenige Systemvoraussetzungen zu schaffen. Dazu gehört gcc mit dem C Compiler wie auch weiterführende Werkzeuge zur Übersetzung von Programmen:

50 http://www.southern-storm.com.au/treecc.html

```
$ sudo apt-get install gcc build-essentials bison flex automake
                       autoconf
```

Installation der Übersetzungswerkzeuge für pnet

Nach erfolgreicher Installation obiger Programme kann portable.NET in der aktuellen CVS Version heruntergeladen werden. Der Vollständigkeit halber wird an dieser Stelle auch die Installation der Konsolenversion des CVS Clients angeführt.

Um eine klare Trennung zwischen portable.NET für das Entwicklungssystem und der Variante für SILC zu schaffen, empfiehlt es sich zuvor eine entsprechende Verzeichnisstruktur zu schaffen:

```
$ sudo apt-get install cvs
$ mkdir portable.NET.locale
$ cd portable.NET.locale
$ cvs -z3 -d:pserver:anonymous@cvs.sv.gnu.org:/sources/dotgnu-
                                                pnet  checkout .
```

CVS checkout von portable.NET

Je nach Geschwindigkeit der Internetverbindung benötigt dieser Vorgang etwas Zeit, schließlich wird neben des Interpreters `pnet` auch der Sourcecode des experimentellen JIT-Compilers `libjit`, `treecc`, das C Compilerfrontend `pnetC`, Testprogramme und ergänzende .NET Bibliotheksimplementierungen `pnetlib` heruntergeladen.

Zur Verifikation der Demonstrationsprogramme wird lediglich der Interpreter `pnet` benötigt. Die Konfiguration und Installation erfolgt analog zu Kapitel 75 Vorbereitung von portable.NET um bestmöglichen Vergleich zuzulassen. Bei Installation der CVS Version müssen vor Konfiguration notwendige Makefiles (`Makefile.in`) erstellt werden. Das mitgelieferte Shellscript `auto_gen.sh`, welches sich u.A. der zuvor installierten Programme `automake` und `autoconf` bedient, erledigt dies sofern alle Vorbedingungen erfüllt sind automatisch:

```
$ ./auto_gen.sh
```

Generieren der notwendigen Makefiles

Nach erfolgreicher Abarbeitung dieses Schrittes liegen alle für die weitere Konfiguration notwendigen Dateien vor. Folgend sind die unter Linux üblichen Schritte zur Vollendung der Installation - `./configure`, `make` und schlussendlich `make install` beschrieben. Der Hilfe des `./configure` Skriptes ist die Verwendung der relevanten Konfigurationsparameter zu entnehmen:

```
$ ./configure –help
`configure' configures this package to adapt to many kinds of
systems.
Usage: ./configure [OPTION]... [VAR=VALUE]...
...
Installation directories:
--prefix=PREFIX   install architecture-independent files in PREFIX
...
Optional Features:
...
--disable-tools   Remove the developer tools from the build
...
Optional Packages:
...
--without-libffi  disable libffi support
...
--without-libgc   disable libgc support
```

Hilfe zu ./configure auf dem Entwicklungssystem

Hinsichtlich der eingeschränkten Verwendung des Interpreters am Zielsystem SCPU wird pnet auch am Entwicklungssystem weitestgehend einfach und klein gehalten. Dies wird über obig beschriebene Konfigurationsoptionen `--disable-tools`, `--without-libffi` und `--without-libgc` erreicht. Um den Interpreter von portable.NET am Entwicklungssystem in dieser Konfiguration im Verzeichnis pnet_x86 des Benutzerverzeichnisses zu installieren ist `./configure` folglich wie folgt auszuführen:

```
$ ./configure --prefix=~/pnet_x86 --disable-tools
                                  --without-libffi
                                  --without-libgc
```

Aufruf von ./configure am Entwicklungssystem

Nach erfolgreicher Vollendung von `./configure` fehlt lediglich noch der Aufruf von `make` und `make install` um die Installation abzuschließen:

```
$ make
...
$ make install
```

Kompilieren und Installieren von pnet am Entwicklungssystem

Um die korrekte Installation zu überprüfen wird das soeben installierte Programm `ilrun` aufgerufen:

```
$ cd ~/pnet-x86/bin
$ ./ilrun --version
ILRUN 0.8.1 - IL Program Runtime
...
```

Testen von ilrun am Entwicklungssystem

4.5.4 Testen der Demonstrationsprogramme am Entwicklungssystem

Zum Test der in Bytecode vorliegenden Demonstrationsprogramme werden diese zwecks Einfachheit in die soeben erstellte Verzeichnisstruktur von `pnet` kopiert. Von dort könne diese durch Aufruf mittels `ilrun` einfach getestet werden:

```
$ cp <QUELLCODEVERZEICHNIS>/*.exe ~/pnet-x86/bin/

$ ./ilrun matrix_multiplication.exe 10
Time in milliseconds: 0,646

$ ./ilrun random_access.exe 10
Time in milliseconds: 0,591
```

Testen der Demonstrationsprogramme

Die abgebildeten Resultate differenzieren natürlich abhängig vom verwendeten Entwicklungssystem.

4.5.5 Einbinden der Demonstrationsprogramme in SnapGear

Um die erstellten CIL Programme in SnapGear ausführen zu können, müssen diese zuerst in das Betriebssystemabbild integriert werden. Da es sich hierbei praktisch nur um ein Kopieren der Dateien handelt, gestaltet sich dieser Schritt recht einfach. Um die Handhabung auf der Entwicklungsplattform zu vereinfachen, werden die Programme gleich in das /bin Verzeichnis der Distribution kopiert, in diesem findet sich auch der portable.NET Interpreter ilrun. Im Testablauf können somit die Programme leicht gestartet werden.

```
$ cd ~/linux/snapgear-1.0.31/romfs/bin
$ cp <QUELLCODEVERZEICHNIS>/*.exe .
```

Einbinden der Demonstrationsprogramme in SnapGear

Im Anschluss müssen die Betriebssystemimages neu erstellt werden. Dazu reicht ein make image, eine erneute Compilation von SnapGear ist natürlich nicht notwendig.

```
$ cd ~/linux/snapgear-1.0.31/
$ make image
```

Erneutes Erstellen der Betriebssystemabbilder

4.6 Testablauf

Der Testablauf gliedert sich in zwei Teile. Zum Einen muss das erstellte Betriebssystem geladen und gestartet werden, zum Anderen werden die erstellten Testprogramme zwecks Messdatengewinnung aufgerufen.

Für die folgenden Schritte wird eine korrekte serielle Verbindung zwischen Entwicklungssystem und Linuxrechner vorausgesetzt. Um das Betriebssystem auf SCPU zu laden und zu starten, wird das Programm `grmon` verwendet. Um mit dem laufenden SnapGear-System zu kommunizieren wird am Linuxrechner das Programm `minicom` verwendet.

Da `grmon` am Windowssystem läuft, muss zuerst das Betriebssystemabbild (`image.dsu`) auf den Windowsrechner transferiert werden. Zwecks Einfachheit empfiehlt es sich in der Windows-Shell in das Verzeichnis zu wechseln, in dem die Datei `image.dsu` gespeichert wurde.

Beim Start von `grmon` ist jedenfalls die Option `-nb` anzugeben. Das Laden des Betriebssystems erfolgt über das `grmon` Kommando `load`. Das geladene Betriebssystem wird durch das `grmon` Kommando `run` gestartet. In Folge erscheint die Ausgabe des startenden Betriebssystems in der zuvor gestarteten `minicom` Applikation am Linuxrechner.

```
Welcome to minicom 2.2
OPTIONS: I18n
Compiled on Mar  7 2007, 15:10:03.
Port /dev/ttyS0

Press CTRL-A Z for help on special keys
```

Starten von minicom

`minicom` muss für die verwendete Boud-Rate von 38600 konfiguriert werden. Wird dies unterlassen, erscheint zwar eine Ausgabe, die Zeichen sind jedoch verstümmelt.

```
C:\>grmon-eval -nb
 GRMON LEON debug monitor v1.1.19a (evaluation version)
 Copyright (C) 2004,2005 Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to support@gaisler.com

 This evaluation version will expire on 28/6/2007
try open device //./com1
###opened device //./com1

 GRLIB plug&play not found, switching to LEON2 legacy mode

 initialising ............
 detected frequency:  50 Mhz

 Component                            Vendor
 LEON2 Memory Controller              European Space Agency
 LEON2 AHB Status & Failing Addr      European Space Agency
 LEON2 SPARC V8 processor             European Space Agency
 LEON2 Write Protection               European Space Agency
 LEON2 Configuration register         European Space Agency
 LEON2 Timer Unit                     European Space Agency
 LEON2 UART                           European Space Agency
 LEON2 UART                           European Space Agency
 LEON2 Interrupt Ctrl                 European Space Agency
 LEON2 I/O port                       European Space Agency
 AHB Debug UART                       Gaisler Research
 LEON2 Debug Support Unit             Gaisler Research
 Use command 'info sys' to print a detailed report of attached
cores

grlib> load image-2.6-V0.31.40.dsu
section: .stage2 at 0x40000000, size 10180 bytes
section: .vmlinux at 0x40004000, size 2302016 bytes
total size: 2312196 bytes (87.6 kbit/s)
read 3505 symbols
entry point: 0x40000000
grlib> run
```

Laden und Starten des Betriebssystems mittels grmon

Der Bootprozess von SnapGear gibt Auskunft über die erkannte Hardware und grundlegende Parameter. Nach erfolgreichem Betriebssystemstart wird automatisch eine Shell gestartet über die in Folge die Demonstrationsprogramme ausgeführt werden können.

```
Booting Linux...
PROMLIB: Sun Boot Prom Version 0 Revision 0
Linux version 2.6.18.1 (heinix@SCPU) (gcc version 3.2.2) #16 Tue
Jul 3 18:07:01 CEST 2007
ARCH: LEON
TYPE: Leon2/3 System-on-a-Chip
Ethernet address: 0:0:0:0:0:0
CACHE: direct mapped cache, set size 8k
Boot time fixup v1.6. 4/Mar/98 Jakub Jelinek (jj@ultra.linux.cz).
Patching kernel for srmmu[Leon2]/iommu
node 2: /<NULL> (type:cpu) (props:.node device_type mid mmu-nctx
clock-frequency uart1_baud uart2_baud )
PROM: Built device tree from rootnode 1 with 914 bytes of memory.
DEBUG: psr.impl = 0xf  fsr.vers = 0x7
Built 1 zonelists.  Total pages: 15794
Kernel command line: console=ttyS0,38400 rdinit=/sbin/init
PID hash table entries: 256 (order: 8, 1024 bytes)
Todo: init master_l10_counter
Console: colour dummy device 80x25
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 61568k/65536k available (912k kernel code, 3956k
reserved, 104k data, 1224k init, 0k highmem)
Mount-cache hash table entries: 512
io scheduler noop registered
io scheduler cfq registered (default)
Serial: Leon driver, author: Konrad Eisele<eiselekd@web.de>
Serial: system frequency: 50000 khz, baud rates: 38400 38400
ttyS0 at MMIO 0x80000070 (irq = 3) is a Leon
ttyS1 at MMIO 0x80000080 (irq = 2) is a Leon
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024
blocksize
loop: loaded (max 8 devices)
Freeing unused kernel memory: 1224k freed
init started:  BusyBox v0.60.5 (2007.07.03-16:02+0000) multi-call
binary

Sash command shell (version 1.1.1)
/>
```

Bootprozess von SnapGear

Die eigentliche Programmausführung von `ilrun` und den Demonstrationsprogrammen unterscheidet sich nicht von der Ausgabe am Linuxrechner. `ilrun` wie auch die Demonstrationsprogramme finden sich im Verzeichnis `bin`.

```
$ cd bin
$ ./ilrun matrix_multiplication.exe 10
Time in milliseconds: 2,337
pid 15: failed 58368
$ ./ilrun random_access.exe 10
Time in milliseconds: 2,112
pid 15: failed 58743
```

Testen der Demonstrationsprogramme auf SnapGear

Die Fehlermeldung beim Terminieren der Programmläufe kann getrost ignoriert werden. Diese tritt in der konfigurierten Version von SnapGear bei jedem Programmaufruf auf. Es konnten jedoch keine Beeinträchtigungen des Verhaltens festgestellt werden.

5 Resultate und Zusammenfassung

Wie bereits erwähnt, entsprechen die gewonnenen Testergebnisse nicht ganz den ursprünglichen Erwartungen. Ein Geschwindigkeitsvorteil durch die getätigten Modifikationen in Hard- und Software konnte nicht festgestellt werden. Dies hat mehrere, in den vorhergehenden Kapiteln besprochene, Gründe. Der rein rechnerisch zu erwartende Geschwindigkeitsvorteil ist verglichen mit Softwareoptimierungen nicht sehr groß. Von den sechs notwendigen Instruktionen für eine Wertebereichsüberprüfung inklusive Fehlerbehandlung können lediglich die beiden Vergleiche durch parallele Abarbeitung eliminiert werden. Die eigentlichen Zeitfresser, das Laden von oberer und unterer Grenze in die jeweiligen Register und die bedingten Sprunganweisungen können naturgemäß nicht verhindert werden. Somit ergibt sich mit dieser Betrachtung unter der Annahme, dass all diese Instruktionen gleich lange brauchen (z.B. bei x86 oder SPARC), ein rein rechnerisches Einsparungspotential von 2/6.

Diese Einsparung wird in der vorliegenden Implementierung allerdings durch systembedingten Unzulänglichkeiten mehr als aufgefressen. Diese sind vor allem in der Fehlerbehandlung zu suchen. Der Umstand, dass ein vorhandener Trap kopiert und adaptiert wurde, dabei aber das Signal an das Betriebssystem nicht angepasst wurde, erfordert eine spezielle Behandlung des Signals. Vor der Abarbeitung eines „*bound checks*“ muss der eigentliche Signalhandler gesichert und mit dem neuen überschrieben werden. Nach der Abarbeitung muss dieses wieder rückgängig gemacht werden. Bei der Messdatengewinnung wurde auf dieses korrekte Vorgehen zwecks Test verzichtet, d.h. das Signal wurde permanent überschrieben. Dieses Vorgehen brachte leicht bessere Ergebnisse.

Ein weiterer bremsender Faktor ist, dass die Ausnahmebehandlung nicht im Signalhandler ausgeführt werden kann. In betriebssystemspezifischer Literatur wird oftmals gepredigt, Signalhandler so kurz wie möglich zu halten um etwaige Seiteneffekte weitestgehend auszuschließen. Dieser Empfehlung

wurde dadurch Rechnung getragen, als im Signalhandler nur eine globale Variable gesetzt wurde und damit ein Fehlerfall angezeigt wird. In Folge muss nach dem eigentlichen „bound check“ diese Variable überprüft werden. Diese Überprüfung entspricht dem bedingten Sprung, der auch in der konventionellen Umsetzung anfällt.

Um Aussage über den Laufzeitbedarf von „*bound checks*“ in der unveränderten portable.NET Laufzeitumgebung treffen zu können, wurde zum Test diese Funktionalität deaktiviert. Anzumerken ist jedoch, dass die gewonnenen Ergebnisse immer in Relation zu den Benutzerprogrammen zu sehen sind. Bei zugriffsintensiven Programmen ist der Anteil naturgemäß hoch.

Folgende Tabelle zeigt das Laufzeitverhalten der beiden Demonstrationsprogramme in vier Konfigurationen der Laufzeitumgebung: ohne Wertebereichsüberprüfung (I), mit normaler Wertebereichsüberprüfung (II), mit SCPU Wertebereichsüberprüfung (III) und mit global aktiviertem Signalhandler (IV). Die Angaben entsprechen der gemessenen Laufzeit bei 10.000 Iterationen der relevanten Programmteile in Millisekunden:

	I	**II**	**III**	**IV**
Matrixmultiplikation	5054 (87%)	5789 (100%)	6830 (118%)	6473 (112%)
Zufälliger Zugriff	4876 (94%)	5213 (100%)	5776 (111%)	5617 (108%)

Trotz dieser Ergebnisse birgt die grundlegende Idee hinter SCPU und SILC, Wertebereichsüberprüfungen in die CPU zu bringen, durchaus Potential. Bei verbesserter Umsetzung ist ein Geschwindigkeitsvorteil durchaus realistisch, obgleich dieser Aspekt eigentlich nicht im Vordergrund des Forschungsprojekts steht. Durch die Hardwareunterstützung könnten insbesondere in eingebetteten Systemen, Softwarehersteller eher geneigt sein, auf implizite Wertebereichsüberprüfung zu setzen. Der damit verbundene Vorteil in Bezug auf Codequalität und Sicherheit kann in vielen Fällen einen Einsatz derartiger Systeme mit entsprechenden Hochsprachen wie C# rechtfertigen, auch wenn durch die reine Hardwareerweiterung nicht

zwingend gewaltige Geschwindigkeitsvorteile zu erwarten sind. Diese sind, wie bereits erläutert, leichter durch Compileroptimierungen zu erreichen.

Neben dem Einsatz von SCPU und SILC für „`bound checks`“ bietet sich die Ausweitung derselben Technologie auf Wertebereichsüberprüfungen von allen Datentypen, insbesondere auch von primitiven Datentypen an. Entsprechende Programmiersprachen vorausgesetzt, können so Subrange Types integraler Bestandteil des Hardware-Software Verbandes werden und ebenfalls helfen, Codequalität und Sicherheit positiv zu beeinflussen.

5.1 Raum für Verbesserungen

In diesem Abschnitt werden Vorschläge für Verbesserungen gebracht. Diese beziehen sich auf die beiden Teilbereiche Hardware und Software.

Hardwareseitig empfiehlt es sich, auf die aktuelle Version von Leon, also Leon3, umzusteigen. Eine weiterführende Arbeit mit Leon2, insbesondere wenn unbekannte Wege beschritten werden wollen, ist aufgrund der mangelnden Unterstützung von Gaisler Research nicht empfehlenswert.

Die SPARCV8 Architektur sieht Raum für benutzerdefinierte Traps vor. Anstelle einen bestehenden Trap - und damit ein bestehendes Signal für die Anzeige von Wertebereichsverletzungen zu verwenden - könnten diese Benutzerdefinierten eingesetzt werden.

Softwareseitig gibt es ebenfalls viel Raum für Verbesserungen. Mit dem Einzug von Leon3 könnte auch Debian als Betriebssystem mit den damit einhergehenden Annehmlichkeiten eingesetzt werden. Dies betrifft vor allem die komfortable Paketverwaltung oder die leichtere Netzwerkanbindung. Überdies verfügt auch die SPARC/Linux Variante von Debian über eine Vielzahl von Paketen für den praktischen Alltag.

Mit diesem Schritt wäre auch der Einsatz einer anderen CLI Laufzeitumgebung denkbar, wobei sich vor allem Mono anbieten würde. In diesem Zuge müssten allerdings die getätigten Modifikationen grundlegend überdacht werden. Schließlich arbeitet Mono mit einem JIT Compiler,

während portable.NET einen Interpreter bemüht. Dieser Fortschritt wäre allerdings sicherlich auch der Geschwindigkeit dienlich.

Quellcode- und Kommandoverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Abrams, 1998]: Marc Abrams, World Wide Web - Beyond the Basics, 1998

[AMD, 1995]: Daniel Mann, Evaluating and Programming the29KE RISC FamilyThird Edition, , 1995

[Chuang et al., 2007]: Weihaw Chuang, Satish Narayanasamy, Brad Calder, "Bounds Checking with Taint-Based Analysis" in International Conference on High Performance Embedded Architectures & Compilers, 2007

[CLIWP, 2007]: Wikipedia, Common Language Infrastructure, http://de.wikipedia.org/w/index.php?title=Common_Language_Infrastructure&oldid=32502599, zuletzt besucht am: 2007

[dotgnu, 2007]: Norbert Bollow, DotGNU Portable.NET, http://dotgnu.org/pnet.html, zuletzt besucht am: 05.07.2007

[ECMA-355, 2006]: Microsoft, Hewlet Packard, Intel et. al, Common Language Infrastructure (CLI), 2006

[Egyedi, 2001]: Tineke M. Egyedi, "Why Java™ Was Not Standardized Twice" in Proceedings of the 34th Hawaii International Conference on System Sciences - 2001, 2001

[Fischer, 2007]: Marcus Fischer, Ubuntu GNU/Linux, 2007

[Gaisler GRMON, 2007]: Gaisler Research, GRMON User’s Manual, , 2007

[Gaisler, 2003]: Jiri Gaisler, Preparations for a next-generation SPARC processor, 2003

[Gaisler, 2004]: Gaisler Research, LEON2 Processor User's Manual, XST Edition, , 2004

[Gaisler, 2007]: Daniel Hellström, SnapGear Linux for LEON, Gaisler Research, 2007

[Grasser, 2007]: Micheal Grasser, Secure CPU, Eine sichere Prozessorarchitektur für den Einsatz in eingebetteten Systemen, 2007

[Hyde, 2004]: Randall Hyde, The Art of Assembly Language, 2004

[Intel, 1998]: Intel Corporation, i960® VH Processor, Developer's Manual, , 1998

[Intel, 1999]: INTEL CORPORATION, Instruction Set Reference, , 1999

[Intel, 2007]: INTEL CORPORATION, Instruction Set Reference, A-M, , 2007

[ISO/IEC 23271,2006]: , Information technology -- Common Language Infrastructure (CLI) Partitions I to VI, , 2006

[Lam, Chiueh 2005]: Lam, Chiueh, "Checking array bound violation using segmentation hardware" in Dependable Systems and Networks, 2005, 2005

[Lam, Chiueh, 2005]: Lam, Chiueh, "Checking array bound violation using segmentation hardware" in Dependable Systems and Networks, 2005, 2005

[MONO, 2007]: , Mono Project, http://www.mono-project.com/, zuletzt besucht am: 22.08.2007

[NETWP, 2007]: Wikipedia, .NET Framework Wikipedia The Free Encyclopedia, http://en.wikipedia.org/w/index.php?title=.NET_Framework&oldid=152120819, zuletzt besucht am: 2007

[PatHen, 1996]: D. A. Patterson, J. L. Hennessy, Computer Architecture: A Quantitative Approach, 1996

[Patil, Fischer, 1997]: Harish Patil, Charles Fischer, "Low-cost, Concurrent Checking of Pointer and Array Accesses in C Programs" in SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(1), 1997

[Pender, 2006]: Gaisler Research AB, Pender Electronic Design GmbH, GR-XC3S-1500 Development Board User Manual, , 2006

[Rajiv Gupta, 1990]: Rajiv Gupta, "A Fresh Look at Optimizing Array Bound Checking" in Proceedings of the ACM SIGPLAN’SO Conference on Programming Language Design and Implementation, 1990

[Rajiv Gupta, 1993]: Rajiv Gupta, "Optimizing Array Bound Checks Using Flow Analysis" in ACM Letters on Programming Languages and Systems, Vol.2, 1993

[Shao et al., 2005]: Zili Shao et al., "Efficient Array & Pointer Bound Checking Against BufferOverflow Attacks via Hardware Software" in IEEE International Conference onInformation Technology: Coding and Computing, 2005

[Sparc, 1991]: Englewood Cliffs, SPARC International, Inc., The SPARC Architecture Manual, Version 8, , 1991

[Sun, 2007]: Sun, OPENJDK COMMUNITY TCK LICENSE AGREEMENT V 1.0, , 2007

[WP, 2007]: , Overview of the Common Language Infrastructure, http://en.wikipedia.org/w/index.php?title=Image:Overview_of_the_Common_Language_Infrastructure.png&oldid=118752688, zuletzt besucht am: 2007

[Xi, Xia, 1999]: Hongwei Xi, Songtao Xia, "Towards array bound check elimination in Java TM virtual machine language" in Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, 1999