

Kapitel 5

Korrektheit der Codeerzeugung und der Assemblierung

Wolf Zimmermann

Verifikation von Übersetzern

Wolf Zimmermann

247

5.1 Einleitung

Aufgaben der Codeerzeugung

- Transformation des Zwischencodes nach Binärcode
- Ggf. Vorbereiten zum Binden bei getrennter Übersetzung
- ☞ Wird hier nicht betrachtet.

Vorgehen

Ausgangspunkt:

- Menge von Grundblockgraphen: Pro Prozedur und Hauptprogramm jeweils ein Grundblockgraph (Sicht BB)
- ☞ Zuordnung ist über Definitionstabelle bekannt
- Jeder Grundblock ist eine Folge von Bäumen (Sicht BB_T)
 - Zuweisung (ST-Befehle), SAVE, RESTORE, PUSH, POP, JMP, BEQ etc., CALL und RET sind Wurzeln von Bäumen
 - Die Bäume sind über die Use-Def-Beziehungen definiert

CodeSelektion: Für alle Nicht-Sprungbefehle wird der komplette Binärcode selektiert, die Grundblockgraphenstrukturen bleiben erhalten

Assemblierung: Linearisierung der Menge der Grundblockgraphen und Auflösung der Sprünge

Wolf Zimmermann

249

Inhalt

Ziele

- Kennenlernen der termersetzungs-basierten Codeerzeugung
 - Verifikation der Codeerzeugung
 - Bewusstsein für die Problematik des Zusammenwirkens von Registerzuteilung und Codeerzeugung
 - Verifikation der Assemblierung
- 1 Codeerzeugung durch Termersetzung
 - 2 Codeerzeugung durch Termersetzung
 - 3 Verifikation von termersetzungs-basierter Codeerzeugung
 - 4 Überprüfung der Korrektheit der Codeerzeugung
 - 5 Verifikation der Assemblierung

Wolf Zimmermann

248

Methoden der CodeSelektion

Makroexpansion

- Betrachtet jeden Zwischensprachbefehl als Folge von Maschinenbefehlen
- Register werden durch abstrakte Interpretation *on-the-fly* zugeteilt
- Ggf. wird *spill-code* erzeugt
- Einfach, aber liefert ineffizienten Code

Entscheidungstabellen

- Wie Makroexpansion, aber über Entscheidungstabellen werden Bedingungen für Alternativen definiert
- ☞ Besser als Makroexpansion, aber sehr aufwändig

Termersetzungs-systeme

- Termersetzungs-system, das Bäume auf den Term • reduzieren
- Pro TES-Regel wird Code angegeben, der bei Anwendung erzeugt wird
- Ausnutzung algebraischer Identitäten ist möglich
- Registerzuteilung kann vorher geplant werden oder nachdem alle Regeln angewandt wurden erfolgen
- ☞ Wird hier betrachtet (mit Planung der Registerzuteilung vorher)

Wolf Zimmermann

250

Diskussion

Beobachtungen

- Die Codeerzeugung verwendet die Sicht BB_T
- Die Optimierung verwendet häufig die Sicht BB
- ⇒ ASM-Semantik auf diesen Sichten
- ⇒ Korrektheitsbeweise für Sichtwechsel

Registersicherung und –wiederherstellung

- Reihenfolge der zu sichernden Register ist dem Übersetzer bekannt
- ☞ Analog spill-code
- ⇒ Übersetzer kann alten Zustand wiederherstellen
- ⇒ Registernummern müssen nicht mitgesichert werden
- ⇒ Optimierung
- ☞ Diese kann bereits auf IL -Ebene als korrekt nachgewiesen werden

Fazit

- Nachweis der Korrektheit der optimierten Zustandssicherung und -wiederherstellung
- Nachweis der Korrektheit des Sichtwechsels $IL \leftrightarrow BB$
- Nachweis der Korrektheit des Sichtwechsels $BB \leftrightarrow BB_T$

Korrektheitsnachweise

Beweisidee für Korrektheit $BB \leftrightarrow BB_T$

- Grundblockstruktur bleibt erhalten
- Definition einer abgeleiteten dynamischen Funktion $eval$, die Ausdrücke auswertet
- Semantik der Befehle werten Argument mit $eval$ aus.
- Die Reihenfolge der Befehle ST, SAVE, RESTORE, PUSH, POP, JMP, BEQ etc., CALL und RET in einem Grundblock bleibt erhalten
- Jedes Register r in BB entspricht einem (Unter-)Baum $expr_r$ in BB_T (und umgekehrt)
- Für alle Register r mit nicht-lebendigen Werten an diesen Stellen in BB gilt $\llbracket val(r) \rrbracket_B = \llbracket eval(expr_r) \rrbracket_{BT}$
- Mit dieser Funktion wird eine n -1-Simulation definiert

Beispiel 5.1: Regel zum Speichern in BB_T

if CT is ST then

$Store(eval(Opd_1), eval(Opd_2))$	$eval(const(x))$	$= x$
	$eval(mkload(x))$	$= Read(eval(x))$
	$eval(mkloada(x))$	$= BP +_I eval(x)$
$Proceed$	$eval(mkloadg(x))$	$= UP +_I eval(x)$
	$eval(x \oplus_I y)$	$= eval(x) +_I eval(y)$

Korrektheitsnachweise

Beweisidee zur Optimierung der Sicherung

- Definition des Laufzeitkellers der optimierten Version und optimierten analog Kapitel 3
- Vergleich der entsprechenden Aktivierungsverbände auf dem Laufzeitkeller
- ☞ Identisch für den Speicher außer ungesicherten Variablen (wenn auch an unterschiedlichen Adressen)
- ☞ Vergleich der optimierten gesicherten Register mit den unoptimierten gesicherten Registern

Beweisidee für $IL \leftrightarrow BB$

Bijektive Zuordnung zwischen Befehlsnr. in Befehlssequenz und Befehlsnummer in einem Grundblock:

- Befehlszeiger im Grundblockgraph ist ein Paar aus Grundblocklabel und Befehlsnr.
- Mit $\llbracket ip \rrbracket_Z = \llbracket labtobef \rrbracket_Z(semprog_{iZ}, \llbracket bb(ip_B) \rrbracket_B) + \llbracket num(ip_B) \rrbracket_B$ wird eine 1-1-Simulation zwischen BB und IL definiert.

5.2 Termersetzungssysteme

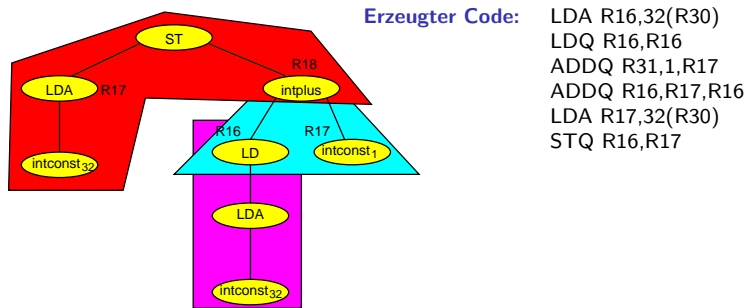
Beobachtungen (Weingart, 1973)

- Jeder aus Ausdrucksbaum e erzeugte Befehl b deckt Teil des Baums ab
- Gesamtcode überdeckt Gesamtbaum überlappungsfrei

Idee

- Jeder Ausdrucksbaum ist Term einer Termalgebra \mathfrak{T}
 - Maschinenbefehle sind Terme einer anderen Termalgebra \mathfrak{T}'
- ⇒ Ersetze Ausdrucksbaum $e \in \mathfrak{T}$ durch Befehlssequenz $b \in \mathfrak{T}'$

Beispiel 5.2: Codeerzeugung mit TES



Erzeugter Code: LDA R16,32(R30)
LDQ R16,R16
ADDQ R31,1,R17
ADDQ R16,R17,R16
LDA R17,32(R30)
STQ R16,R17

Regeln: $intconst_j \rightarrow reg_i$ $\{ADDQ R31, j, R_i\}$ falls $-128 \leq j \leq 127$
 $intplus(reg_j, reg_k) \rightarrow reg_i$ $\{ADDQ R_j, R_k, R_i\}$
 $LD reg_j \rightarrow reg_i$ $\{LDQ R_i, R_j\}$
 $LDA intoconst_j \rightarrow reg_i$ $\{LDA R_i, j(R30)\}$ falls $-2^{15} \leq j < 2^{15}$
 $ST reg_i, reg_j \rightarrow \bullet$ $\{STQ R_i, R_j\}$

Weitere Regeln: $intplus(reg_j, intconst_k) \rightarrow reg_i$ $\{ADD R_j, k, R_i\}$ falls $-2^{15} \leq k < 2^{15}$
 $LD (LDA intoconst_j) \rightarrow reg_i$ $\{LDQ R_i, (k)R_j\}$ falls $-128 \leq j \leq 127$
 $ST (LDA intoconst_j) \rightarrow reg_i$ $\{STQ R_i, (k)R_j\}$ falls $-128 \leq j \leq 127$

Erzeugter Code: LDA R16(32),R30
ADDQ R16,1,R18
STQ R18,(32)R30

5.3 Verifikation von termersetzungsbasierter Codeerzeugung

Beobachtung

- Grundblockstruktur bleibt erhalten
- ⇒ Bei Simulation muss nur gezeigt werden, dass der aus einer Regel erzeugte Code dem der Quelle entspricht
- ⇒ Befehlszeiger wird nur erhöht

Vorgehen

- Formalisierung der DEC-Alpha-Semantik
- Klassifikation der Simulationsbeweise
- Beispiele

Diskussion

Beobachtungen

- Optimierungen durch bessere Regelabdeckungen
- ✚ Keinen Einfluss auf Korrektheit
- Registerzuteilung und Reihenfolge der Regelanwendungen muss mit dem Ermitteln von Regelabdeckungen geplant werden
- Für Korrektheit ist wichtig, dass kein Register zum Schreiben zugeteilt wird, das einen lebendigen Wert enthält
- Spill-Code kann mit integriert werden
- Werkzeuge erlauben Vollständigkeitsprüfung
- Trotz Notation können bereits binär codierte Befehle erzeugt werden
- Grundblockgraphstruktur bleibt erhalten
- ⇒ Grundblöcke enthalten bis auf Sprünge bereits binär codierte Maschinenbefehle (BB_α)
- Semantik von BB_α enthält jetzt nur noch die Register der DEC-Alpha und ersetzt die Zustandsübergangsregeln von BB_T durch die der DEC-Alpha. Bei Sprüngen wird auf Register statt auf Ausdrücke zugegriffen.

Fazit

Für die Korrektheit der Transformation $BB_T \rightarrow BB_\alpha$ muss nur die Korrektheit der TES-Regeln nachgewiesen und die Korrektheit der Annotationen geprüft werden.

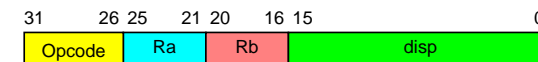
Formalisierung der DEC-Alpha-Semantik

Ausgangspunkt

Binärformat der Befehle

- Formalisierung der Zugriffsfunktionen und Befehlsklassifikation (als statische Funktionen)
- Makros zur Befehlszuordnung
- Formalisierung der Registertransfers

Binärformat für Speicherzugriffsbefehle



$opcode : QUAD \rightarrow BITSEQ_6 \quad opcode(b) \doteq sel_{32}^{31,26}(b)$
 $opd_1 : QUAD \rightarrow BITSEQ_5 \quad opd_1(b) \doteq sel_{32}^{25,21}(b)$
 $opd_2 : QUAD \rightarrow BITSEQ_5 \quad opd_2(b) \doteq sel_{32}^{20,16}(b)$
 $disp : QUAD \rightarrow BITSEQ_{16} \quad disp(b) \doteq sel_{32}^{15,0}(b)$

Für lange Sprünge wird dasselbe Format benutzt, nur dass $disp$ für Hinweise zur Sprungvorhersage verwendet wird.

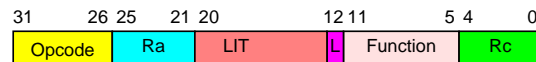
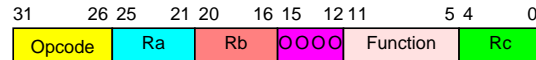
Formalisierung des Binärcodes

Binärformat für relativen Sprungbefehle



$jump : \text{QUAD} \rightarrow \text{BITSEQ}_{21} \quad jump(b) \doteq sel_{32}^{20,0}(b)$

Binärformat für arithmetische und logische Operationen



$dest : \text{QUAD} \rightarrow \text{BITSEQ}_5 \quad dest(b) \doteq sel_{32}^{4,0}(b)$
 $constop : \text{QUAD} \rightarrow \text{BIT} \quad constop(b) \doteq bit_{12}(b)$
 $lit : \text{QUAD} \rightarrow \text{BITSEQ}_8 \quad lit(b) \doteq sel_{32}^{20,13}(b)$
 $fnctn : \text{QUAD} \rightarrow \text{BITSEQ}_6 \quad fnctn(b) \doteq sel_{32}^{11,5}(b)$

Makros zur Befehlserkennung

Grundprinzip

- Befehlszeiger bz verweist auf Befehl
- Klassifikation über $opcode$ und ggf. $fnctn$

Beispiel 5.3: Klassifikation von Befehlen

$Bef \triangleq Read(bz)$
 $Cl \text{ is LDQ} \triangleq opcode(Bef) \doteq L O L O O L$
 $Cl \text{ is STQ} \triangleq opcode(Bef) \doteq L O L L O O$
 $Cl \text{ is BEQ} \triangleq opcode(Bef) \doteq L L L O O L$
 $Cl \text{ is BR} \triangleq opcode(Bef) \doteq L L O O O O$
 $Cl \text{ is ADDQ} \triangleq opcode(Bef) \doteq O L O O O O \wedge fnctn(Bef) \doteq L O O O O O O \wedge bit_{12}(Bef) \doteq O$
 $Cl \text{ is ADDQI} \triangleq opcode(Bef) \doteq O L O O O O \wedge fnctn(Bef) \doteq L L O O O O O \wedge bit_{12}(Bef) \doteq L$
 $Cl \text{ is SUBQ} \triangleq opcode(Bef) \doteq O L O O O O \wedge fnctn(Bef) \doteq O L O L O O L \wedge bit_{12}(Bef) \doteq O$
 $Cl \text{ is MULQ} \triangleq opcode(Bef) \doteq O L O O L L \wedge fnctn(Bef) \doteq O L O O O O O \wedge bit_{12}(Bef) \doteq O$

Zustandsraum der DEC Alpha

Zustandsraum DEC-Alpha

$spec \text{ STATE}_\alpha \text{ extends VALUE}_\alpha$
 $sorts \text{ BYTELIST}$
 $operations \quad mem_\alpha : \text{QUAD} \rightarrow \text{QUAD}$
 $in_\alpha : \rightarrow \text{BYTELIST}$
 $out_\alpha : \rightarrow \text{BYTELIST}$
 $fpcr : \rightarrow \text{QUAD}$
 $reg : \text{BITSEQ}_5 \rightarrow \text{QUAD}$
 $freg : \text{BITSEQ}_5 \rightarrow \text{QUAD}$
 $bz : \rightarrow \text{QUAD}$
 $axioms \quad reg(\text{LLLLL}) \doteq O^{64}$
 $freg(\text{LLLLL}) \doteq O^{64}$

Zustandsraum BB_α

$spec \text{ STATE}_\alpha^{BB} \text{ extends VALUE}_\alpha$
 $sorts \text{ BYTELIST}$
 $operations \quad mem_\alpha : \text{QUAD} \rightarrow \text{QUAD}$
 $in_\alpha : \rightarrow \text{BYTELIST}$
 $out_\alpha : \rightarrow \text{BYTELIST}$
 $fpcr : \rightarrow \text{QUAD}$
 $reg : \text{BITSEQ}_5 \rightarrow \text{QUAD}$
 $freg : \text{BITSEQ}_5 \rightarrow \text{QUAD}$
 $prog_{BB_T} : \rightarrow \text{PROG}_{BB_T}$
 $axioms \quad reg(\text{LLLLL}) \doteq O^{64}$
 $freg(\text{LLLLL}) \doteq O^{64}$
 $IP_{BB_T} \rightarrow IP_{BB_T}$

Zustandsübergangsregeln

Grundprinzip

Formalisierung der Registertransfers der Maschinenspezifikation als ASM-Aktualisierungen

Beispiel 5.4: Zustandsübergangsregeln in DEC-Alpha

Makros:
 $Opd_i \triangleq reg(opd_i(Bef))$
 $Disp \triangleq disp(Read(Bef))$
 $Lit \triangleq lit(Read(Bef))$
 $Proceed_\alpha \triangleq bz := bz + 1 \ 4$
Statische Funktion:
 $sext_i : \text{BITSEQ}_i \rightarrow \text{QUAD}$
 $sext_i(O : b) \triangleq O^{65-i} ++ b$
 $sext_i(L : b) \triangleq L^{65-i} ++ b$

if Cl is LDQ then
if $\neg opd_1(Bef) \doteq \text{LLLLL}$ then
 $Opd_1 := Read(Opd_2 + sext_8(Disp))$
 $Proceed_\alpha$
if Cl is LDA then
if $\neg opd_1(Bef) \doteq \text{LLLLL}$ then
 $Opd_1 := Opd_2 + sext_8(Disp)$
 $Proceed_\alpha$
if Cl is STQ then
 $Store(Opd_2 + sext_8(Disp, Opd_1))$
 $Proceed_\alpha$
if Cl is ADDQ then
if $\neg opd_3(Bef) \doteq \text{LLLLL}$ then
 $Opd_3 := Opd_1 + Opd_2$
 $Proceed_\alpha$
if Cl is ADDQI then
if $\neg opd_3(Bef) \doteq \text{LLLLL}$ then
 $Opd_3 := Opd_1 + sext_{16}(Lit)$
 $Proceed_\alpha$
if Cl is BR then
 $Opd_1 := bz$
 $bz := bz + 1 \ Sext_{32}(4 * 1 \ Disp)$
if Cl is BEQ then
if $Opd_1 \doteq O^{64}$ then
 $bz := bz + 1 \ Sext_{32}(4 * 1 \ Disp)$
else Proceed

Simulationsbeweise für Korrektheit von TES-Regeln

Statische Funktionen

$regmap : REGISTER \rightarrow ?BITSEQ_5$ Registerzuteilung
 $fregmap : REGISTER \rightarrow ?BITSEQ_5$ Registerzuteilung
 $rule : IP_{BB_T} \rightarrow ?RULE$ angewandte Regel
 $sched : IP_{BB} \rightarrow ?IP_{BB}$ nächste Stelle mit Regelanwendung

Anforderung 5.1 (Registerzuteilung)

- Es dürfen keine globalen Register zugeteilt werden (R14,R15,R29,R30)
- Register R31 und F31 dürfen nicht zugeteilt werden
- Register $regmap(ip_{BB_T})$ darf zum Zeitpunkt des Beschreibens keinen lebendigen Wert enthalten.

Klassifikation der Simulationsbeweise

Beobachtung

Es gibt zwei Arten von Termersetzungsregeln:

- $t \rightarrow reg_i$ übersetzt einen echten Unterbaum. Dessen Wert muss in Register reg_i sein.
- $t \rightarrow \bullet$ übersetzt einen Befehl (z.B. ST) in einen entsprechenden Maschinenbefehl

Diskussion

- Bei der ersten Art von Regel wird nur reg oder $freg$ verändert.
- Sei \mathfrak{B}' Zustand nach dem Maschinencode. Dort muss gelten:
 $\llbracket eval \rrbracket_{\mathfrak{B}}(t) \doteq \llbracket reg \rrbracket_{\alpha}(\llbracket regmap(t) \rrbracket_B)$ bzw.
 $\llbracket eval \rrbracket_{\mathfrak{B}}(t) \doteq \llbracket freg \rrbracket_{\alpha}(\llbracket regmap(t) \rrbracket_B)$
- Bei der zweiten Art von Regel wird nur mem_{α} , die globalen Register, das Statusregister oder die Ein-/Ausgabeströme verändert.
- Zu Beginn sind alle Operanden ausgewertet und die den Operanden zugeordneten Register enthalten deren Werte im Zustand vor den erzeugten Befehlen.

Definition der Simulationsrelation

Sei \mathcal{B}_T die ASM für BB_T und \mathcal{B}_{α} die ASM für BB_{α} . Es ist $(\mathfrak{B}_{\alpha}, \mathfrak{B}_T) \in \rho$ gdw.:

- Speicher, Ein- und Ausgabestrom sowie Statusregister sind identisch
- Die lebendigen Inhalte der Register sowie der global verwendeten Register sind identisch mit den Werten der entsprechenden Teilausdrücke
- Die Befehlszeiger sind immer im Bildbereich von ρ . Sie sind immer Befehlszeigern im selben Grundblock zugeordnet, die den Anfang bzw. Ende der Übersetzung eines BB_T -Befehls betreffen.
- Dazu sei $size : PROG_{BB_T} \times IP_{BB_T} \rightarrow INT$ die Anzahl der bis zu einem Befehl generierten Befehle

$$\begin{aligned}
 \llbracket mem_{\alpha} \rrbracket_T &= \llbracket mem_{\alpha} \rrbracket_{\alpha} & (1) \\
 \llbracket inp_{\alpha} \rrbracket_T &= \llbracket inp_{\alpha} \rrbracket_{\alpha} & (2) \\
 \llbracket out_{\alpha} \rrbracket_T &= \llbracket out_{\alpha} \rrbracket_{\alpha} & (3) \\
 \llbracket fpcr \rrbracket_T &= \llbracket fpcr \rrbracket_{\alpha} & (4) \\
 \llbracket eval(t) \rrbracket_T &= \llbracket reg \rrbracket_{\alpha}(\llbracket regmap(t) \rrbracket_B) & \text{ falls } \llbracket isLeb \rrbracket(eval(t), IP_T) & (5) \\
 \llbracket BP \rrbracket_T &= \llbracket R30 \rrbracket_{\alpha} & (6) \\
 \llbracket SP \rrbracket_T &= \llbracket R29 \rrbracket_{\alpha} & (7) \\
 \llbracket HP \rrbracket_T &= \llbracket R15 \rrbracket_{\alpha} & (8) \\
 \llbracket UP \rrbracket_T &= \llbracket R14 \rrbracket_{\alpha} & (9) \\
 \llbracket eval(t) \rrbracket_T &= \llbracket freg \rrbracket_{\alpha}(\llbracket fregmap(t) \rrbracket_B) & \text{ falls } \llbracket isLeb \rrbracket(eval(t), IP_T) & (10) \\
 \llbracket ip \rrbracket_T &\in \text{RAN}(\rho) & \text{ falls } \mathfrak{B}_T \models D(\text{getbef}_T(\text{prog}_T, ip)) & (11) \\
 \llbracket ip \rrbracket_{\alpha} &= \langle L, k \rangle & \text{ falls } \llbracket ip \rrbracket_T = \langle L, i \rangle \text{ und } \text{sembssize}_T(i) \leq k < \llbracket bbsize \rrbracket_{\mathcal{B}_T}(i+1) & (12)
 \end{aligned}$$

Korrektheit

Satz 5.2 (Korrektheit der TES-basierten Übersetzung)

Sei \mathcal{B}_T und \mathcal{B}_{α} die ASMs, die die Semantik von BB_T bzw. BB_{α} definieren

- \mathfrak{B}_T und \mathfrak{B}_{α} Zustände von \mathcal{B}_T bzw. \mathcal{B}_{α} , so dass $(\mathfrak{B}_{\alpha}, \mathfrak{B}_T) \in \rho$
- $t \in \text{BEF}_T$ mit $\llbracket IP \rrbracket_B = t$
- t' ein echter Unterterm von t

Wenn die folgenden beiden Eigenschaften

- Wenn $t'[reg_1, \dots, reg_k] \rightarrow reg_i\{m_1; \dots; m_n\}$ eine angewandte TES-Regel ist, \mathfrak{B}'_{α} Zustand mit $\llbracket ip \rrbracket_{\alpha} = m_1$ und $(\mathfrak{B}'_{\alpha}, \mathfrak{B}_T) \in \rho$, dann ist auch $(\mathfrak{B}''_{\alpha}, \mathfrak{B}_T) \in \rho$ für den Zustand \mathfrak{B}''_{α} nach Ausführung dieser Befehlssequenz.
- Für die TES-Regel $t[reg_1, \dots, reg_k] \rightarrow \bullet \{m_1; \dots; m_n\}$ wird ein Zustand \mathfrak{B}'_{α} mit $\mathfrak{B}'_{\alpha} \models ip_{\alpha} \doteq m_1$ und $(\mathfrak{B}'_{\alpha}, \mathfrak{B}_T) \in \rho$ in einen Zustand \mathfrak{B}''_{α} mit $(\mathfrak{B}''_{\alpha}, \mathfrak{B}_T) \in \rho$ nach Ausführung dieser Befehlssequenz überführt.

sowie Anforderung 5.1 erfüllt ist, dann definiert ρ eine n - m -Simulation

Beweisskizze

- Induktion über den Aufbau von BB_T -Befehlen beweist eine 1- n -Simulation
- Anwendung von Satz 1.3 (Vertikale Dekomposition)

Beispiel 5.5: Additionsbefehl

TES-Regel: $reg_1 \oplus_I reg_2 \rightarrow reg_3 \{ADDQ reg_1, reg_2, reg_3\}$

Nachweis der Eigenschaft Satz 5.2(i): Sei \mathfrak{B}_T und \mathfrak{B}_α , so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

- ⇒ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand \mathfrak{B}'_α
 - Sei $t = t_1 \oplus t_2$ der Baum, auf den die TES-Regel angewendet wird, $mapreg(t_1) = R_1$, $mapreg(t_2) = R_2$ und $mapreg(t) = R_3$
- ⇒
$$\begin{aligned} \llbracket eval(t) \rrbracket_{B_T} &= \llbracket eval(t_1 \oplus t_2) \rrbracket_{B_T} \\ &= \llbracket \oplus_I \rrbracket_{B_T} (\llbracket eval(t_1) \rrbracket_{B_T}, \llbracket eval(t_2) \rrbracket_{B_T}) \\ &= \llbracket eval(t_1) \rrbracket_{B_T} +_I \llbracket eval(t_2) \rrbracket_{B_T} \\ &= \llbracket reg(R_1) \rrbracket_{B_\alpha} +_I \llbracket reg(R_2) \rrbracket_{B_\alpha} \end{aligned}$$
- Es gilt $Update(\mathcal{B}_A) = \{reg(R_3) := reg(R_1) +_I reg(R_2), Proceed_B\}$
- ⇒
$$\begin{aligned} \llbracket reg(R_3) \rrbracket_{B'_\alpha} &= \llbracket reg(R_1) \rrbracket_{B_\alpha} +_I \llbracket reg(R_2) \rrbracket_{B_\alpha} \\ &= \llbracket eval(t) \rrbracket_{B_T} \end{aligned}$$
- ⇒ $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$

Beispiel 5.7: Lesen vom Speicher

TES-Regel: $LD reg_2 \rightarrow reg_1 \{LDQ reg_1, 0(reg_2)\}$

Nachweis der Eigenschaft Satz 5.2(i): Sei \mathfrak{B}_T und \mathfrak{B}_α , so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

- ⇒ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand \mathfrak{B}'_α
 - Sei $t = LD t_1$ der Baum, auf den die TES-Regel angewendet wird, $mapreg(t_1) = R_1$ und $mapreg(t) = R_2$
- ⇒
$$\begin{aligned} \llbracket eval(t) \rrbracket_{B_T} &= \llbracket eval(LD t_1) \rrbracket_{B_T} \\ &= \llbracket Read \rrbracket_{B_T} (\llbracket eval(t_1) \rrbracket_{B_T}) \\ &= \llbracket Read \rrbracket_{B_\alpha} (\llbracket reg(R_1) \rrbracket_{B_\alpha}) \end{aligned}$$
- Es gilt $Update(\mathcal{B}_A) = \{reg(R_2) := Read(reg(R_1)); Proceed_B\}$
- ⇒
$$\begin{aligned} \llbracket reg(R_2) \rrbracket_{B'_\alpha} &= \llbracket Read \rrbracket_{B_\alpha} (\llbracket reg(R_1) \rrbracket_{B_\alpha}) \\ &= \llbracket eval(t) \rrbracket_{B_T} \end{aligned}$$

Beispiel 5.6: Additionsbefehl mit kleiner Konstante

TES-Regel: $reg_1 \oplus_I intconst_j \rightarrow reg_3 \{ADDQI reg_1, byte_0(j), reg_3\}$ falls $-128 \leq j < 127$

Nachweis der Eigenschaft Satz 5.2(i): Sei \mathfrak{B}_T und \mathfrak{B}_α , so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

- ⇒ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand \mathfrak{B}'_α
 - Sei $t = t_1 \oplus (CONST j)$ der Baum, auf den die TES-Regel angewendet wird, $mapreg(t_1) = R_1$ und $mapreg(t) = R_3$
- ⇒
$$\begin{aligned} \llbracket eval(t) \rrbracket_{B_T} &= \llbracket eval(t_1 \oplus (CONST j)) \rrbracket_{B_T} \\ &= \llbracket \oplus_I \rrbracket_{B_T} (\llbracket eval(t_1) \rrbracket_{B_T}, \llbracket eval((CONST j)) \rrbracket_{B_T}) \\ &= \llbracket eval(t_1) \rrbracket_{B_T} +_I j \\ &= \llbracket reg(R_1) \rrbracket_{B_\alpha} +_I j \end{aligned}$$
- Es gilt $Update(\mathcal{B}_A) = \{reg(R_3) := reg(R_1) +_I sext_8(byte_0(j)), Proceed_B\}$
- ⇒
$$\begin{aligned} \llbracket reg(R_3) \rrbracket_{B'_\alpha} &= \llbracket reg(R_1) \rrbracket_{B_\alpha} +_I \llbracket sext_8(byte_0(j)) \rrbracket_\alpha \\ &= \llbracket eval(t) \rrbracket_{B_T} \end{aligned}$$
- weil $\mathfrak{B}_\alpha \models -128 \leq j < 127 \Rightarrow j \doteq sext_8(byte_0(j))$
- ⇒ $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$

Laden einer kleinen Konstante

$intconst_j \rightarrow reg_1 \{ADDQI R31, byte_0(j), reg_1\}$ falls $-128 \leq j < 127$

Beispiel 5.8: Lesen von kleinen Relativadressen

TES-Regel: $LD (LDA (CONST j)) \rightarrow reg_1 \{LDQ reg_1, \bar{j}(R30)\}$ falls $-2^{15} \leq j < 2^{15}$ und $\bar{j} \triangleq sel_{64}^{15,0}(j)$

Nachweis der Eigenschaft Satz 5.2(i): Sei \mathfrak{B}_T und \mathfrak{B}_α , so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

- ⇒ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand \mathfrak{B}'_α
 - Sei $mapreg(t) = R_1$
- ⇒
$$\begin{aligned} \llbracket eval(t) \rrbracket_{B_T} &= \llbracket eval(LD) \rrbracket_{B_T} (\llbracket BP \rrbracket_{B_T} +_I j) \\ &= \llbracket Read \rrbracket_{B_T} (\llbracket BP \rrbracket_{B_T} +_I j) \\ &= \llbracket Read \rrbracket_{B_\alpha} (\llbracket reg(R30) \rrbracket_{B_\alpha} +_I j) \end{aligned}$$
- Es gilt $Update(\mathcal{B}_A) = \{reg(R_2) := Read(reg(R30) +_I sext_{16}(sel_{64}^{15,0}(j))); Proceed_B\}$
- ⇒
$$\begin{aligned} \llbracket reg(R_2) \rrbracket_{B'_\alpha} &= \llbracket Read \rrbracket_{B_\alpha} (\llbracket reg(R30) \rrbracket_{B_\alpha} +_I \llbracket sext_{16}(sel_{64}^{15,0}(j)) \rrbracket_{B_\alpha}) \\ &= \llbracket eval(t) \rrbracket_{B_T} \end{aligned}$$
- weil $\mathfrak{B}_\alpha \models -2^{15} \leq j < 2^{15} \Rightarrow j \doteq sext_{16}(sel_{64}^{15,0}(j))$

Beispiel 5.9: Laden von Relativadressen

TES-Regel: $LDA (CONST j) \rightarrow reg_1 \{LDA reg_1, \bar{j}\}(R30)$ falls $-2^{15} \leq j < 2^{15}$ und $\bar{j} \triangleq sel_{64}^{15,0}(j)$

Nachweis der Eigenschaft Satz 5.2(i): Sei \mathfrak{B}_T und \mathfrak{B}_α , so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

⇒ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand \mathfrak{B}'_α

• Sei $t = LDA t_1 (CONST j)$ $mapreg(t_1) = R_1$ und $mapreg(t) = R_2$

⇒ $\llbracket eval(t) \rrbracket_{B_T} = \llbracket reg(R_1) \rrbracket_{B_T} +_I j$
 $= \llbracket reg(R_1) \rrbracket_{B_\alpha} + j$

• Es gilt

$Update(B_A) = \{reg(R_2) := reg(R_1) +_I sext_{16}(sel_{64}^{15,0}(j)); Proceed_B\}$

⇒ $\llbracket reg(R_2) \rrbracket_{B'_\alpha} = \llbracket reg(R_1) \rrbracket_{B_\alpha} +_I \llbracket sext_{16}(sel_{64}^{15,0}(j)) \rrbracket_{B_\alpha}$ weil
 $= \llbracket eval(t) \rrbracket_{B_T}$

$\mathfrak{B}_\alpha \models -2^{15} \leq j < 2^{15} \Rightarrow j \doteq sext_{16}(sel_{64}^{15,0}(j))$

Laden einer mittelkleinen Konstante $intconst_j \rightarrow reg_1 \{LDA reg_1, \bar{j}\}(R31)$
 falls $-2^{15} \leq j < 2^{15}$

Beispiel 5.10: Abhilfe

Fehlerursache

- Vorzeichenerweiterung

⇒ Kein Befehl zum Unterdrücken der Vorzeichenerweiterung

if Cl is ZAPI then Statische Funktion $bytezap : QUAD \times$

if $\neg opd_3 \doteq LLLLL$ **then**

$Opd_3 := bytezap(Opd_1, Disp)$

Proceed_α

BYTE \rightarrow QUAD mit $byte_i(bytezap(w, b)) \doteq O^8 \Leftrightarrow bit_i(b) \doteq L$ und
 $byte_i(bytezap(w, b)) \doteq byte_i(w) \Leftrightarrow bit_i(b) \doteq O$

Korrektur

$LDA (CONST j) \rightarrow reg_1 \{$
 $LDA reg_1, J0(R31)$
 $ZAP reg_1, LLLLLLOO, reg_1$
 $LDAH reg_1, J1(reg_1)\}$

Übung

Beweisen Sie nun die Korrektheit

Beispiel 5.10: Laden einer mittleren Konstante

DEC-Alpha-Befehl LDAH: **if** Cl is LDAH **then**

if $\neg opd_1 \doteq LLLLL$ **then**

$Opd_1 := Opd_2 + sext_{16}(Disp * 65536)$

Proceed_α

TES-Regel: $LDA (CONST j) \rightarrow reg_1 \{LDA reg_1, J0(R31); LDAH reg_1, J1(reg_1)\}$ falls $-2^{31} \leq j < 2^{31}$, $J0 \triangleq sel_{64}^{15,0}(j)$ und $J1 \triangleq sel_{64}^{31,16}(j)$

Nachweis der Eigenschaft Satz 5.2(i): Sei \mathfrak{B}_T und \mathfrak{B}_α , so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

⇒ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand \mathfrak{B}'_α

• Sei $t = LDA t_1 (CONST j)$ $mapreg(t_1) = R_1$ und $mapreg(t) = R_2$

⇒ $semeval(t)_{B_T} = j$

• Es gilt

$Update(B_A) = \{reg(R_2) := reg(R_1) +_I sext_{16}(sel_{64}^{15,0}(j)); Proceed_B\}$

⇒ $\llbracket reg(R_1) \rrbracket_{B'_\alpha} = \llbracket reg(R31) \rrbracket_{B_\alpha} +_I \llbracket sext_{16}(sel_{64}^{15,0}(j)) \rrbracket_{B_\alpha}$
 $= \llbracket sext_{16}(sel_{64}^{15,0}(j)) \rrbracket_{B_\alpha}$

⇒ $Update(B'_A) = \{reg(R_2) := reg(R_1) +_I sext_{16}(sel_{64}^{31,16}(j) * 65536); Proceed_B\}$

⇒ $\llbracket reg(R_1) \rrbracket_{B'_\alpha} = \llbracket reg(R_1) \rrbracket_{B'_\alpha} +_I \llbracket sext_{16}(sel_{64}^{31,16}(j) * 65536) \rrbracket_{B_\alpha}$
 $= \llbracket sext_{16}(sel_{64}^{15,0}(j)) \rrbracket_{B_\alpha} + \llbracket sext_{16}(sel_{64}^{31,16}(j) * 65536) \rrbracket_{B_\alpha}$

Problem

Falls $bit_{15}(j) = L$ ist $\llbracket sext_{16}(sel_{64}^{15,0}(j)) \rrbracket_{B_\alpha} +_I \llbracket sext_{16}(sel_{64}^{31,16}(j) * 65536) \rrbracket_{B_\alpha} \neq j$

Fazit

Wenn Anforderung 5.1 erfüllt ist, dann genügt es für jede Transformationsregel die entsprechende Simulation zu berechnen

- Im Prinzip handelt es sich um eine symbolische Auswertung der linken Seite und symbolische Zustandsübergänge auf der rechten Seite

⇒ Es entstehen einfache Gleichungen über Bitfolgen

⇒ Automatisierung möglich (realisiert in PVS)

5.4 Überprüfung der Korrektheit der Codegenerierung

Beobachtung

Anforderung 5.1 muss erfüllt sein, damit die Codeselektionsphase korrekt ist

- Mit den Attribute *regmap*, *fregmap* spezifiziert der Übersetzer die Registerzuteilung, mit dem Attribut *rule* die anzuwendende Regel und mit dem Attribute *sched* wird eine Reihenfolge auf den anzuwendenden Regeln definiert
- ⇒ Wert in einem Register ist lebendig, wenn er mit einer Termersetzungsregel erzeugt wurde, aber noch nicht durch eine Termersetzungsregel verbraucht wurde
- ⇒ Kann statisch geprüft werden
- ⇒ Programmprüfung möglich

Überprüfung der Korrektheit der Anwendung der Transformation

- Überprüfung, ob annotierte Regel anwendbar
- Anwendung der Regeln gemäß *sched*
- Vergleich mit dem Resultat des Compilers

Assemblierung

Transformationsregeln

Kein Sprung: $JMP\ L \rightarrow \bullet \{ \}$ falls

$labtoaddr(CB) +_1 bbsize(CB) +_1 8 \doteq labtoaddr(L)$ und $CB \triangleq first(IP_{BB})$ der aktuelle Block ist.

Kurzer Sprung: $JMP\ L \rightarrow \bullet \{ BR\ R1\ Disp \}$ falls

$labtoaddr(CB) +_1 bbsize(CB) +_1 4 \doteq labtoaddr(L) +_1 sext_{23}(disp * 4)$

Mittlerer Sprung: $JMP\ L \rightarrow \bullet \{ BR\ R1, O^{20}L \}$ falls

LDA $R2, sel_{64}^{15,0}(R31)$
ZAP $R2, LLLLLLOO, R2$
LDAH $R2, sel_{64}^{31,16}(R2)$
JMP $R1, R2 \}$

$labtoaddr(CB) +_1 bbsize(CB) +_1 4 \doteq labtoaddr(L) +_1 sext_{32}(dist)$

Ganz langer Sprung: Muss dann noch zusätzlich Linksshifts einbauen

5.5 Assemblierung

Aufgabe

Abbildung der Grundblöcke in Speicher (mit Relativadressen) und Umsetzen der Sprungstruktur

Vorgehen

- Transformationsregeln für kurze, mittlere und große Sprünge.
- Der Übersetzer bildet die Sprungmarken auf Relativadressen ab: $labtoaddr : LABEL \rightarrow QUAD$
- Die Befehle in den Grundblöcken werden in derselben Reihenfolge im Speicher abgelegt.
- ⇒ Statische Funktion $bbsize : LABEL \rightarrow QUAD$
- Am Ende werden die Sprünge gemäß den obigen Transformationsregeln generiert

Beobachtungen

- Aufeinanderfolgende Grundblöcke die hintereinander angeordnet werden, dürfen sich nicht überlappen
- Kurze Sprünge können als Relativadresse in den Befehl codiert werden
- Mittlere und lange Sprünge erfordern das Laden des Sprungziels in ein Register
- ⇒ Optimierproblem
- Übersetzer berechnet pro Sprungziel in einem Grundblock, ob dieser Sprung eingespart werden kann, ein kurzer, mittelgroßer oder langer Sprung ist
- ⇒ Annotation mit Transformationsregeln

Korrektheit der Assemblierung

Relation ρ

Sei B_α die ASM für die Semantik von BB_α und \mathcal{A} die Semantik der Maschinensprache sowie \mathfrak{B}_α und \mathfrak{A} zwei Zustände von B_α bzw. \mathcal{A} . Es sei $\langle \mathfrak{B}_\alpha, \mathfrak{A} \rangle \in \rho$ gdw.

$$\begin{aligned} \llbracket mem_\alpha \rrbracket_{B_\alpha}(x) &= \llbracket mem_\alpha \rrbracket_{B_\alpha}(x) \text{ für } x \geq \llbracket reg(14) \rrbracket_{B_\alpha} & \text{wobei } iptoaddr : IP_B \rightarrow \\ \llbracket reg \rrbracket_{B_\alpha} &= \llbracket reg \rrbracket_{\mathcal{A}} \\ \llbracket freg \rrbracket_{B_\alpha} &= \llbracket freg \rrbracket_{\mathcal{A}} \\ \llbracket inp_\alpha \rrbracket_{B_\alpha} &= \llbracket inp_\alpha \rrbracket_{\mathcal{A}} \\ \llbracket out_\alpha \rrbracket_{B_\alpha} &= \llbracket out_\alpha \rrbracket_{\mathcal{A}} \\ \llbracket bz \rrbracket_{\mathcal{A}} &= iptoaddr(ip) \\ \mathfrak{B}_\alpha &\models D(getbef(prog_\alpha, x)) \Rightarrow iptoaddr(x) < reg(14) \end{aligned}$$

$$\llbracket Read \rrbracket_{\mathcal{A}}(\llbracket iptoaddr(x) \rrbracket_{B_\alpha}) = \llbracket getbef(prog_\alpha, x) \rrbracket_{B_T}$$

QUAD statische Funktion mit $iptoaddr(\langle l, i \rangle) \doteq labtoaddr(l) + intoquad(i * 4)$ ist.

Satz 5.3 (Korrektheit der Transformation für Sprünge)

ρ definiert eine 1- n -Simulation.

Korollar 5.4 (Korrektheit der Übersetzung)

Die Übersetzung von C-- zu DEC-Alpha-Assembler Code ist korrekt.

Beobachtung

Die letzten beiden Eigenschaften der Relation ρ können statisch überprüft werden.

Programmprüfung

Beweisverpflichtungen

- Die Grundblöcke dürfen sich im Speicher nicht überlappen
- Die anderen Befehle müssen im Speicher in derselben Reihenfolge wie im Grundblock vorkommen
- Die Sprünge müssen korrekt abgebildet sein

$$\mathfrak{B}_\alpha \models \text{LabAddr}(l) \leq \text{LabAddr}(l') + \text{BBsize}(l') + \text{Jmp}(l) \wedge \text{LabAddr}(l) \geq \text{LabAddr}(l') \Rightarrow l \doteq l'$$

$$\llbracket \text{getbef} \rrbracket_{\mathfrak{B}_\alpha}(\llbracket \text{prog} \rrbracket_{\mathfrak{B}_\alpha}, x) = \llbracket \text{Read} \rrbracket_\alpha(\llbracket \text{iptoaddr}(x) \rrbracket_{\mathfrak{B}_\alpha})$$

Sprünge als Übung

$$\text{LabAddr}(l) \triangleq \text{labeltoaddr}(\text{prog}, l)$$

$$\text{BBSize}(l) \triangleq \text{bbsize}(\text{prog}, l)$$

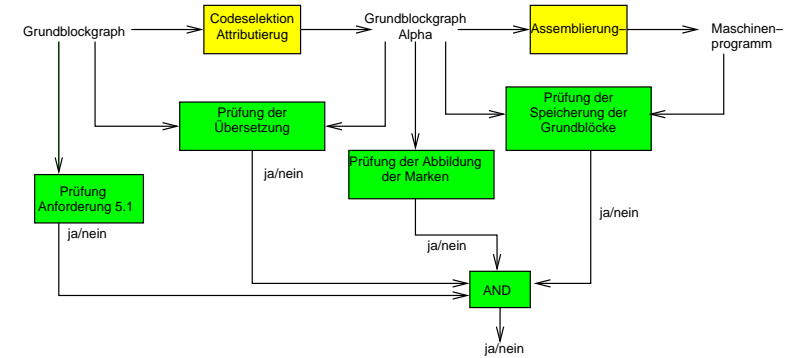
$$\text{Jmp}(l) \triangleq \text{jump}(\text{prog}, l)$$

Beobachtung

Diese Eigenschaften können statisch überprüft werden.

⇒ Programmprüfung stellt Korrektheit sicher

Übersetzungsvalidierung



- Die grünen Funktionen müssen verifiziert werden.
- ⇒ Zusammen mit Satz 5.4 ergibt sich ein verifizierendes Backend
- ⇒ Zusammen mit dem Resultat aus Kapitel 4 sich ein verifizierender Übersetzer