C-Compiler für Embedded Systems

Technologie und Werkzeuge zur Compilerentwicklung

Dr. Rainer Leupers

Universität Dortmund
Lehrstuhl Informatik 12
44221 Dortmund
email: Rainer.Leupers@cs.uni-dortmund.de

Für Embedded Systems setzen sich C-Compiler zur Softwareentwicklung gegenüber der traditionellen Assemblerprogrammierung immer mehr durch. Allerdings werden oft Spezialprozessoren eingesetzt, für die keine C-Compiler verfügbar sind. Da der Entwicklungsaufwand für neue Compiler hoch ist, sollten existierende Tools ausgenutzt werden. Dieser Artikel nennt wichtige Anforderungen an C-Compiler und gibt eine Übersicht verfügbarer Techniken und Tools[1].

Embedded Systems werden heute meist aus Hardware- und Softwarekomponenten aufgebaut, wobei der Trend aufgrund verschiedener Vorteile klar in Richtung Software geht: Software ist flexibler, kann wiederverwendet werden und ist kostengünstiger in der Entwicklung als Hardware. Embedded Software läuft auf speziellen Prozessoren, darunter RISCs, DSPs, Multimedia-Prozessoren und Mikrocontrollern. Zur Programmierung dieser Prozessoren sind C-Compiler notwendig. Die bisher übliche fehleranfällige und zeitaufwendige Assemblerprogrammierung wird zunehmend durch die Verwendung von C-Compilern abgelöst. C bietet auch den Vorteil der leichten Portierbarkeit auf andere Plattformen.

In der Tat etabliert sich C (zukünftig evtl. auch C++ und Java) langsam aber sicher zu einer Sprache zur Spezifikation von Software *und* Hardware von Embedded Systems. Der Grund ist, dass C allgemein bekannt ist, auf praktisch allen Entwicklungsplattformen zur Verfügung steht und abstraktere Spezifikationen im Vergleich zu Hardware-Beschreibungssprachen wie VHDL und Verilog ermöglicht. Ein Beispiel ist die SystemC-Initiative [1] zur Systemspezifikation, die von vielen führenden CAD-, Halbleiter- und Systemhäusern unterstützt wird. Auch bietet C eine wesentlich höhere Simulationsgeschwindigkeit als Hardware-Beschreibungssprachen. Sprachen wie VHDL und Verilog werden daher langfristig nicht mehr zum Design Entry sondern eher als Austauschformate zwischen CAD-Tools dienen.

Die zunehmende Verbreitung von C beim Embedded System Design spiegelt sich auch in neuen Design-Tool-Generationen wieder: Firmen wie ACE [2], Target Compiler Technologies [3] und Archelon [4] bieten sogenannte *retargierbare* C-Compiler an, welche für eine ganze Klasse von Prozessoren (statt wie üblich nur für eine bestimmte Maschine) Assemblercode generieren können (für eine genaue Beschreibung siehe [5]). Dies erleichtert den Übergang auf neue Zielarchitekturen ohne Verlust der bisherigen Toolchain. Dies ist insbesondere für Embedded Systems wichtig, da in diesem Bereich häufig anwendungsspezifische Prozessoren zum Einsatz kommen, für die keine Standard-Entwicklungstools existieren. C Level Design [6] bietet neuartige Hardware-Entwurfswerkzeuge auf der Basis von C/C++ an. Andere Firmen gehen noch einen Schritt weiter: Bei Tensilica [7] kann man per Internet online einen Prozessor-Core konfigurieren und anschliessend ein VHDL-Modell für die Hardwaresynthese sowie die passenden Entwicklungstools inkl. C-Compiler herunterladen.

Der Einsatz von retargierbaren Compilern ist allerdings nur für ganz bestimmte Klassen von eingebetteten Prozessoren möglich. Somit stehen momentan viele Firmen mit anwendungsspezifischen Prozessoren vor dem Problem der Entwicklung eines eigenen C-Compilers, um die Softwareentwicklung großer Applikationen in Assembler zu vermeiden. Dieser Artikel soll daher einen Überblick über die Technologie von C-Compilern geben, um einen Einstieg in das Thema zu ermöglichen und Entscheidungshilfen zu geben.

Compilerqualität

Am Anfang des Compilerentwurfs steht zunächst die Frage: Was soll der Compiler leisten ? Folgende Kriterien sind hierbei zu berücksichtigen:

Kosten: C-Compiler sind komplexe Softwareprodukte mit beträchtlichem Entwicklungsaufwand und -kosten. Die Kosten sollten daher in einem vernünftigen Verhältnis

[1] Publication: Elektronik 19/2000, WEKA Verlag, Munich, Sept 2000

zum Nutzen des Compilers stehen. Wird der Compiler vorwiegend firmenintern eingesetzt, so sollte soweit wie möglich auf fertige **Tools zur Compilerentwicklung** zurückgegriffen werden. Der Einsatz solcher Tools geht zu Lasten der Compilereffizienz, d.h. der Compiler wird größer und langsamer, allerdings wird der Entwicklungsaufwand deutlich reduziert.

Codequalität: Dies ist besonders für Embedded Systems mit Realzeitanforderungen und Systems-on-a-Chip mit kleinem Programmspeicher wichtig. Häufig wird gefordert, dass die Qualität von compilergeneriertem Code der von handgeschriebenem Assemblercode sehr nahe kommen muss. Hierzu sind spezielle Optimierungstechniken erforderlich, z.B. bei digitalen Signalprozessoren [8]. Neben **Performance und Codegröße** wird bei portablen Systemen zunehmend auch die Verlustleistung als Qualitätsmaß wichtig (Stichwort **Low Power**), die ebenfalls durch den Compiler beeinflusst wird.

Korrektheit: Da ein Compiler als Basiswerkzeug zur Softwareentwicklung eingesetzt wird, muss er (möglichst) korrekt arbeiten. Korrekte C-Programme müssen in korrekten Assemblercode übersetzt werden, während fehlerhafte C-Programme mit entsprechenden Fehlermeldungen zurückgewiesen werden müssen. Da eine formale Verifikation von Compilern zu aufwendig ist, gibt es hauptsächlich zwei Wege, um Korrektheit sicherzustellen: Zum einen umfangreiche **Tests** mit einer hohen Fehlerabdeckung (z.B. mit Hilfe von kommerziellen Testsuites [9]), und zum anderen der Einsatz von Tools zur **automatischen Erzeugung von Compilerkomponenten**. Beispiele hierfür werden später genannt.

Übersetzungsgeschwindigkeit: Compiler für Embedded Systems müssen nicht sehr schnell sein. Einerseits sind die zu übersetzenden C-Programme meist nicht sehr groß und andererseits lässt sich eine sehr hohe Codequalität nur durch den Einsatz zeitaufwendiger Optimierungen erreichen. Es ist sinnvoll, mit verschiedenen **Optimierungsstufen** zu arbeiten: Während der Entwicklungs- und Debug-Phase eines Programms braucht der Compiler nicht zu optimieren und soll schnell arbeiten. Für das fertige Produkt soll der Compiler dann einmal mit hohem Optimierungsaufwand z.B. möglichst kompakten Code erzeugen, um Chipfläche für den Programmspeicher zu sparen.

Aufbau eines C-Compilers

Ein C-Compiler übersetzt ein maschinenunabhängiges Quellprogramm in maschinenspezifischen Assemblercode und durchläuft dabei verschiedene Phasen (Abb. 1).

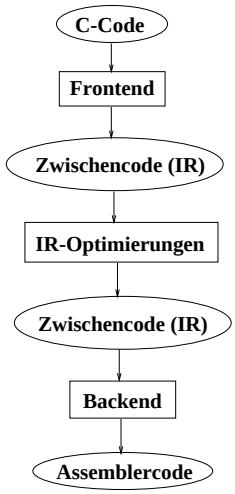

Abbildung 1: *Compilerphasen*

Der C-Quellcode wird zunächst durch das **Frontend** analysiert. Man unterscheidet lexikalische, syntaktische und semantische Analyse. In der lexikalischen Analyse werden Zeichenfolgen erkannt, z.B. C-Schlüsselwörter wie *while* oder *float*, die für den Compiler eine bestimmte Bedeutung haben. Die syntaktische Analyse prüft, ob das C-Programm korrekt "zusammengebaut" ist (z.B. ob Anweisungen mit einem Semikolon abgeschlossen sind) und in welcher Reihenfolge Ausdrücke auszuwerten sind (z.B. Multiplikation vor Addition). Zuletzt wird in der semantischen Analyse geprüft, ob Variablen korrekt deklariert sind und welche Datentypen die einzelnen Ausdrücke haben. Bei fehlerhaften Programmen gibt das Frontend entsprechende Meldungen aus.

Ansonsten wird als Ergebnis ein **Zwischencode** (intermediate representation, IR) generiert, welcher verschiedene Zwecke erfüllt. Zum einen können die nachfolgenden Phasen nun auf garantiert korrektem Code aufsetzen und müssen keine Prüfungen mehr vornehmen. Zum anderen besitzt die IR einen wesentlich einfacheren Aufbau als der C-Quellcode, wodurch Optimierungen und die Umsetzung in Assemblercode vereinfacht werden. Üblich ist die Verwendung von **Drei-Adress-Code** als IR: Komplexe Anweisungen werden in Folgen von einfachen Anweisungen mit jeweils höchstens drei Operanden aufgespalten. Ebenso werden Kontrollstrukturen wie *for* oder *while* durch Sprungbefehle (gotos) ersetzt.

Die vom Frontend erzeugte IR kann zunächst maschinenunabhängig optimiert werden. Beispielsweise kann der Compiler konstante Ausdrücke erkennen und bereits während der Übersetzung statt zur Laufzeit ausrechnen (**constant folding**). Weitere wichtige IR-Optimierungen dienen der Beschleunigung von Schleifen, z.B. Verschieben von schleifeninvariantem Code (**loop-invariant code motion**), und zur Entfernung von redundantem Code, z.B. Erkennen von Mehrfachberechnungen (**common subexpression elimi-**

nation) und unbenutztem Code (**dead code elimination**). Eine Beschreibung dieser und ähnlicher Techniken findet man in der Compiler-Fachliteratur [10, 11, 12, 13]

Das **Backend** ist der maschinenabhängige Teil des Compilers, der die Abbildung der optimierten IR in Assemblercode leistet. Auch im Backend müssen Optimierungen durchgeführt werden, um möglichst kompakten und/oder schnellen Code zu erzeugen. Leider findet man im Gegensatz zu den IR-Optimierungen in Lehrbüchern relativ wenige Informationen zum Entwurf guter Backends, insbesondere wenn Code für einen Prozessor mit sehr speziellen Eigenschaften (z.B. einen DSP mit parallelen Befehlen und Adressarithmetik) zu erzeugen ist. Hier bleibt einem nichts anderes übrig, als eigene Techniken zu entwickeln oder auf neueste Forschungsarbeiten in der Fachliteratur zurückzugreifen.

Die Grundstruktur eines Backends folgt allerdings meist einem festen Schema: In der **Codeselektion** werden zunächst die IR-Befehle in (möglichst kurze) Folgen von Assemblerbefehlen umgewandelt. Die Basistechnik hierfür ist die Überdeckung von Ausdrücken durch Muster, die jeweils einzelne Assemblerbefehle repräsentieren. Die Codeselektion berücksichtigt der Einfachheit halber allerdings noch nicht die beschränkte Anzahl verfügbarer Register. Dies wird erst in der **Registerallokation** vorgenommen. Ausgehend von "unendlich" vielen Registern wird die Anzahl tatsächlich benutzter Register schrittweise reduziert. Techniken zur Registerallokation basieren oft auf dem Modell der Graphfärbung: Zwei Variablen, die gleichzeitig benötigt werden, dürfen nicht dasselbe Register belegen. Ihnen werden im Graphmodell dann verschiedene "Farben" (d.h. Registernummern) zugewiesen.

Die letzte Backend-Phase ist das **Scheduling**, welches die zeitliche Abfolge der Assemblerbefehle festlegt. Das Ziel ist es, die Ausführungsgeschwindigkeit durch Umordnung von Befehlen (zur Vermeidung von Pipeline-Konflikten) oder Parallelisierung (bei Prozessoren mit mehreren Funktionseinheiten) zu maximieren. Hierfür existieren verschiedene Verfahren, z.B. "list scheduling", die ebenfalls von einem Graphmodell ausgehen.

Der im Backend erzeugte Assemblercode kann anschliessend mittels Assembler und Linker zu einem ausführbaren Programm weiterverarbeitet werden.

Tools zur Compilerentwicklung

Welche Tools sind verfügbar, um die Compilerentwicklung entsprechend der o.g. Phaseneinteilung zu erleichtern ? Zum einen kann man auf retargierbare Compiler zurückgreifen, von denen einige bereits genannt wurden. Der Aufwand ist gering, da "nur" ein passendes Prozessormodell entwickelt werden muss. Allerdings gibt es Probleme, wenn der Compiler auf eine Prozessorklasse angepasst werden soll, für die er ursprünglich nicht entworfen wurde.

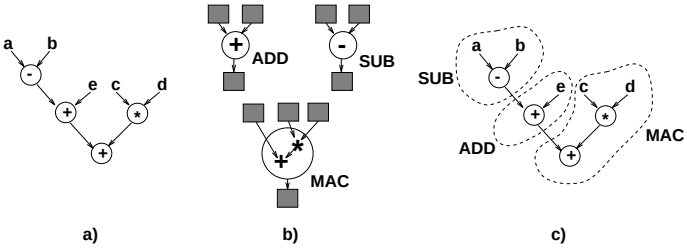

Abbildung 2: *Codeselektion durch Baumüberdeckung*

Ein bekannter retargierbarer und frei verfügbarer Compiler ist der GNU **C-Compiler** [14], welcher für Workstations sehr guten Code generiert. Verschiedene Versuche, den GNU Compiler z.B. auf DSPs anzupassen, müssen aufgrund der schlechten Codequalität allerdings als gescheitert angesehen werden. Der Grund ist, dass sich prozessorspezifische Merkmale oft nur schlecht in das starre Modell eines retargierbaren Compilers hineinpressen lassen.

Betrachtet man Tools zur Realisierung der einzelnen Compilerphasen, so gibt es eine Reihe von Möglichkeiten. Für den **Frontend-Entwurf** gibt es Standardwerkzeuge wie LEX und YACC. Diese lesen eine Grammatik (z.B. für die Sprache C) ein und generieren C-Code für die lexikalische und syntaktische Analyse, welcher leicht in einen Compiler eingebunden werden kann. Ebenso gibt es auch fertige kommerzielle C-Frontends, z.B. von EDG [15], welche allerdings recht kostspielig sind.

Wiederverwendbare Tools zur **IR-Optimierung** existieren praktisch nicht, da es keine einheitlichen IR-Formate gibt. Verwendet man Komplettlösungen wie den GNU Compiler oder das COSY-System von ACE [2], so sind einige IR-Optimierungen bereits integriert, aber man hat die bereits genannten Probleme bei der Backend-Entwicklung. Im universitären Bereich ist das SUIF-System [16] weit verbreitet, welches sich allerdings z.Zt. in einer Redesign-Phase befindet.

Leistungsfähige Tools existieren insbesondere für den **Backend-Bereich**. Ein Beispiel ist IBURG [17], mit dessen Hilfe die Entwicklung eines Codeselektors automatisiert werden kann. Das Tool liest eine Befehlssatzbeschreibung des Prozessors, für den Code erzeugt werden soll, und generiert daraus (ähnlich zu LEX und YACC) C-Code für einen Codeselektor, der in den Compiler eingebunden werden kann. Die Funktionsweise ist in Abb. 2 dargestellt. Enthält z.B. ein C-Programm den Ausdruck $a - b + e + c * d$, so wird nach Erzeugung der IR durch das Frontend zunächst ein entsprechender Ausdrucksbaum aufgebaut (Abb. 2 a). Verfügt der Prozessor über die Befehle ADD, SUB und MAC (multiply-accumulate), siehe Abb. 2 b), so könnte der Codeselektor eine Überdeckung des Ausdrucksbaums wie in Abb. 2 c) gezeigt ausgeben. Der IR-Code wurde somit in Assemblercode übersetzt.

Da jedem Befehl ein Kostenmaß wie Größe oder Ausführungsdauer zugeordet werden kann, garantiert IBURG sogar eine optimale Überdeckung und damit optimalen Code für Ausdrucksbäume. Darüber hinaus ist der erzeugte Codeselektor auch für große Ausdrucksbäume sehr schnell und damit für den praktischen Einsatz geeignet. Weiterentwicklungen von IBURG erlauben auch die Integration weiterer Phasen, d.h. Registerallokation und Scheduling.

Ein Beispiel: Compilerentwicklung mit dem LANCE-2 System

Das an der Universität Dortmund entwickelte Compilersystem LANCE-2 nutzt verfügbare Tools zur effizienten Compilerentwicklung für Forschung und Industrie. Eine Demoversion ist per WWW für Solaris, Linux und Windows 95 frei verfügbar [18]. LANCE-2 verfügt über ein **ANSI C Frontend**, welches den C-Quellcode zunächst in eine IR in Form von Drei-Adress-Code übersetzt. Das Frontend wurde mit Hilfe des Tools OX implementiert, eine Weiterentwicklung von LEX und YACC, die auch die Entwicklung der semantischen Analyse automatisieren kann.

Eine Besonderheit von LANCE-2 ist, dass die **IR selbst wieder in C** dargestellt wird, allerdings auf sehr niedriger, maschinennaher Ebene. Dies dient vor allem der Sicherstellung der Korrektheit: Die generierte IR kann genau wie der ursprüngliche C-Quellcode mit einem auf der Entwicklungsplattform verfügbaren normalen C-Compiler übersetzt werden. Somit kann die Übereinstimmung zwischen Quellcode und IR durch Ausführung leicht überprüft werden. Die Wahl von C als IR erleichtert auch die Einarbeitung in das IR-Format für neue Entwickler, da die IR im Gegensatz zu spezifischen Formaten unmittelbar verständlich ist. Nachfolgend ist ein Beispiel angegeben. Für das C-Programm

```
int A[10],B[10],C[10];

void f()
{
  int i;
  for (i=0; i<10; i++)
   A[i] = B[i] + C[i];
}
```

wird die IR aus Abb. 3 generiert. Das LANCE-2 Frontend generiert eine Reihe von Hilfsvariablen (mit ”t” beginnend) und fügt Labels und Sprünge ein, um Schleifen aufzulösen. Die Entsprechung zwischen Source- und IR-Code wird mit Hilfe von C-Kommentaren für Debugzwecke dargestellt.

Der Zugriff auf die IR wird durch ein **Application Program Interface** (API) in Form einer C++-Library ermöglicht. Hiermit lassen sich IR-Files einlesen, manipulieren und wieder zurückschreiben. LANCE-2 umfasst neben dem C-Frontend eine Reihe von **IR-Optimierungen**,

```
int A[10],B[10],C[10];

void f()
{
  int i_6,t1,t3,*t7,*t11,t12,*t16,t18;
  char *t4,*t6,*t8,*t10,*t13,*t15;

 /* 8 "test.c" */
 /* for (i=0; i<10; i++) */

  i_6 = 0;

  LL3:

 /* 9 "test.c" */
 /* A[i] = B[i] + C[i]; */

  t6 = (char *)B;
  t18 = i_6 * 4;
  t4 = t6 + t18;
  t7 = (int *)t4;
  t10 = (char *)C;
  t8 = t10 + t18;
  t11 = (int *)t8;
  t12 = *t7 + *t11;
  t15 = (char *)A;
  t13 = t15 + t18;
  t16 = (int *)t13;
  *t16 = t12;

 /* 8 "test.c" */
 /* for (i=0; i<10; i++) */

  t3 = i_6 + 1;
  i_6 = t3;
  t1 = t3 < 10;
  if (t1) goto LL3;

}
```

Abbildung 3: *Intermediate representation (IR) in* LANCE-*2*

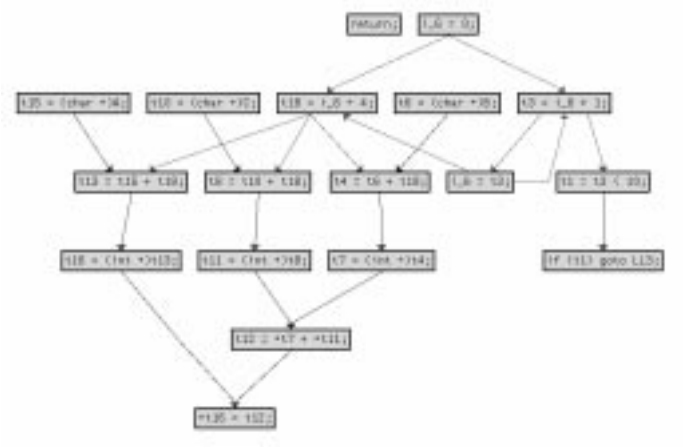

Abbildung 4: *Datenflussgraph des obigen C-Programms*

die sich der Libraryfunktionen bedienen. Weitere IR-Optimierungen lassen sich einfach per ”plug-and-play” einbauen, da alle Optimierungen als separate Tools implementiert sind und auf exakt dem gleichen IR-Format arbeiten. Hierdurch lässt sich wie auch beim C-Frontend der Korrektheitstest der IR-Optimierungen unabhängig vom Backend durchführen.

Prozessorspezifische Backends arbeiten ebenfalls auf der LANCE-2 Library. Zum Aufbau der für die Codeselektion benötigten Ausdrucksbäume (siehe Abb. 2) stellt das API Funktionen zur **Kontroll- und Datenflussanalyse** bereit, welche die Programmstruktur für das Backend sichtbar machen. Mit Hilfe eines weiteren Tools (VCG [19]) können die Ergebnisse auch für Debug-Zwecke visualisiert werden (Abb. 4). Für die eigentliche Backend-Entwicklung wird häufig das Tool OLIVE eingesetzt, eine verbesserte Version des o.g. IBURG. OLIVE bietet die Möglichkeit, die bei der Codeselektion zu Grunde gelegten Befehlskosten dynamisch festzulegen, was z.B. bei Spezialprozessoren mit mehreren Registerfiles wie z.B. DSPs nützlich ist. Des weiteren kann die Ausgabe von Assemblercode direkt an die ausgewählten Befehle gekoppelt werden. Hierdurch wird das Backend modular und damit leichter wartbar.

Das LANCE-2 System wurde bereits für eine Reihe von Forschungsprojekten eingesetzt. Momentan wird ein C-Compiler für den ARM RISC-Prozessor [20] entwickelt, der den speziellen Low Power-Anforderungen mobiler Embedded Systems genügen soll. Daneben wird LANCE-2 in kommerziellen C-Compilerprojekten eingesetzt, die vom Informatik Centrum Dortmund (ICD) im Industrieauftrag durchgeführt werden.

Fazit

Angesichts steigender Komplexität von Embedded Systems und den darin verwendeten Prozessoren führt zukünftig wohl kein Weg mehr an der Verwendung von Compilern vorbei. Nachdem Embedded Systems lange Zeit vorwiegend in Assembler programmiert wurden, ist C als Programmiersprache logischerweise der nächste Schritt. Zukünftig könnten auch C++ oder Java interessant werden, sind aber momentan noch mit zuviel Overhead für den Embedded-Bereich behaftet.

Da in Embedded Systems oft applikationsspezifische Prozessoren zum Einsatz kommen, müssen z.Zt. viele C-Compiler neu entwickelt werden. Um den Aufwand zu minimieren, ist es empfehlenswert, die verfügbaren Entwicklungstools soweit wie möglich auszunutzen. Bei Komplettlösungen wie retargierbaren Compilern muss man u.U. Abstriche an der Codequalität in Kauf nehmen, was vor allem bei zeitkritischen Anwendungen ungünstig ist. Erfolgversprechender ist es, für einen konkreten Prozessor die jeweils besten Tools und Optimierungstechniken auszuwählen und mit begrenztem Aufwand zu einem vollständigen Compiler zu kombinieren.

Literatur

[1] Open SystemC Initiative: www.systemc.org, 2000

[2] ACE Associated Compiler Experts: www.ace.nl

[3] Target Compiler Technologies: www.retarget.com

[4] Archelong Inc.: www.archelon.com

[5] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997

[6] C Level Design Inc.: www.cleveldesign.com, 2000

[7] Tensilica: www.tensilica.com

[8] R. Leupers: *Schneller Code statt schnelle Compiler – Neuartige Code-Optimierungen für digitale Signalprozessoren*, Elektronik Nr. 22, 1999

[9] Plum Hall Inc., www.plumhall.com, 2000

[10] A.V. Aho, R. Sethi, J.D. Ullman: *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986

[11] R. Wilhelm, D. Maurer: *Compiler Design*, Addison-Wesley, 1995

[12] S.S. Muchnik: *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997

[13] A.W. Appel: *Modern Compiler Implementation in C*, Cambridge University Press, 1998

[14] Free Software Foundation: www.gnu.org, 2000

[15] Edison Design Group: www.edg.com, 2000

[16] The Stanford Compiler Group: suif.stanford.edu, 2000

[17] IBURG Download: www.cs.princeton.edu/software/iburg

[18] WWW: ls12-www.cs.uni-dortmund.de/∼leupers

[19] G. Sander: *VCG – Visualization of Compiler Graphs*, ftp.cs.uni-sb.de/pub/graphics/vcg

[20] ARM home page: www.arm.com, 2000