Diplomarbeit

Plattformabhängige Eliminierung gemeinsamer Teilausdrücke auf Quellcode-Ebene

Michael Vogt

5. November 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

1.1.1 Eingebettete Systeme

Wenn man heute von der großen Verbreitung von Computern spricht, so stellt man sich häufig graue PCs unter jedem Schreibtisch vor. Jeder Büroarbeitsplatz, jeder Student und eine große Zahl von Privathaushalten ist heute mit einem Computer ausgestattet. In Wirklichkeit ist die Verbreitung von Computern aber noch viel größer. Sie wird kaum wahrgenommen, da es sich nicht um Rechner im traditionellen Sinn handelt, sondern um *eingebettete Systeme*. Solche Computer verrichten ihren Dienst weitgehend unbemerkt, da sie meist in größere Produkte integriert sind. Bei jedem Telefonat mit einem modernen Telefon oder Handy wird ein eingebettetes System benutzt. Im Jahr 2002 verfügten 64% der privaten Haushalte über ein Handy, aber nur 55% über einen PC [Bun03]. Ähnliche Entwicklungen lassen sich in vielen anderen Bereichen des täglichen Lebens beobachten. Ein modernes Oberklassenfahrzeug wie die Mercedes S-Klasse verfügt bereits heute über mehr als 50 Steuergeräte, also kleine Computer, die zum Beispiel elektronische Bremsunterstützung, verfeinerte Motorsteuerung oder Fahrzeugstabilisierung (ESP) ermöglichen. Dieser Trend wird sich fortsetzen und dafür sorgen, daß auch in den unteren Marktsegmenten die Zahl an Mikroprozessoren im Fahrzeug steigt. Mehr „Intelligenz“ im Fahrzeug verspricht mehr Sicherheit und Effizienz [Gre03].

Man kann also zusammenfassend zwei Arten von Computersystemen unterscheiden. Zum einen die traditionellen Rechner, die für keine bestimmte Aufgabe festgelegt sind (general purpose). Darunter fallen fast alle Arten von „normalen“ Rechnern, vom PC bis zum Supercomputer. Zum anderen gibt es die Computersysteme, die nur für einen bestimmten Zweck entwickelt und programmiert werden. Diese eingebetteten Systeme (embedded systems, special purpose computer) verrichten ihren Dienst weitgehend unbemerkt und sind für den Anwender nicht direkt als Computersystem erkennbar.

Eingebette Systeme werden in fast allen Bereichen der Wirtschaft immer wichtiger. In dem Maße, in dem die Elektronik „intelligenter“ wird, wächst auch der Bedarf an eingebetteten Systemen. Bereits im Jahr 2000 wurden 75% aller 32 Bit Prozessoren in eingebetteten System verwendet. Man spricht auch schon vom Zeitalter „des verschwindenden Computers“ [Mar03].

Neben den eingangs erwähnten Beispielen finden sich eingebettete Systeme noch in zahlreichen weiteren Bereichen des modernen Lebens:

- Unterhaltungselektronik: Digitales Fernsehen, digitale Videorecorder, DVD und MP3 Abspielgeräte, Digitalkameras

- Kommunikationstechnik: Handys, UMTS, Internetrouter
- Fahrzeugtechnik: Bremsunterstützung, Unterhaltung, Navigation, Einparkhilfe, Motoroptimierung
- Hausbau: Beleuchtungsregelung, Temperatursteuerung
- Robotik: Fabrikautomatisierung, Haushaltsunterstützung, medizinische Anwendungen
- Medizinsysteme: Kernspintomographie, Intensivmedizin, minimal invasive Eingriffe

An ein eingebettetes System werden andere Anforderungen gestellt als an einen PC. Zahlreiche Probleme, die bei einem PC hingenommen werden, sind für ein eingebettetes System nicht tolerierbar. Dazu gehören schwer zu erlernende Benutzerschnittstellen, die Notwendigkeit, Software und Treiber zu installieren, lange Startzeiten, Systemabstürze und die Gefahr durch Viren und Würmer. Ein eingebettetes System muß intuitiv, zuverlässig und sicher sein. Zudem spielt Effizienz von Energie, Speicherplatz und Geschwindigkeit eine viel größere Rolle als bei traditionellen Rechnern, da hohe Effizienz niedrige Herstellungskosten bedeutet.

Traditionell wurden eingebettete Systeme in Assembler programmiert, um den Effizienzanforderungen gerecht zu werden. Mit der steigenden Komplexität der Anwendungen wurde dies aber immer weniger praktikabel. Auch der Wunsch, Programme und Programmbibliotheken für verschiedene Prozessoren zu nutzen, ist ein weiterer Grund, von der direkten Assemblerprogrammierung Abstand zu nehmen und auf eine Hochsprache zu wechseln. Es ist heute üblich, daß viele eingebettete Systeme in C programmiert werden. C ist maschinennah genug, um die nötige Kontrolle über den Rechner zu behalten und abstrahiert gleichzeitig von der konkreten Architektur, indem der Prozessor als Blackbox mit linearem Speicher angesehen wird[1]. Die verstärkte Nutzung von C führt dazu, daß die Anforderungen an die Compiler steigen. Nur wenn diese den Effizienzanforderungen gerecht werden, ist die Entwicklung in einer Hochsprache wie C praktikabel.

1.1.2 Compiler

Compiler sind Softwareanwendungen, die aus einer Hochsprache semantisch äquivalenten plattformabhängigen Maschinencode erzeugen. Besonders für eingebettete Systeme spielen effiziente Compiler eine wichtige Rolle, da jede Verbesserung in der Laufzeit oder der Größe des Programmcodes sich positiv auf den Preis des Systems auswirkt.

Phasen

Ein moderner Compiler arbeitet typischerweise in mehreren Phasen. Zunächst wird die Eingabe analysiert und in eine Zwischendarstellung umgewandelt (frontend), anschlie-

[1] Dies hat zu dem Ausspruch geführt: „C is just a fancy assembler language“.

ßend wird daraus der Maschinencode erzeugt (backend) [Muc97]. In der Analysephase werden folgende Schritte durchgeführt:

1. Lexikalische Analyse: Die Eingabe wird zerlegt (Scanner, Lexer).
2. Syntaktische Analyse: Die syntaktische Korrektheit des Programms wird überprüft. Der Syntaxbaum wird aufgebaut (Parser).
3. Semantische Analyse: Es wird geprüft, ob bestimmte Rahmenbedingungen eingehalten werden, z. B. ob Zuweisungen verträglich sind.

Im Backend (Synthesephase) wird erst optimiert und anschließend der Maschinencode erzeugt. Die Abbildung 1.1 stellt die Phasen dar.

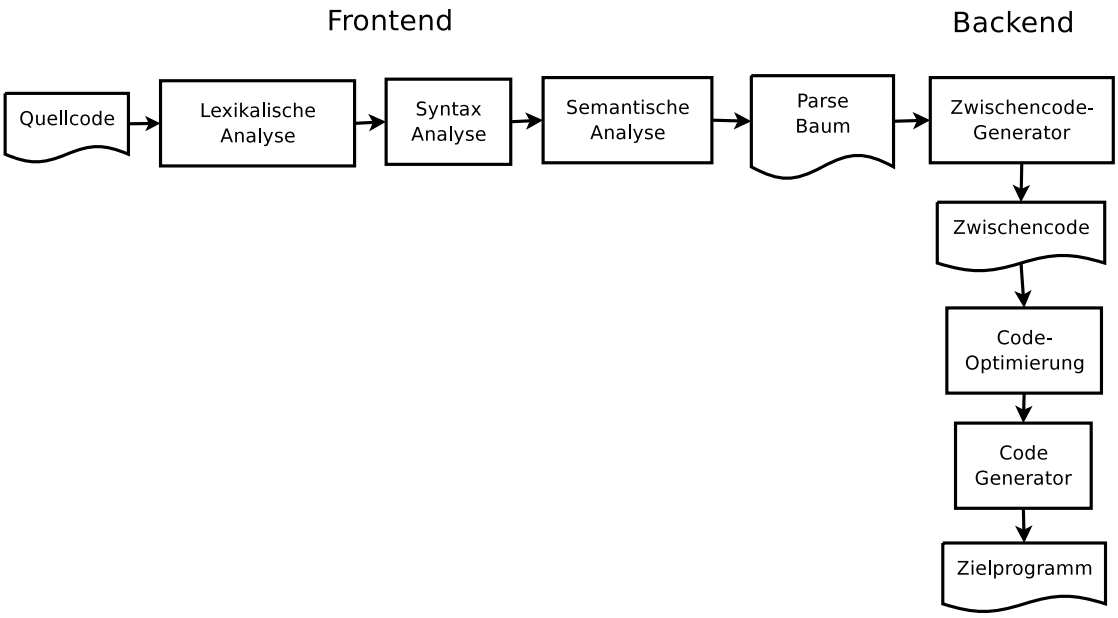

Abbildung 1.1: Struktur eines optimierenden Compilers

Optimierung

Moderne Compiler wenden verschiedene Techniken zur Programmoptimierung an. Dazu arbeiten sie auf der Zwischendarstellung des Programms. Üblicherweise werden zuerst prozessorunabhängige Optimierungen durchgeführt. Hierzu ist eine „medium level“ Zwischendarstellung gängig. Danach wird diese Darstellung in eine „low level“ Darstellung umgewandelt. Auf dieser werden dann architekturspezifische Optimierungen vorgenommen. Beispiele für prozessorunabhängige Optimierungstechniken eines modernen Compilers sind:

- Common subexpression elimination: Hier werden mehrfach vorkommende Ausdrücke eliminiert. Diese Technik wird ausführlich im Kapitel 1.2 beschrieben.

```
for(i=0; i<n; i++)                          for(i=0; i<n; i++)
  for(j=0; j<n; j++) {                        for(j=0; j<n; j++) {
                                                c = C[i][j];
    for(k=0; k<n; k++)                  ⇒       for(k=0; k<n; k++)
      C[i][j] += A[i][k]*B[k][j];                 c += A[i][k]*B[k][j];
                                                C[i][j] = c;
  }                                           }
```

Abbildung 1.2: Beispiel für Scalar Replacement

- Dead code elimination: Programmcode, der nie ausgeführt werden kann, wird entfernt. Ein Beispiel wäre ein `if()` Ausdruck, dessen Bedingung nie erfüllt werden kann.

  ```
  if(false) {
    ...
  };
  ```

  Dieser Code kann entfernt werden.

- Loop-invariant code motion: Wenn eine Berechnung in einem Schleifenkörper erfolgt, die Parameter der Berechnung aber nicht von der Schleife abhängen, so kann die Berechnung auch schon außerhalb der Schleife erfolgen.

  ```
  for(i=0; i<MAX; i++) {
    ...
    a=b+c;
  };
  ```

  Im Beispiel kann die Berechnung von `a` vor dem Schleifenanfang durchgeführt werden, solange `b` und `c` nicht von `i` abhängen.

- Constant folding: Ausdrücke, die schon beim Kompilieren ausgewertet werden können, werden berechnet und zusammengefaßt.

  ```
  screen_buffer = 1024*768*3;
  ```

  Hier sind alle Konstanten zum Zeitpunkt des Kompilierens bekannt. Der Compiler kann also die Berechnung durch `screen_buffer=2359296` ersetzen.

- Scalar replacement: Hier wird versucht, häufig benutzte Array Elemente in Variablen zwischenzuspeichern. Diese Variablen können dann in Registern zwischengespeichert werden und so können Speicherzugriffe vermieden werden. Abbildung 1.2 zeigt ein Beispiel für diese Optimierung.

Dieser verbesserte und prozessorunabhängige Code wird nun weiter optimiert. Dabei werden prozessorabhängige Charakteristika einbezogen:

- Code selection: Die Auswahl, welcher Maschinencode für eine bestimmte Operation benutzt wird.
- Register allocation: Die Auswahl, welche Werte zum jeweiligen Zeitpunkt des Programmablaufs in Registern gespeichert werden. Eine optimale Lösung dieses Problems ist NP-hart. Die Zahl der Register ist beschränkt und jedes Datum, das nicht in einem Register gespeichert werden kann, muß im Hauptspeicher abgelegt werden, der im schlimmsten Fall um Größenordnungen langsamer ist als ein Register. Ziel der Registerallokation ist es, möglichst wenig Daten zwischen Hauptspeicher und Registern zu transferieren. Solche Transfers werden durch Spill-Code ausgeführt, der an Stellen eingefügt wird, wo die Zahl der Register nicht ausreicht. Die Registerallokation erfolgt meist im Low Level Zwischencode oder im Maschinencode, als Algorithmus hat sich das Graph Coloring durchgesetzt. Der Algorithmus verläuft in vier Phasen:
  1. Zunächst wird jedem Objekt ein virtuelles Register `s1`, `s2`, ... zugewiesen.
  2. Dann wird überprüft, welche Objekte wirklich in Registern allokiert werden müssen.
  3. Ein Unverträglichkeits-Graph (interference graph) wird erzeugt, dessen Knoten aus allokierbaren Objekten und echten Registern besteht. Eine Kante zwischen zwei Objekten stellt dabei dar, daß beide zur gleichen Zeit lebendig sind und damit nicht das gleiche Register belegen können, eine Kante zwischen Objekt und Register, daß dieses Objekt nicht in dieses Register allokiert werden kann (z. B. eine Ganzzahl in ein Fließkommaregister).
  4. Der Graph wird mit R Farben eingefärbt, wobei R die Zahl der verfügbaren Register darstellt. Zwischen jedem verbundenen Knoten muß eine andere Farbe verwendet werden. Ist das nicht möglich („Registerdruck zu hoch“), muß ein Objekt in den Hauptspeicher ausgelagert werden (spilling). Dadurch wird der Graph vereinfacht, und das Einfärben kann erneut versucht werden.
  5. Jedes Objekt wird in das Register mit der gleichen Farbe allokiert.
- Instruction scheduling: Um parallel arbeitende Einheiten eines Prozessors auszunutzen (z. B. zwei ALUs), wird der Programmcode so umgeordnet, daß möglichst immer alle diese Einheiten genutzt werden.

Dies stellt nur eine kleine Auswahl aller verwendeten Methoden in einem modernen Compiler dar. Für den GNU C Compiler findet sich zum Beispiel eine komplette Übersicht aller verwendeten Techniken auf der GCC Webseite[2].

1.2 Konventionelle CSE Optimierung

Die Common Subexpression (CSE) Eliminierung ist ein gängiges Verfahren in der Compileroptimierung. Bei ihm werden mehrfach im Quellcode vorkommende Berechnungen

[2] `http://www.redhat.com/software/gnupro/technical/gnupro_gcc.html`

optimiert, indem diese nur einmal durchgeführt werden und an den anderen Stellen jeweils der Wert derselben eingesetzt wird. Dabei dürfen sich die Operanden des Ausdrucks nicht ändern. Die grundlegenden Algorithmen zur CSE Optimierung werden in [Coc70] und [Muc97] vorgestellt.

Ein Beispiel für eine CSE Eliminierung könnte so aussehen:

```
gauss_x_compute += (int)in_pixels[(x - 1) % 3];
computex = (y-2)%3 + (x - 1) % 3;
```

Es ist zu erkennen, daß der Ausdruck `(x-1)%3` zweimal berechnet wird. Da sich aber `x` nicht ändert, ist die zweite Berechnung redundant und kann eliminiert werden. Durch die Eliminierung des zweiten Ausdrucks wird der Code wie folgt verändert:

```
cse_023_45 = (x - 1) % 3;
gauss_x_compute += (int)in_pixels[cse_023_45];
computex = (y-2)%3 + cse_023_45;
```

Die Berechnung erfolgt nur noch einmal, der Wert wird in einer neuen lokalen Variable gespeichert, und anschließend wird der berechnete Wert benutzt.

1.2.1 Lokale CSE Optimierung

Bei der lokalen CSE Optimierung werden nur Basisblöcke betrachtet. Basisblöcke sind in der Compilertechnik Blöcke, die in jedem Fall sequentiell durchlaufen werden; Sprungbefehle sind nur am Ende enthalten. Basisblöcke stellen Knoten im Kontrollflußgraphen dar.

Definition Basisblock: Ein Basisblock $B = (I_1, \ldots, I_n)$ ist eine Folge von Befehlen mit maximaler Länge, so daß:

1. B nur durch den ersten Befehl I_1 betreten werden kann
2. B nur durch den letzten Befehl I_n verlassen werden kann.

Die lokale CSE Optimierung kann leicht während der Erzeugung des Zwischencodes vorgenommen werden. Das Finden identischer Ausdrücke und das Überprüfen, ob sich die Operanden des Ausdrucks verändern, ist hier leicht möglich.

Durch die Beschränkung auf Basisblöcke können keine CSEs gefunden werden, die nicht im gleichen Basisblock liegen. Im folgenden Beispiel in Abbildung 1.3 wird `y%3` als CSE erkannt und eliminiert, `(x-1)%3` bleibt dagegen unentdeckt.

1.2.2 Globale CSE Optimierung

Die Beschränkung auf Basisblöcke wird bei der globalen CSE Optimierung (global common subexpression elimination) aufgehoben. Diese Analyse ist wesentlich aufwendiger als die lokale CSE Optimierung, da auch der Kontrollfluß berücksichtigt werden muß, um zu überprüfen, ob sich die Operanden ändern können.

Im folgenden Beispiel in Abbildung 1.4 entdeckt die globale CSE Optimierung beide CSEs, obwohl eine davon in zwei verschiedenen `if()` Ausdrücken liegt.

```
if(x>=2)
  gxc += pix[(x-1)%3];
if(y>=1) {
  gxc += pix[y%3-(x-1)%3];
  gxc += a[y%3];
  ...
}
```

$\Longrightarrow$

```
if(x>=2)
  gxc += pix[(x-1)%3];
if(y>=1) {
  int cse0 = y%3;
  gxc += pix[cse0-(x-1)%3];
  gxc += a[cse0];
  ...
}
```

Abbildung 1.3: Lokale CSE Eliminierung

```
if(x>=2)
  gxc += pix[(x-1)%3];
if(y>=1) {
  gxc += pix[y%3-(x-1)%3];
  gxc += a[y%3];
  ...
}
```

$\Longrightarrow$

```
int cse1=(x-1)%3;
if(x>=2)
  gxc += pix[cse1];
if(y>=1) {
  int cse0 = y%3;
  gxc += pix[cse0-cse1];
  gxc += a[cse0];
  ...
}
```

Abbildung 1.4: Globale CSE Eliminierung

1.3 Quellcodeoptimierung

Neben diesen traditionellen Techniken der Optimierung von Programmcode hat sich in den letzten Jahren ein neuer Trend herausgebildet, um noch bessere Optimierungsergebnisse zu erzielen. Bei der Quellcodeoptimierung wird ein bestehender Quellcode analysiert und unter Beibehaltung der Semantik in einen effizienteren Quellcode transformiert. Dieser Quellcode wird anschließend von einem normalen Compiler übersetzt.

Warum betreibt man überhaupt Quellcodeoptimierung? Die Gründe hierfür sind vielfältig. Zum einen ist die Quellcodeoptimierung inhärent portabel, da sowohl Quellprogramm als auch Zielprogramm in der verwendeten Programmiersprache erzeugt werden. Außerdem läßt sich die Korrektheit der Optimierungen leichter überprüfen. Der erzeugte Quellcode ist leichter lesbar als Maschinencode, kann auf jedem Rechner mit passendem Compiler ausgeführt, getestet und mit symbolischen Debuggern untersucht werden. Dagegen muß bei erzeugtem Maschinencode zunächst ein passender Chip oder Simulator vorhanden sein. Debugger sind u.U. noch nicht oder nur rudimentär verfügbar usw. Ein weiterer Grund ist, daß bestimmte Optimierungen leichter auf dem Quellcode durchzuführen sind. Insbesondere solche Optimierungen, die auf einem hohen Abstraktionsniveau arbeiten, haben hier klare Vorteile, da mehr Informationen verfügbar sind als in einer Zwischensprache von niedrigem Abstraktionsniveau. Hinzu kommt, daß bestimmte

Techniken „von Hand“ getestet werden können, bevor eine passende Implementierung im Compiler vorliegt.

Eine Implementierung der globalen CSE Optimierung auf Quellcodeebene wird in [Fal04] vorgestellt (High-Level CSE Optimierung). Die CSE Optimierung arbeitet hier auf der Ebene kompletter Prozeduren, und es wird jede gefundene CSE eliminiert. Die Analyse erfolgt auf der Ebene des C Quellcodes, und es wird auch wieder ein C Quellcode generiert. So gehen keine Informationen über Programmkonstrukte verloren.

Andere Techniken der Sourcecodeoptimierung, wie DTSE (Data Transfer and Storage Exploration) [CDK+02], können zu einer großen Zahl von CSEs im Sourcecode führen, wenn z. B. Adressberechnungen vereinheitlicht werden.

Der Aufwand der globalen CSE führt dazu, daß viele aktuelle Compiler sie nicht vollständig implementieren. Die Ergebnisse der Untersuchungen zeigen, daß eine High-Level CSE auf Quellcodeebene deutliche Laufzeitverbesserungen ermöglicht.

1.4 Plattformabhängige High-Level CSE Optimierung

Eine CSE Optimierung ist immer mit gewissen Kosten verbunden, die je nach verwendetem Prozessor und Compiler variieren. Der mehrfach berechnete Wert muß zwischengespeichert und geladen werden. In den meisten Fällen ist dieses Laden und Speichern zwar kostengünstiger als die Berechnung, dies muß aber für sehr einfache Ausdrücke nicht der Fall sein. Da die Implementierung in [Fal04] diesen Aspekt nicht berücksichtigt und statt dessen jede gefundene CSE eliminiert, stellt sich die Frage, ob sich unter Berücksichtigung dieser plattformabhängigen Aspekte noch bessere Ergebnisse der CSE Optimierung erzielen lassen. Eine solche plattformabhängige High-Level CSE Optimierung soll im Rahmen dieser Arbeit entwickelt werden.

In diesem Abschnitt soll das erhöhte Optimierungspotential bei einer plattformabhängigen Auswahl zu eliminierender Ausdrücke verdeutlicht werden. Hierzu wurde mit einem einfachen Greedy Algorithmus versucht, die optimalen CSEs in dem CAVITY Benchmark zu finden. Der CAVITY Benchmark stammt aus dem Bereich der digitalen Bildverarbeitung und wird in [BTC89] näher beschrieben. Er verfügt über zahlreiche mehrfach vorkommende Ausdrücke, die wegen der verschachtelten Strukur des Programms von normalen Compilern nur schwer gefunden werden.

Der Algorithmus zur plattformabhängigen Ermittlung des Optimierungspotentials („Feedback Algorithmus“) bekommt als Eingabe einen C-Quellcode. Zunächst werden darin alle mehrfach vorkommenden Ausdrücke identifiziert, und für jeden dieser Ausdrücke wird ein Zielquellcode erstellt, in dem dann dieser Ausdruck eliminiert worden ist. Jeder dieser Zielquellcodes wird einzeln mit dem gewählten Compiler übersetzt und auf der realen Hardware ablaufen lassen (Feedback Schritt). Dann wird die CSE ausgewählt, die die höchsten Laufzeitverbesserungen erzielt hat (die Auswahl ist „greedy“) und sie wird permanent eliminiert. Der Quellcode, der jetzt eine CSE weniger enthält, wird nach dem gleichen Muster ein weiteres Mal untersucht. Dies wird so lange fortgesetzt, bis keine Laufzeitverbesserungen mehr gefunden werden oder bis keine CSEs mehr im Quelltext vorhanden sind.

Formal arbeitet er wie folgt:

1. Initialisiere eine leere Liste `beste_cses`
2. Initialisiere eine Liste `alle_cses` mit allen gefundenen CSEs aus dem Quellcode
3. Initialisiere den Wert `beste_laufzeit` mit ∞
4. Für jede `cse` in `alle_cses`:
   a) Füge `cse` temporär zu `beste_cses` hinzu
   b) Eliminiere alle CSEs in `beste_cses` und erzeuge daraus einen neuen Quellcode
   c) Messe die Laufzeit des Programms, das aus diesem Quellcode erzeugt wurde
   d) Wenn die Laufzeit besser ist als `beste_laufzeit`, markiere `cse` als beste CSE der Liste und mache die gemessene Laufzeit zu `beste_laufzeit`
5. Prüfe, ob in diesem Lauf eine Laufzeitverbesserung gemessen wurde (wurde eine CSE in `alle_cses` markiert?). Wenn nicht, gehe zu 7)
6. Füge die als beste markierte CSE permanent zur Liste `beste_cses` hinzu, entferne diese CSE aus der Liste `alle_cses` und fahre mit 4) fort.
7. Gib die beste gefundene Laufzeit und die Liste `beste_cses` aus

Dabei wurden Laufzeitverbesserungen von 11,7%–20,7% gegenüber der High-Level CSE Optimierung gefunden. Abbildung 1.5 stellt die Ergebnisse dar. In dem Diagramm finden sich auf der X-Achse die untersuchten Plattformen und auf der Y-Achse die gemessenen Laufzeiten in Prozent. Für jede Plattform wurden drei Versionen des Quellcodes gemessen: Original ist die unmodifizierte Version des CAVITY Benchmark, High-Level CSE ist die High-Level CSE Optimierung, die in [Fal04] beschrieben wird, und High-Level Feedback CSE meint die Laufzeiten, nachdem der oben beschriebene Algorithmus die besten plattformabhängigen CSEs identifiziert hat. Man erkennt im Diagramm, daß die High-Level CSE Optimierung deutliche Laufzeitgewinne von bis zu 50% erzielen kann, auch wenn keine plattformabhängigen Informationen berücksichtigt werden. Diese bereits guten Ergebnisse werden von der High-Level Feedback Optimierung noch einmal deutlich um 11,7%–20,7% verbessert. Damit ergeben sich Verbesserungen durch die plattformabhängige High-Level CSE von ingesammt 26,6% (Intel/icc) bis 63,3% (Thumb/gcc) gegenüber dem unmodifizierten Original Quellcode.

Dabei sind die Verbesserungen auf jeder untersuchten Plattform und die jeweils gefunden CSEs sehr unterschiedlich. Für den Intel Pentium III mit dem GNU GCC Compiler findet sich z. B. die CSE `(x-1)%3` als eine der besten CSEs. Diese CSE wird auf dem gleichen Prozessor mit dem Intel ICC Compiler nicht als gute CSE identifiziert. Es ist im Gegenteil sogar so, daß diese CSE mit dem Intel ICC Compiler zu 7% Laufzeitverschlechterung führt. Die Ergebnisse zeigen, daß eine plattformabhängige High-Level

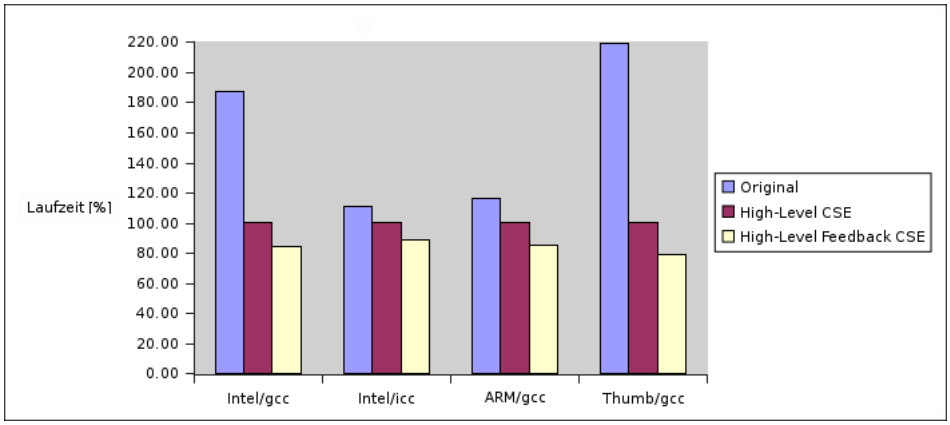

Abbildung 1.5: Zusammenfassung der Feedback Ergebnisse

CSE Optimierung einen interessanten Forschungsgegenstand darstellt, mit dem deutliche Laufzeitverbesserugen möglich sind.

Die experimentelle Feedback-Methode basiert darauf, daß das Programm wiederholt in realer Hardware ausgeführt wird und die so gemessenen Laufzeiten für weitere Läufe berücksichtigt werden. Werden im Programm n CSEs gefunden, so müssen im schlechtesten Fall

$$n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2} \tag{1.1}$$

Testprogramme erzeugt und ausgeführt werden. Dieser Fall tritt dann ein, wenn jede der gefundenen CSEs zu Laufzeitverbesserungen beiträgt. Damit ist der Feedback-Algorithmus nicht für die Praxis geeignet, da dessen Aufwand zu groß ist und die Verfügbarkeit der Zielarchitektur voraussetzt. Es wird eine effiziente analytische Methode benötigt, die ohne das Ausführen des erzeugten Programmes auskommt.

Bei der Bewertung der Kriterien, wann eine CSE eliminiert werden sollte, ist die Literatur vage. Es bleibt bei dem allgemeinen Hinweis, daß eine CSE Eliminierung nicht immer sinnvoll ist. Diese Arbeit untersucht die Einflußfaktoren experimentell auf Quellcodeebene und versucht eine Qualifizierung und Quantifizierung der Parameter.

Ein Problem der klassischen CSE Optimerung im Compiler ist, daß sie bereits zu einem relativ frühen Zeitpunkt der Optimierung stattfindet. Damit liegen viele Informationen noch nicht vor, die für eine Entscheidung über das Eliminieren wichtig sind. Gerade Informationen über den Registerdruck fehlen in der Regel, da die Registerallokation erst in einer späten Phase auf der Low-Level IR erfolgt.

1.5 Ziele der Arbeit

Durch das Feedback Experiment zeigt sich, daß Laufzeitverbesserungen für Anwendungen mit vielen CSEs möglich sind. Die vorliegende Arbeit behandelt die Frage, welche

Parameter bei einer gefundenen CSE Einfluß auf die Laufzeit haben, und ob es möglich ist, durch Analyse dieser Parameter eine geeignete Auswahl der CSEs zu treffen, um die Laufzeitgewinne, die per Feedback Optimierung erreicht wurden, durch Analyse im Vorfeld zu gewinnen.

Die vorhandene Implementierung einer High-Level CSE auf Quellcode-Ebene muß also so erweitert werden, daß plattformabhängige Informationen berücksichtigt werden. Das ermöglicht eine Entscheidung darüber, ob eine gefundene CSE eliminiert wird oder nicht. Eine Eliminierung ist zum Beispiel nicht sinnvoll, wenn der berechnete Ausdruck sehr einfach ist. Sind die Zwischenergebnisse, die zur Berechnung nötig sind, bereits in Registern vorhanden, so ist eine Eliminierung unter Umständen ebenfalls nicht sinnvoll.

Ein weiterer Einflußfakor auf die Laufzeit stellt die Zahl der freien Register dar. Wenn der berechnete Wert in den Speicher übertragen werden muß, weil keine Register mehr frei sind, so schadet das der Geschwindigkeit, da ein Speicherzugriff langsam ist. Dies gilt insbesondere für moderne Architekturen, wo die langsame Speichertaktung einen zunehmenden Flaschenhals darstellt. Betrachten wir den folgenden Fall:

```
b = a + 2;
c = a + 2;
```

Die CSE Optimierung wandelt ihn um zu:

```
cse_0 = a + 2;
b = cse_0;
c = cse_0;
```

Es wird eine zusätzliche Variable erzeugt, die ein Register oder eine Speicherstelle benötigt. Bei modernen Architekturen erfolgt eine Ganzzahladdition in einem Takt, so daß die Optimierung keinen Gewinn bringt, wenn `a` in einem Register gehalten wird. Es erfolgt sogar eine Verschlechterung, falls `cse_0` im Hauptspeicher gehalten werden muß. Dies ist ein besonderes Problem für Architekturen, die eine geringe Zahl von allgemeinen Registern haben.

Erst mit Hilfe einer prozessor- und compilerabhängigen Kostenfunktion können Ausdrücke bezüglich ihrer Komplexität bewertet werden. Sie könnte wie folgt aussehen:

Operation	Kosten
Addition (Integer)	10
Multiplikation (Integer)	11
Division (Integer)	30
...	

In dieser Diplomarbeit werden derartige Kostenfunktionen für verschiedene Kombinationen von Prozessoren und Compilern bestimmt. Neben den Kosten für die Operationen spielt auch der Abstand zwischen zwei Nutzungen einer neu eingefügten CSE Variable eine Rolle. Bei einem zu großen Abstand kann es sinnvoll sein, keine CSE Eliminierung durchzuführen, da sonst ein Register für eine sehr lange Zeit gebunden ist oder ein Hauptspeicherzugriff nötig wird.

Der dramatische Kostenunterschied zwischen einer Register– und einer Hauptspeicherzuweisung legt den Einsatz einer Lebenszeitanalyse nahe, um die Zahl der freien Register abzuschätzen. Sind nur wenige Variablen lebendig, kann z. B. aggressiver eliminiert werden, als wenn sich eine große Zahl von aktiven Variablen um die wenigen freien Register streitet. Ebenso interessant ist die Anzahl der identischen Vorkommen einer CSE. Je größer sie ist, desto mehr Berechnungen werden eingespart, und desto größer ist der Gewinn der Optimierung.

Alle diese Parameter zu gewichten, um daraus die richtigen Schlüsse zu ziehen, stellt eine weitere Herausforderung dieser Arbeit dar. Die aus den Untersuchungen gewonnen Erkenntnisse sollen mit Benchmarks auf verschiedenen Prozessoren und Compilern überprüft werden.

1.6 Verwandte Arbeiten

Im Zusammenhang mit der klassischen CSE Optimierung gibt es eine Reihe von Arbeiten wie [Muc97] und [ASU96]. Auf mögliche Einschränkungen bei der Auswahl, welche CSEs zu eliminieren sind, weisen [HP03] und [Muc97] hin. Häufig wird die Optimierung in lokale und globale CSE Optimierung aufgeteilt. Auf den Zusammenhang zwischen der Eliminierung einer CSE und dem Registerdruck geht [BGS94] ein.

Im Bereich der prozessorspezifischen Quellcode-Optimierung wird im wesentlichen an der Ausnutzung von Compiler Intrinsics geforscht. Intrinsics sind spezielle Funktionen, die direkt auf besondere Features des Prozessors abgebildet werden. Es werden keine normalen Funktionsaufrufe generiert, sondern Sequenzen von Assemblercode, um eine bestimmte Aufgabe, wie z. B. saturierende Addition, zu erledigen. Untersuchungen von [AEG+02] und [CWK+99] zeigen, daß 30% bis 50% Laufzeitverbesserungen durch den Einsatz von Intrinsics möglich sind.

Auch [PBSB01] verwenden in ihrem SWARP Quellcode-Transformations Framework Intrinsics. Dabei wird der Philips TriMedia Prozessor untersucht. Zunächst werden Schleifentransformationen durchgeführt, dann werden mittels Mustererkennung verschiedene Quellcodekonstrukte erkannt und durch Compiler Intrinsics ersetzt.

Ein Nachteil von Intrinsics ist, daß der erzeugte Quellcode nicht mehr zu allen Compilern und Prozessoren kompatibel ist. Außerdem ist das Erkennen und Einsetzen von Intrinsics noch nicht automatisiert. Die Lösung, die im Rahmen dieser Arbeit entwickelt wurde, ist dagegen automatisiert, adaptiv und portabel.

Ein anderer Ansatz der Quellcodeoptimierung stellt das CTT (C transformation tool) Framework dar [BKC99]. Der Benutzer kann anhand von Mustern und Gültigkeitsregeln Transformationen angeben, die dann auf den Quellcode angewendet werden. Damit können z. B. Schleifenoptimierungen durchgeführt werden. Die Arbeit von [Jak02] zeigt aber, daß Mustererkennung nicht ausreicht, da allgemeine Quellcodes beliebige Verschachtelungstiefen haben können, die durch Mustererkennung nicht adäquat abgebildet werden können.

Eine High-Level CSE Optimierung auf Quellcodeebene wird in [Fal04] vorgestellt. Eine

darauf aufbauende plattformabhängige Analyse ist neu und wird in der Literatur bisher nicht beschrieben.

1.7 Übersicht über die Arbeit

In Kapitel 2 werden die Grundlagen erläutert, die für das Verständnis der Arbeit nötig sind. Kapitel 3 beschäftigt sich mit der konkreten Realisierung der plattformabhängigen High-Level Optimierung. Deren Ergebnisse werden in Kapitel 4 vorgestellt. Kapitel 5 stellt eine Zusammenfassung der wichtigsten Erkenntnisse dar und gibt einen Ausblick auf weitere Arbeiten. Im Anhang A finden sich alle Meßergebnisse in tabellarischer Form.

2 Grundlagen und Konzepte

In diesem Kapitel werden die grundlegenden Techniken und Konzepte vorgestellt, die für das weitere Verständnis der Arbeit nötig sind. Es verschafft einen Überblick über die untersuchen Plattformen, die Zwischendarstellungen, die Methodik der Laufzeitmessungen, den Lua Interpreter und die Grundlagen Neuronaler Netze.

2.1 Untersuchte Plattformen

Für den weiteren Verlauf der Arbeit definieren wir eine Plattform als eine Kombination von Prozessor und Compiler. Um einen breiten Bereich abzudecken, wurden zwei verschiedene Compiler und zwei verschiedene Prozessoren ausgewählt. Auf ihnen wurden die Laufzeitverbesserungen durch die plattformabhängige High-Level CSE Analyse untersucht.

Mit dem Intel Pentium III wird ein Prozessor benutzt, der auf hohe Geschwindigkeit optimiert ist. In Kombination mit großen Caches und einem relativ langsamen Speichertakt im Vergleich zum internen Prozessortakt ist er ein guter Vertreter für einen modernen High Performance Prozessor. Damit wird das obere Ende des Leistungsspektrums abgedeckt. Der ARM7TDMI dagegen ist ein typischer Embedded Prozessor. Energieverbrauch und Codegröße wurden beim Design stärker berücksichtigt, die Ausführungsgeschwindigkeit spielt nicht die größte Rolle. Er ist damit als Beispiel für einen Prozessor im Bereich eingebette Systeme gut geeignet.

Der GNU GCC ist ein freier Compiler, der auf einer großen Zahl von Plattformen verfügbar ist. Er hat allerdings den Ruf, daß er bei den implementierten Optimierungen den besten kommerziellen Compilern hinterherhinkt. Daher ist es sinnvoll, zusätzlich einen sehr guten kommerziellen Compiler zu verwenden und zu untersuchen, ob sich auch hier Laufzeitverbesserungen zeigen. Der Intel ICC ist der Compiler, mit dem Intel seine SPEC Messungen für ihre Prozessoren durchführt. Er gilt als der beste verfügbare C Compiler für den Intel Pentium Prozessor.

Die ARM und Thumb Compiler aus dem ARM SDT (Software Developer Tools) wurden aus praktischen Gründen nicht eingesetzt. Die Version, die auf das verfügbare Evaluation Board angepaßt ist, liegt nur in einer Version von 1998 für das Betriebssystem Windows vor. Damit sind die Skriptmöglichkeiten, die für die automatischen Tests nötig sind, nicht verfügbar. Eine Version des ARM SDT, die für das Betriebssystem Solaris verfügbar ist, lag nur ohne die Memory Maps vor, die zum Betrieb des Boards nötig sind. Ohne diese Memory Maps kann der Linker keine Programme generieren, die auf dem Entwicklungsboard ausführbar sind. Kontrollmessungen mit der Solaris-Version des

Compilers sind sinnvoll, nachdem die Memory Maps angepasst worden sind, sind aber aus den angegebenen Gründen nicht Bestandteil dieser Arbeit.

2.1.1 Prozessoren

Intel Pentium III

Der Intel Pentium III Prozessor [Int02] wurde 1999 auf den Markt gebracht und wird mit einer internen Taktrate von 500-1000MHz und einer externen Taktung von 100MHz betrieben. Er besteht aus ca. 9,5 Millionen Transistoren. Er verfügt über 256KB L2 Cache mit halbem Prozessortakt und 16KB L1 Code und 16KB L1 Daten Cache mit vollem Prozessortakt. Er verfügt über acht allgemeine Register, die jeweils 32-bit breit sind, acht 80-bit Fließkommaregister, die über einen Stack adressiert werden, und acht 128-bit Spezialregister für Fließkommaaufgaben in der SSE (Streaming SIMD Extensions).

ARM7TDMI

Der ARM7TDMI [Adv01] ist ein 32-Bit RISC Prozessor, der speziell für eingebettete Systeme entwickelt wurde. Sein Design stammt aus dem Jahr 1993. Es wurde versucht, den besten Kompromiß zwischen Leistung, Größe und Energieverbrauch zu erzielen.

Es stehen zwei Betriebsarten zur Verfügung. Im ARM Modus werden 32-Bit Instruktionen verwendet, um maximale Geschwindigkeit zu erreichen. Es stehen hier 16 nutzbare allgemeine 32-Bit Register zur Verfügung.

Der Thumb Modus verwendet 16-Bit Instruktionen, um eine möglichst hohe Programmcodedichte zu erreichen (laut ARM sinkt die Programmgröße um 20%). Es stehen allerdings nur noch acht allgemeine Register zur Verfügung. Je nach Fertigungsverfahren kann er mit bis zu 133 MHz internem Takt betrieben werden. Er verfügt in der Standardkonfiguration über keinen Cache.

2.1.2 Compiler

GNU GCC

Der GNU GCC (Gnu Compiler Collection)[1] ist eine Sammlung von frei verfügbaren Compilern. Neben den Sprachen C und C++ werden Ada, Fortran, Objective C, Pascal, Java und Chill als Sprachen unterstützt. Maschinencode kann das Backend für eine Reihe von populären Plattformen erzeugen, u. a. Alpha, ARM, Intel IA-32 und IA-64 („Itanium“), Motorola 68000, PA-RISC, PowerPC, Sparc und SuperH. Der GNU GCC ist unter der GPL[2] als Freie Software verfügbar. Seine große Verfügbarkeit, die geringen Kosten und die Möglichkeit, ihn als Crosscompiler einzusetzen, machen den GCC zu einem guten Compiler für eingebette 32-Bit Systeme wie den ARM.

[1] `http://www.gnu.org/software/gcc`
[2] `http://www.gnu.org/licenses/gpl.html`

Intel ICC

Der Intel ICC[3] ist ein proprietärer Compiler der Firma Intel. Er wird für Linux und Windows angeboten und unterstützt C, C++ und Fortran. Programmcode kann für die Intel Pentium Serie (IA-32) und den Intel Itanium 2 erzeugt werden.

2.2 Zwischendarstellungen

Alle modernen Compiler speichern das Ergebnis der Analysephase in einer Zwischendarstellung. Man kann zwischen Zwischendarstellungen von hohem (high level), mittlerem (medium level) und niedrigem (low level) Abstraktionsgrad unterscheiden.

Diese Darstellung kann ein unterschiedliches Maß an Informationen enthalten. Schleifenkonstrukte können z. B. bei einer einfachen Zwischensprache in `goto` Anweisungen umgesetzt werden. In diesem Fall geht ein Teil der Information über die Schleife verloren, Schleifengrenzen können zum Beispiel ohne weiteres nicht mehr identifiziert werden.

Bei der Wahl der Zwischensprache für Quellcodeoptimierung muß eine Zwischendarstellung gewählt werden, bei der möglichst wenige Informationen verloren gehen. Insbesondere die Hochsprachenkonstrukte wie verschiedene Schleifenformen, Arrays und erweiterte Informationen zu Variablen müssen erhalten bleiben, um eine Rücktransformation in die Hochsprache zu ermöglichen.

2.2.1 GNU RTL

Die meisten Optimierungen werden auf der Zwischensprache RTL (Register Transfer Language) des GCC ausgeführt. Direkt nach dem Aufbau des Syntaxbaumes wird dieser in eine RTL Repräsentation umgewandelt. Dabei wird eine Beschreibung der Zielarchitektur verwendet, die u. a. die Zahl der verfügbaren Register enthält [Sta99] und der Assemblersprache schon sehr ähnlich ist.

Die GNU RTL ist eine Zwischensprache von niedriger Abstraktion, in der eine Registermaschine modelliert wird. Die Syntax erinnert an Lisp, jeder Eintrag in einer Liste repräsentiert einen Ausdruck. Es können folgende Konstrukte dargestellt werden:

- Arithmetische Operationen
- Vergleiche
- Sprung und Call Aufrufe
- Register und Speicheradressen

RTL Ausdrücke sind nach Lisp S-Ausdrücken modelliert und sehen z. B. so aus:

```
(set:SI (reg:SI 140) (plus:SI (reg:SI 138)(reg:SI 139)))
```

[3] `http://www.intel.com/software/products/compilers/clin/clinux.htm`

In diesem Ausdruck werden die SingleInteger (Machine Type: SI) Register 138 und 139 additiv verknüpft und im Register 140 gespeichert. Intern werden die Ausdrücke als Pointer auf Strukturen dargestellt.

Ein Nachteil der RTL Darstellung ist, daß sie bereits zum Zeitpunkt der Erzeugung maschinenabhängige Elemente enthält, da schon in diesem Schritt die Maschinenbeschreibung benutzt wird. Damit eignet sich die Zwischensprache des GCC nicht für plattformunabhängige Sourcecodetransformationen.

Bestrebungen des GCC Projekts, mit der neuen Tree SSA Architektur[4] diese frühe Abhängigkeit von der verwendeten Maschine aufzulösen, könnten in Zukunft die Situation verändern. Im GCC 3.5 werden zwei neue sprach- und maschinenunabhängige Bäume eingeführt, auf denen optimiert werden kann. Diese neuen Datenstrukturen heißen GENERIC und GIMPLE. Der Parser erzeugt dann GENERIC Bäume, die anschließend zu einer SSA (Static single assignment form)[5] basierten GIMPLE Darstellung umgewandelt werden. Auf der Ebene von GIMPLE wird dann der Großteil der neuen Optimierungen angewendet. Damit können globale Optimierungen auf Funktionsebene im GCC relativ leicht implementiert werden.

2.2.2 SUIF

Das Stanford University Intermediate Framework (SUIF) wird seit 1994 an der Universität von Stanford entwickelt und ist für Forschungszwecke frei verfügbar. Es stellt ein allgemeines Framework zur Verfügung, um optimierende Compiler zu entwickeln und an verschiedenen Optimierungen zu forschen. Das Frontend kann ANSI C und Fortran 77 verarbeiten. Die anschließenden Optimierungen werden als eigene Programme aufgerufen, die gegen den SUIF Kern gelinkt sind. Damit kann man schnell eigene Optimierungen realisieren.

SUIF baut aus dem C Quellcode einen abstrakten Syntaxbaum (High SUIF) auf. Dieser Syntaxbaum enthält alle Informationen, um daraus wieder einen C Quellcode zu generieren. Durch die hohe Abstraktionsebene sind Analysen von höherem Niveau möglich.

Neben dieser Zwischensprache existiert auch noch eine „Low SUIF“ Zwischensprache, die nur noch assemblerartige Ausdrücke enthält.

SUIF ist in zwei Versionen verfügbar, SUIF1 und SUIF2. Da die Optimierungen, die im Rahmen dieser Diplomarbeit entwickelt wurden, in das bestehende Framework von Dr. Falk integriert werden müssen, wurde die Implementierung in SUIF1 vorgenommen. Dies ist keine Einschränkung, da es keine fundamentalen Unterschiede zwischen SUIF1 und SUIF2 gibt, solange C als Sprache benutzt wird [Fal04].

In High-SUIF besteht der aufgebaute Syntaxbaum aus Knoten der folgenden Typen:

- for Schleife
- do-while Schleife

[4] `http://gcc.gnu.org/projects/tree-ssa/`

[5] Eine Zwischendarstellungsfamilie, in der jede Variable genau einmal zugewiesen wird. Dies erleichtert zahlreiche Optimierungsalgorithmen.

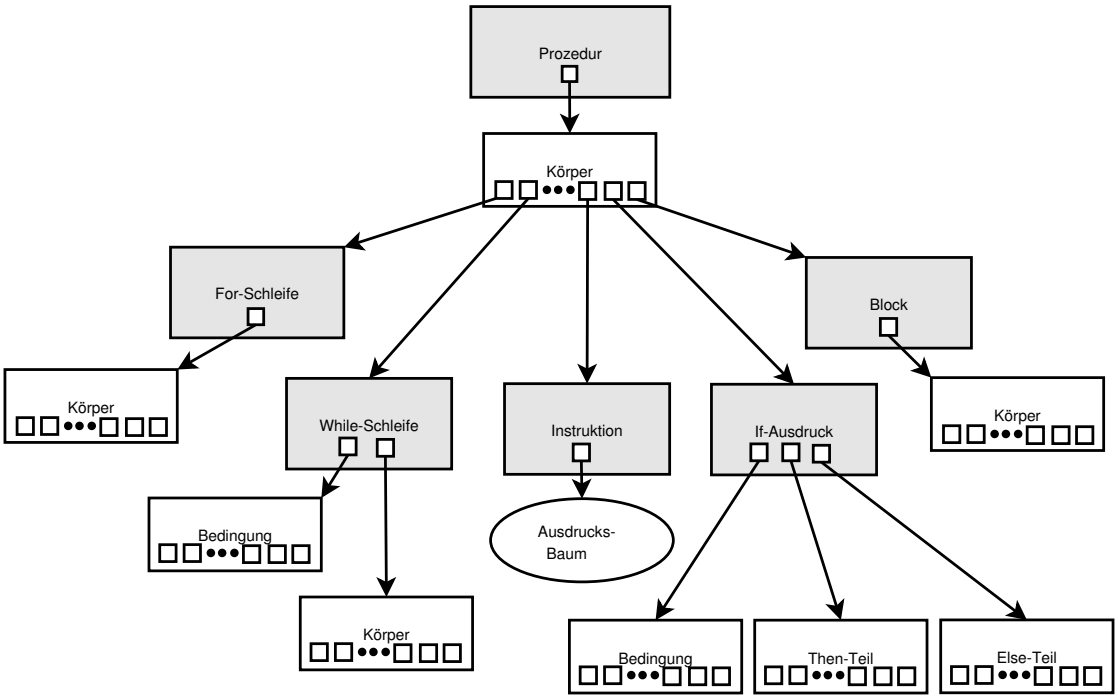

Abbildung 2.1: Der SUIF Syntaxbaum

- if Anweisung
- Codeblöcke

Eine Übersicht über die Struktur findet sich in Abbildung 2.1.

Aus dem Syntaxbaum wird später wieder C-Code erzeugt. Dabei werden Eigenschaften wie Variablennamen u. ä. beibehalten. SUIF stellt damit ein sehr interessantes Framework für Quellcodeoptimierungen dar.

2.3 Laufzeitmessungen

2.3.1 Verwendete Plattformen

Für alle Messungen, die im Rahmen der Diplomarbeit durchgeführt worden sind, kamen vier unterschiedliche Plattformen zum Einsatz. Die erste Architektur ist der Intel Pentium III mit 650 MHz. Es wurde auf einem SuSE Linux-System mit dem GNU GCC 3.3.1 und dem Intel ICC 7.1 gemessen. Als zweiter Prozessor wurde ein ARM7TDMI mit 33 MHz auf einem AT91EB01 Entwicklungsboard der Firma Atmel verwendet. Er wurde einmal im ARM- und einmal im Thumb-Modus betrieben. Als Compiler kam ein GCC 3.3.2 zum Einsatz, der unter Linux als Crosscompiler übersetzt wurde. Als libc wurde mit der newlibc in der Version 1.11.0 gearbeitet.

Um das Atmel Board in Betrieb zu nehmen, waren weitere Anpassungen nötig. Zunächst mußte ein Linker Script geschrieben werden, das die Speicheradressen für RAM und ROM des Atmel Boards beschreibt. Diese Informationen wurden mit Hilfe des `objdump` Tools aus einem vom Windows ARM Compiler erzeugten Programm gewonnen. Dann mußte die Exit-Routine der libc so verändert werden, daß der Prozessor in jedem Fall zurück in den ARM Modus geschaltet wird. Ist dies nicht der Fall, so verläßt der Prozessor im Thumb Modus die libc und stürzt anschließend ab, weil die nächsten Instruktionen im ARM Befehlssatz vorliegen. Nach diesen Modifikationen konnten ELF Images mit dem `arm-elf-gdb` auf das Board über die serielle Schnittstelle geladen werden.
Die Hardware im Überblick:

Chip	Register
Pentium III	8 nutzbare 32-Bit general purpose Register
ARM7	16 nutzbare 32-Bit Register
THUMB	8 nutzbare 32-Bit Register

Die Plattformen im Überblick:

Chip	Compiler
Pentium III	GNU GCC 3.3.1
Pentium III	Intel ICC 7.1
ARM	GNU GCC 3.3.2 (Crosscompiler)
Thumb	GNU GCC 3.3.2 (Crosscompiler)

2.3.2 Meßmethoden

Auf dem Intel Pentium III wurde mittels der `times()` C Funktion die Zeit gemessen, die für die betrachtete Berechnung nötig war. Times liefert eine Struktur zurück:

```
struct tms {
  clock_t tms_utime;  /* user time */
  clock_t tms_stime;  /* system time */
  clock_t tms_cutime; /* user time of dead children */
  clock_t tms_cstime; /* system time of dead children */
};
```

Dabei wurde nur die `tms_utime` berücksichtigt, da für den Benchmark nur die Zeit der Berechnung interessant ist. Während eines Benchmarks liefen keine weiteren Anwendungen.

Auf dem ARM Prozessor wurde die Laufzeit mittels der eingebauten Performancecounter gemessen. Die Zeiten sind somit zyklengenau.

2.3.3 Verwendete CFLAGS

Für den CAVITY Benchmark wurde als Compileroption für den GCC und den ICC -O2 verwendet. Eine Optimierung mit -O3 ist nur wenig interessant, da -O3 unter dem GCC nur automatisches Funktionsinlining (-finline-functions) und Rename Registers (-frename-register)[6] aktiviert. Beide Optimierungen werden nicht durch CSE Optimierungen beeinflußt.

Kontrollmessungen ergaben, daß die Verhältnisse der Laufzeiten und die erzielten Optimierungen zwischen -O2 und -O3 beim GCC im wesentlichen gleich sind. Beim Intel ICC konnte zwischen einer -O2 und -O3 Optimierung kein Unterschied in der Laufzeit gemessen werden.

2.4 Lua Interpreter

Applikationen werden häufig in einem anderen Kontext eingesetzt, als der Autor vorhergesehen hat. In einem solchen Fall ist es sinnvoll, wenn das Programm flexible Mechanismen zur Verfügung stellt, um es zu skripten und zu erweitern. Eine Möglichkeit dazu stellt das Einbetten einer Programmiersprache in die Applikation dar. Damit kann der Benutzer komplexe Aufgaben in Form von einfachen Skripten erledigen. Um die Klassifizierung von CSEs flexibel zu gestalten, wurde hier die Programmiersprache Lua [Ier03] eingebettet.

Lua ist eine imperative Programmiersprache, die speziell für den Einsatz als eingebettete Sprache in größeren Anwendungen konzipiert wurde. Sie ist einfach, klein und doch mächtig genug, um als vollwertige Programmiersprache zu gelten. Die Syntax ist an Pascal angelehnt, die Datentypen sind aber deutlich mächtiger. Lua unterstützt assoziative Arrays, hat dynamische Typen, wird als Bytecode ausgeführt und hat automatische Speicherverwaltung mit Garbage Collection.

Ein Beispiel für ein Luascript:

```
-- Dies ist ein Kommentar
function factorial(n)
         if n == 0 then
            return 1
         else
            return n * factorial(n - 1)
         end
end
```

Lua wird an der Katholischen Universität von Rio de Janeiro seit 1993 als Freie Soft-

[6] Eine Technik, bei der eine Variable in verschiedenen Registern gehalten werden kann und von der Architekturen mit vielen Registern am meisten profitieren. Für Details siehe [Sta99].

ware unter der MIT Lizenz[7] entwickelt. Die aktuelle Version 5.0 stammt aus dem Jahr 2003. Ausführliche Informationen zu Lua finden sich in [Ier03] und im Internet[8].

Lua wird in verschiedenen proprietären und freien Projekten erfolgreich eingesetzt. Durch die geringen Ressourcenanforderungen eignet sich Lua auch als Sprache in eingebetten Systemen.

2.4.1 Einbetten von Lua

Das Design von Lua erlaubt ein leichtes Einbetten in größere Programme. Lua und das Hostprogramm kommunizieren mit Hilfe einer stackbasierten C API. Die Lua-Bibliothek ist dabei nur 160KB groß.

Beispielhaft wird im folgenden beschrieben, wie ein Lua Skript aus einem C bzw. C++ Programm aufgerufen werden kann.

Zunächst wird ein Lua Ausführungskontext vom Typ `lua_State` mittels lua_open() erzeugt. Dann schreibt das Hostprogramm Daten oder Tabellen auf den Stack des Luakontextes mit den Kommandos lua_pushnumber(), lua_pushliteral() und lua_setglobal(). Diese Daten stehen dann als Variablen oder Tabellen im Lua Script zur Verfügung. Dann wird das eigentliche Lua Skript mittels luaL_loadfile() (luaL steht für Funktionen aus der Lua-Library) geladen und per lua_pcall() ausgeführt. Die Ergebnisse des Scripts können per lua_gettop() direkt vom Stack des Lua Kontextes gelesen werden. Zum Schluß wird der Kontext per lua_close() aus dem Speicher entfernt.

2.5 Neuronale Netze

Einer groben Schätzung nach besteht das menschliche Gehirn aus 10 Milliarden Neuronen, die jeweils im Mittel mit mehreren tausend anderen Neuronen verknüpft sind. Zwischen den Neuronen werden Nachrichten ausgetauscht. Dabei wird allerdings nicht jede Nachricht sofort weitergeleitet, sondern es wird abgewartet, bis eine bestimmte kritische Schwelle überschritten ist. Dann „feuert“ das Neuron. Vereinfacht gesagt lernt das Gehirn, indem es die Verbindungen zwischen den Neuronen anlegt und gewichtet. Nach diesem natürlichen Vorbild werden künstliche Neuronale Netze entwickelt. Sie sind insbesondere für Klassifizierungsaufgaben gut geeignet. Außerdem besitzen sie die Fähigkeit, von einer kleinen Menge von Eingabebeispielen ausgehend zu generalisieren. Einen sehr guten Überblick über neuronale Netze bietet [Zel94].

Ein künstliches Neuronales Netz ist ein Graph, der nach dem Vorbild des menschlichen Nervensystems modelliert ist. Es besteht aus künstlichen Neuronen (Zellen), die Knoten im Graph dargestellen und miteinander verknüpft sind. Jede Verknüpfung hat dabei ein bestimmtes Gewicht, das bestimmt, ob ein Signal verstärkt oder geschwächt wird. Ein Neuron „feuert“ ein Signal ab, wenn eine bestimmte Signalstärke erreicht wird. Dann wird das Signal an alle verknüpften Nachfolgeneuronen weitergegeben, wobei die Gewichte der Verbindungen berücksichtigt werden.

[7] `http://www.opensource.org/licenses/mit-license.html`
[8] `http://www.lua.org` und `http://www.lua-users.org`

2.5.1 Neuronen

Vereinfacht kann man sich ein Neuron als eine Black-Box vorstellen, die eine Reihe von gewichteten Eingabedaten bekommt und abhängig von diesen Eingabedaten ein Ausgabedatum erzeugt.

Formal bekommt jedes künstliche Neuron j eine Menge von Eingabedaten $x_1, x_2, \ldots, x_n$. Jedes Eingabedatum erhält eine Gewichtung $w_1, w_2, \ldots, w_n$. Außerdem kann ein Schwellenwert θ_j vorgegeben werden, der überschritten werden muß, damit das Neuron vollständig aktiviert ist. Die Summe der gewichteten Eingabedaten ergibt den Aktivierungszustand a_j. Es ist also $a_j = x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_n \cdot w_n$ oder

$$a_j = \sum_{i=1}^{n} w_i \cdot x_i \tag{2.1}$$

Jedes Neuron j hat eine Ausgabe y_j, die häufig in den Intervallen $[0, 1]$ oder $[-1, +1]$ liegt. Diese Ausgabe wird mittels der Aktivierungsfunktion f_{act} bestimmt. Ein Beispiel für eine einfache Aktivierungsfunktion stellt die Auswahlfunktion dar:

$$y_j = \begin{cases} 1 & \text{falls } a_j \geq \theta_j \\ 0 & \text{sonst} \end{cases} \tag{2.2}$$

Neuronen, die mit der Auswahlfunktion arbeiten, heißen auch McCulloch-Pitts Neuronen oder Perceptron, sie wurden 1957 von Rosenblatt ([Ros58]) entwickelt.

Die Abbildung 2.2 stellt ein künstliches Neuron dar. Wenn die Aktivierungsfunktion die Auswahlfunktion ist, kann man zur Veranschaulichung in die Grafik auch statt f_{act} gleich den Schwellenwert θ eintragen, der überschritten werden muß, damit das Neuron „feuert“.

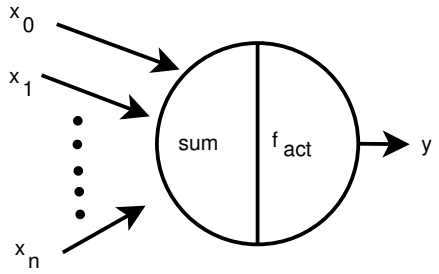

Abbildung 2.2: Ein künstliches Neuron

Eine alternative Beschreibung eines Neurons kommt ohne den Schwellenwert θ im Neuron aus. Statt θ als Eigenschaft des Neurons zu begreifen, kann es einfach als ein weiteres Eingabedatum mit einem negativen Gewicht verstanden werden, das immer angelegt wird. Man spricht dann von einem Biasneuron oder „on“-Neuron. Dann sieht die Gleichung folgendermaßen aus:

$$a_j = \left(\sum_{i=1}^{n} w_i \cdot x_i \right) - \theta \tag{2.3}$$

Diese Darstellung wird oft gewählt, da die mathematische Beschreibung einfacher ist und beim Lernen der Spezialfall des Schwellenwertes nicht gesondert berücksichtigt werden muß, da er nur ein weiteres Gewicht darstellt. Eine binäre Aktivierungsfunktion muß dann nur noch auf $a_j \geq 0$ prüfen.

Die Aktivierungsfunktion ist meist eine nichtlineare Funktion, an die zusätzliche Anforderungen wie Stetigkeit und Differenzierbarkeit gestellt werden. Neben der binären Schwellenwertfunktion, die heute nur noch selten zum Einsatz kommt, da sie an der Sprungstelle nicht differenzierbar ist, sind Aktivierungsfunktionen wie der tangens hyperbolicus (2.4) oder die logistische Funktion (2.5) gängig (vgl. Abbildung 2.3).

$$f(a) = \tanh(a) \tag{2.4}$$

$$f(a) = \frac{1}{1 + e^{-a}} \tag{2.5}$$

Diese Funktionen nennt man Sigmoide (S-förmige). Sie lösen zusätzlich das Problem, daß ein Netz sowohl mit großen als auch mit kleinen Eingabedaten zurecht kommen muß. Bei kleinen Signalen soll es dabei sensibler reagieren als bei großen. Um den Schwellenwert haben die sigmoiden Funktionen daher ihre größte Sensibilität.

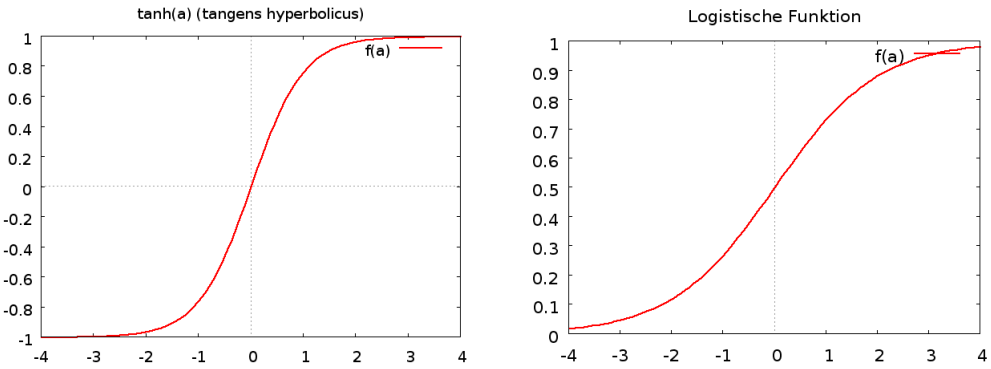

Abbildung 2.3: Gängige Aktivierungsfunktionen

Beispiel

Betrachten wir als Beispiel ein Neuron mit zwei Eingaben, die jeweils mit Eins gewichtet sind. Als Aktivierungsfunktion wird die Auswahlfunktion 2.2 mit dem Schwellenwert $\theta = 1,5$ gewählt. Für die vier möglichen Eingabewerte werden folgende Ausgabewerte erzeugt:

x_1	x_2	a	y
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

Damit kann dieses Neuron die UND Verknüpfung berechnen.

Geometrische Interpretation

Man kann ein Neuron mit einer Auswahlfunktion auch geometrisch interpretieren. Man stellt sich dann die vier möglichen Eingaben des Beispiels als Punkte in einem zweidimensionalen Graphen vor. Aus $w_1 \cdot x_1 + w_2 \cdot x_2 = \theta$ ergibt sich durch Einsetzen der Werte und Umformen: $x_2 = -x_1 + 1,5$. Der Graph dieser Gleichung stellt die Grenze im Raum dar, an dem sich die Klassifizierung des Neurons verändert. Der Raum der vier möglichen Eingaben wird also in zwei Bereiche geteilt, die die Ausgabe des Neurons darstellen. Der eingefärbte obere Bereich in Abbildung 2.4 stellt dabei den Ausgabewert Eins dar, der andere den Wert Null.

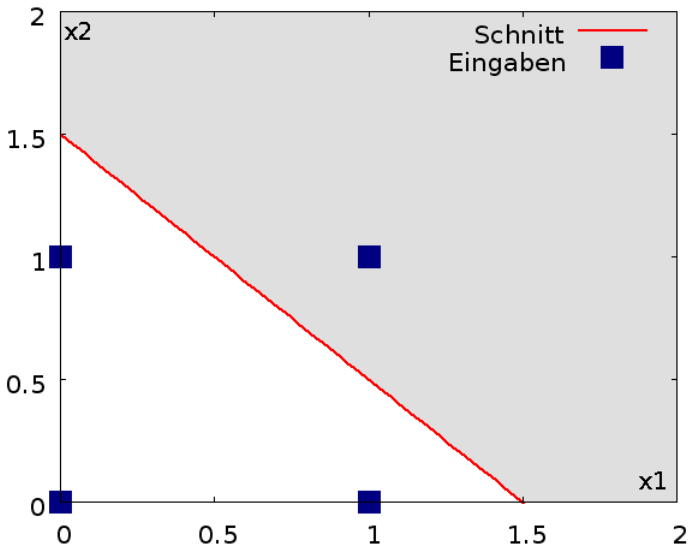

Abbildung 2.4: Lineare Separierbarkeit der UND Funktion

Dies stellt ein allgemeines Prinzip von Neuronen mit Auswahlfunktion dar. Eine Menge von N Eingaben stellt Punkte im N-dimensionalen Raum dar. Wenn diese Punkte durch eine Hyperebene geschnitten werden können, so existiert auch eine passende Menge von Gewichten und ein θ, mit dem ein Neuron erzeugt werden kann, das dann genau diesen Schnitt erzeugt. Man spricht auch von linearer Separierbarkeit.

2.5.2 Struktur Neuronaler Netze

Die Zellen werden untereinander durch ein gewichtetes Verbindungsnetzwerk verknüpft. Dabei gibt die Propagierungsfunktion $net_j(t)$ an, wie die Netzeingabe eines Neurons aus den Ausgaben der anderen Neuronen und den Verbindungsgewichten berechnet wird.

In dieser Arbeit werden multilayer feedforward Netze verwendet, die die am weitesten verbreitete Art von neuronalen Netzen darstellen. Ein solches Netz ist in Schichten angeordnet. Es gibt eine Eingabeschicht, eine Ausgabeschicht und dazwischen findet sich eine beliebige Anzahl von verdeckten Schichten. Charakteristisch ist, daß Verbindungen nur von einer Schicht zur nächsten verlaufen. Es gibt auch andere Arten von Netzen, die beliebige Verbindungsarten zulassen[9], diese sind aber in erster Linie dann interessant, wenn ein Zeitverhalten bei der Klassifizierung zu berücksichtigen ist. In Abbildung 2.5 findet sich ein Beispiel für ein Netzwerk mit vier Eingabeneuronen, acht Neuronen in einer verdeckten Schicht, und einem einzigen Ausgabeneuron. Dieses Netz ist vollständig verbunden, d. h. jedes Neuron in einer Schicht ist mit jedem Neuron in der nächsten Schicht verknüpft. Ein Netz muß nicht vollständig verknüpft sein, jedes vollständig verknüpfte Netz kann aber ein beliebiges weniger verknüpftes Netz simulieren, indem die überzähligen Verbindungen mit Null gewichtet werden.

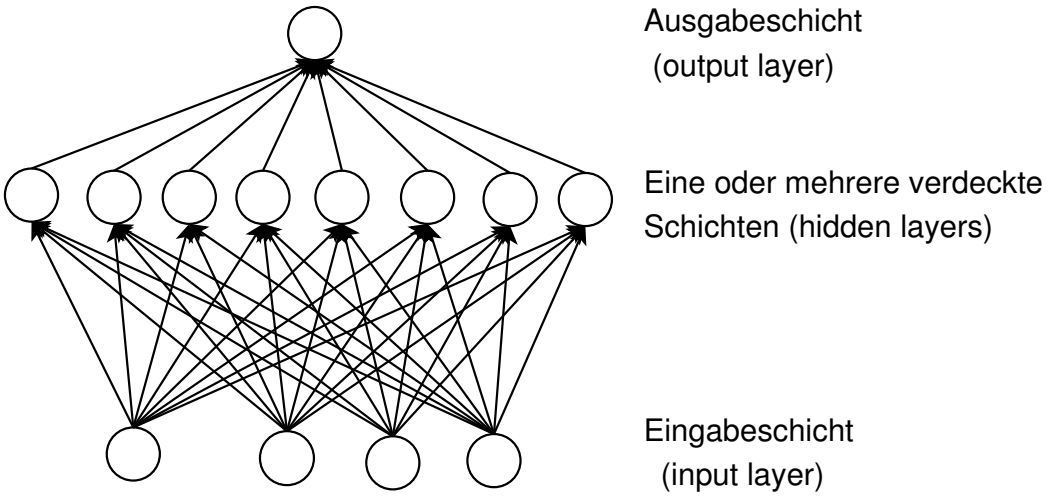

Abbildung 2.5: Ein feedforward Netzwerk mit drei Zellschichten

Minsky und Papert haben 1969 [MP69] gezeigt, daß ein neuronales Netz mit einer einzigen Schicht bestimmte Probleme nicht lösen kann. Das einfachste Beispiel stellt die XOR Funktion dar. Hier müssen zwei getrennte Regionen im Raum zu einer einzigen Klasse verschmolzen werden.

[9] Eine ausführliche Auswahl findet sich in [Zel94]

Beispiel

In diesem Beispiel soll ein Netz dargestellt werden, mit dem man die XOR Funktion berechnen kann. Die Funktion hat die Form:

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Hier kommt man nicht mehr mit einem Neuron aus, da hier keine lineare Separierbarkeit möglich ist.

Zunächst ein Netzwerk mit Schwellenwerten. Dabei kommt für alle Neuronen als Aktivierungsfunktion die Auswahlfunktion 2.2 zum Einsatz.

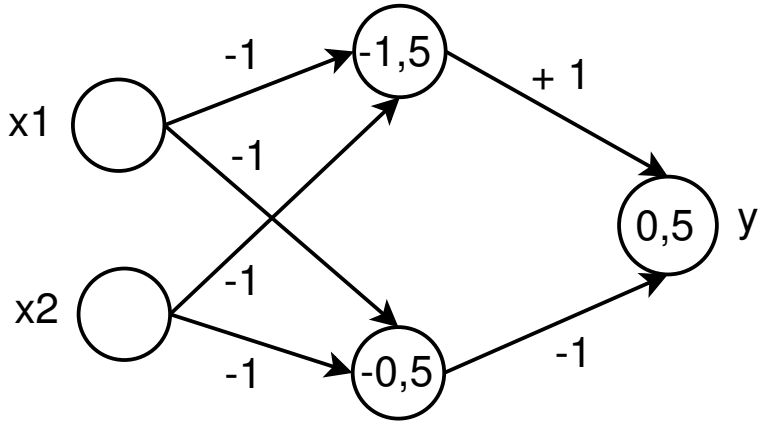

Abbildung 2.6: Neuronales Netz mit Schwellenwerten

Die Berechnung wird beispielhaft für die Eingabe (0,1) vorgenommen. Am ersten (oberen) versteckten Neuron wird der Wert $a_{hidden1} = (w_1 \cdot x_1 + w_2 \cdot x_2)$ berechnet, also $(-1 \cdot 0) + (-1 \cdot 1) = -1$. Da das Ergebnis größer $\theta = -1,5$ ist, wird eine $y_{hiden1} = 1$ vom ersten verdeckten Neuron weitergegeben. Das zweite verdeckte Neuron berechnet $(-1 \cdot 0) + (-1 \cdot 1) = -1$. Dieser Wert ist kleiner $\theta = -0,5$ und damit wird $y_{hidden2} = 0$ vom zweiten verdeckten Neuron weitergegeben. Das Ausgabeneuron berechnet nun $(+1 \cdot 1) + (-1 \cdot 0) = 1$. Dieser Wert überschreitet $\theta = 0,5$ und liefert damit den Wert 1 am Ausgang.

Analog liefert die Eingabe (1,1) die Werte $a_{hidden1} = (-1 \cdot 1) + (-1 \cdot 1) = -2$, und $a_{hidden2} = -2$. Damit wird keines der beiden θ überschritten und beide Neuronen liefern eine 0 als Ausgabe. Somit ist $a_y = (+1 \cdot 0) + (-1 \cdot 0) = 0$. Dieser Wert ist kleiner als $\theta = 0,5$ und am Ausgang wird der Wert 0 ausgegeben.

Abbildung 2.7 zeigt ein äquivalentes Netzwerk mit „On“-Neuron statt Schwellenwer-

ten. Die Aktivierungsfunktion wird so modifiziert, daß statt θ immer der Wert 0 überschritten werden muß.

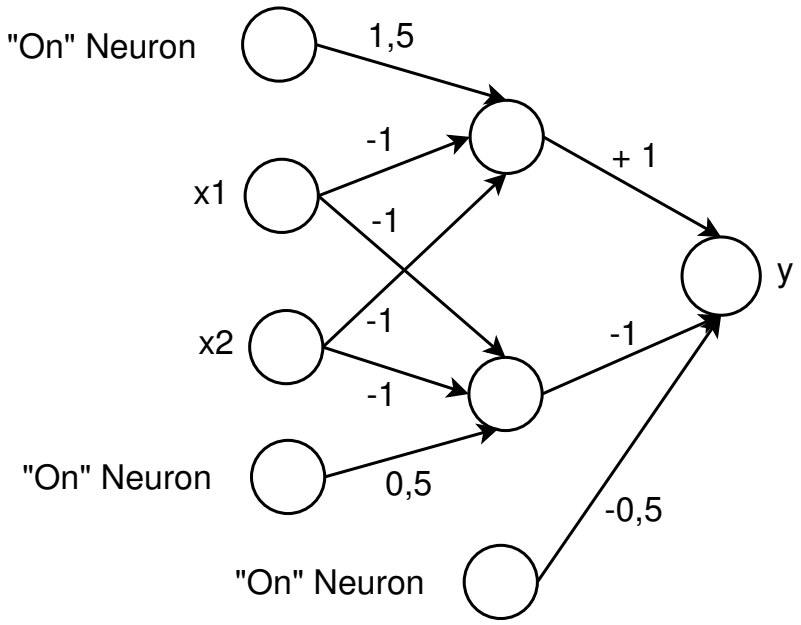

Abbildung 2.7: Neuronales Netz mit „On“-Neuron

Auch hier läßt sich leicht prüfen, daß die gewünschten Ergebnisse erreicht werden.

Geometrische Interpretation

Geometrisch betrachtet teilt das obere verdeckte Neuron den Raum durch die Gerade $x_2 = -x_1 + 1,5$. Das Neuron kann als boolesche Funktion $h_1 = (x_1 \wedge x_2)$ verstanden werden. Das untere Neuron teilt den Raum durch die Gerade $x_2 = -x_1 + 0,5$. Dies kann als boolesche Funktion $h_2 = x_1 \vee x_2$ verstanden werden. Die linke Seite der Abbildung 2.8 zeigt die beiden Geraden.

Durch das Neuron in der Ausgabeschicht werden nun die beiden Gebiete der vorherigen Schicht zusammengefaßt. Die Gerade hat den Wert $h_2 = h_1 - 0,5$ (dabei sind h_1, h_2 die Ausgaben der Neuronen der verdeckten Schicht) und stellt damit die Funktion $y = h_1 \wedge \overline{h_2}$ dar, die äquivalent zur ($x1$ xor $x2$) Funktion ist. Die rechte Seite der Abbildung 2.8 zeigt diese Gerade.

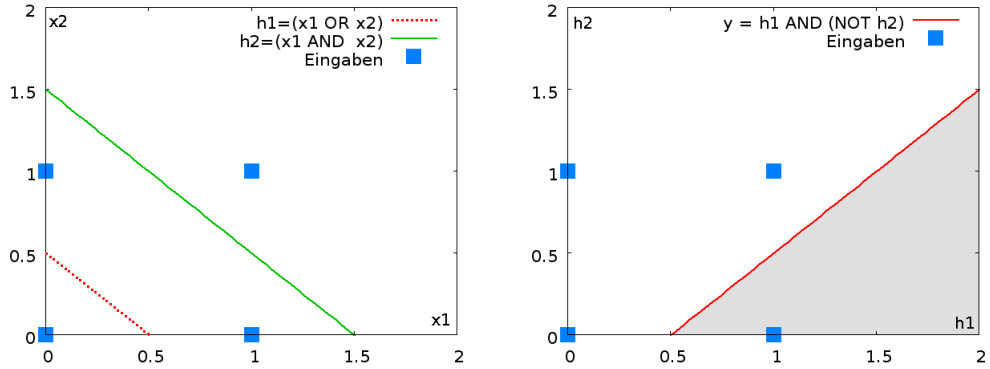

Abbildung 2.8: Die XOR Funktion

2.5.3 Lernen

Neuronale Netze werden nicht im traditionellen Sinn für eine bestimmte Aufgabe programmiert, sondern trainiert. Generell erfolgt zunächst eine Lernphase, in der mittels einer Lernregel und einer Reihe von Beispielen das Netz auf seine künftige Aufgabe vorbereitet wird. Die Auswahl geeigneter Trainingsdaten ist dabei von entscheidender Bedeutung.

Das Lernen ist dem natürlichen Lernen des Gehirns nachgebildet. Neuronale Netze lernen, indem die Gewichte der Verknüpfungen zwischen den Neuronen verändert werden. Zusätzlich wird auch der Schwellenwert θ verändert.

Beim „überwachten Lernen“ werden dem Netz Eingabedaten und die gewünschten Ergebnisse präsentiert. Die Gewichte des Netzes werden vom Lernalgorithmus nun derart verändert, daß die präsentierten Ergebnisse möglichst korrekt berechnet werden. Dabei wird versucht, den Fehler zwischen der tatsächlichen Ausgabe und der erwarteten Ausgabe zu minimieren. Nach dieser Lernphase sollte das Netz die Fähigkeit der Generalisierung gewonnen haben, d. h. es kann nun unbekannte, aber den Trainingsdaten ähnliche Eingabedaten korrekt klassifizieren.

Ein Lernalgorithmus kann z. B. wie folgt aussehen:

1. Anlegen der Eingabe
2. Verarbeiten der Eingabe
3. Vergleich des berechneten Ergebnisses mit dem gewünschten Ergebnis
4. Minimierung des Fehlers durch Modifikation der Gewichte, solange der Fehler nicht klein genug ist

Einzelne Neuronen

Ein einzelnes Neuron kann relativ einfach durch überwachtes Lernen traniert werden.

```
def train(neuron):
  a=0
  need_more_training = TRUE
  while need_more_training:
    # berechne Aktivierungszustand a des Trainingsdatums
    for (x,w) in (inputs, weights):
      a = a + (x*w)
    # berechne Ausgabe y
    y = f_act(a,theta)
    # vergleiche Fehler mit maximal erlaubtem Fehler
    if error(y, desired_value) < max_error:
      need_more_training = FALSE
      return
    # modifiziere die Gewichte
    for w in weights:
      modify_weight(w)
```

Besondere Bedeutung kommt dabei der Methode der Modifikation der Gewichte zu (`modify_weight` im Beispiel). Diese Modifikation wird durch die Lernregel bestimmt. Ein Beispiel für eine solche Regel ist die Perceptron-Lernregel, die Rosenblatt 1962 ([Ros62]) entwickelt hat. Die Veränderung der Gewichte erfolgt dabei immer zu einem bestimmten Bruchteil von der Differenz zwischen gewünschter Ausgabe (t) und berechneter Ausgabe (y). Formal: $\Delta w_i = \alpha(t - y) \cdot x_i$. Dabei gibt die Konstante α aus dem Intervall $[0, 1]$ die Lernrate an. Ein kleiner Wert führt zu konservativen Änderungen, ein großer Wert zu größeren Sprüngen bei den Gewichten. Rosenblatt konnte zeigen, daß für jede linear separierbare Eingabe nach endlich vielen Schritten eine Gewichtung gefunden wird, die diese Eingabe separiert.

Backpropagation

Um komplette Netzwerke zu trainieren, ist ein komplexerer Trainingsalgorithmus nötig.

Beim Training wird das Netzwerk mit einer Menge von Eingabedaten versorgt, für die korrekte Ausgabewerte bekannt sind. Ziel ist es nun, daß die Ausgaben des Netzes den gewünschten Ausgaben entsprechen. Allerdings ist zu starkes Training nicht erwünscht, da es zum sogenannten „over-fitting“ führt. Hier generalisiert das Netz nicht mehr, sondern berechnet ausschließlich die trainierten Werte.

Man kann das Trainieren eines Neuronalen Netzes als ein Optimierungsproblem verstehen. Es geht darum, den mittleren quadratischen Fehler der gesamten Menge der Trainingsdaten zu minimieren. Dieses Problem kann auf viele Arten gelöst werden. Eine sehr verbreitete Methode stellt das Gradientenabstiegsverfahren „Backpropagation“ dar. Dieser Algorithmus arbeitet nur mit Aktivierungsfunktionen, die differenzierbar sind. Es existieren eine Vielzahl von weiteren Verfahren und Varianten, die unter [Zel94] aufgeführt werden.

Der Backpropagation Algorithmus

Der Backpropagation Algorithmus arbeitet, indem zunächst eine Eingabe in das Netz erfolgt. Diese Eingabe wird vorwärts propagiert, dann wird am Ausgang der Fehler berechnet und im Netz zurück propagiert. Bei diesem Zurückpropagieren werden die Gewichte so angepaßt, daß der Fehler immer kleiner wird. Beim Training wird immer eine Trainingseingabe nach der anderen bearbeitet.

Als erstes wird die Eingabe angelegt und die Ausgabe des Netzes berechnet. Dann wird der Fehler e_j für ein Ausgabeneuron j berechnet:

$$e_j = t_j - y_j \tag{2.6}$$

Dabei ist t_j das gewünschte Ergebnis (teaching input) und y_j das tatsächliche Ergebnis.

Mit diesem Fehlerwert e_j wird nun eine Zahl δ_j berechnet, die zum Anpassen der Gewichte benutzt wird. Dieses δ_j berechnet sich wie folgt:

$$\delta_j = e_j \cdot f'_{act}(y_j) \tag{2.7}$$

Dabei ist f'_{act} die Ableitung der Aktivierungsfunktion.

Nachdem δ_j berechnet worden ist, kann auch ein δ_k der vorherigen Schichten berechnet werden. Der δ_k Wert der vorherigen Schicht wird aus dem δ_j Wert dieser Schicht mittels folgender Formel berechnet:

$$\delta_k = \eta \cdot f'_{act}(y_k) \cdot \sum_{j=0}^{J} \delta_j \cdot w_{kj} \tag{2.8}$$

Dabei ist J die Zahl der Neuronen in dieser Schicht, w_{kj} die Gewichtung zwischen dem k-ten und dem j-ten Neuron und η die Lernrate, die angibt, wie stark die Gewichte angepaßt werden sollen.

Mit diesen δ Werten können nun die Δw Werte berechnet werden. Diese geben an, wie stark die einzelnen Gewichte verändert werden sollen. Die Berechnung ist:

$$\Delta w_{kj} = \delta_k \cdot y_j \tag{2.9}$$

Mit diesem Wert Δw_{kj} wird nun das Gewicht w_{kj} angepaßt, indem $w_{kj} = w_{kj} + \Delta w_{kj}$ berechnet wird. Dann fährt der Backpropagation Algorithmus mit der nächsten Traininsgseingabe fort und verändert die Gewichte wie gezeigt. Dies wird so lange fortgesetzt, bis ein bestimmtes Stopp Kriterium erreicht ist. Dies ist üblicherweise eine maximale Zahl von Iterationen oder ein angestrebter mittlerer Fehler. Dabei wird die Menge der Trainingseingaben meist mehrfach durchlaufen, bis der angestrebte Fehler klein ist.

Eine Herleitung dieses Algorithmus kann in [Zel94] und [Cal03] nachgelesen werden.

3 Realisierung einer plattformabhängigen Common Subexpression Eliminierung

In diesem Kapitel wird die Realisierung des High-Level CSE Algorithmus vorgestellt. Danach werden die Erweiterungen erklärt, die nötig sind, um plattformabhängige Parameter in die Optimierung einzubeziehen. Dann wird der Feedback Algorithmus vorgestellt, mit dem Abschätzungen über die maximal zu gewinnenden Laufzeitverbesserungen möglich sind. Zum Schluß wird eine Klassifizierung der plattformabhängigen Parameter auf der Basis eines neuronalen Netzes vorgestellt.

3.1 High Level CSE Implementierung

Die CSE Implementierung, die in [Fal04] beschrieben wird, findet und eliminiert jede CSE. Dabei wird kein Unterschied zwischen den gefundenen CSEs gemacht, sie werden alle eliminiert. Damit werden auch Ersetzungen vorgenommen, die potentiell negative Auswirkungen auf die Laufzeit haben, weil jede gefundene CSE Schwächen haben kann, wie sie in Kapitel 1.4 beschrieben wurden. Die Struktur der bisherigen Implementierung wird in Abbildung 3.1 deutlich gemacht.

Als Eingabe erfolgt ein C-Quellcode, der dann an die CSE-Entdeckungs- und Eliminierungsumgebung weitergegeben wird. Hier wird der Quelltext untersucht, und jede gefundene CSE wird eliminiert. Als Ausgabe erfolgt wieder ein C-Quelltext, der dann mit jedem C-Compiler übersetzt werden kann.

Der Eliminierungs-Algorithmus arbeitet auf der Basis von C-Funktionen. Zunächst wird überprüft, ob die Funktion Seiteneffekte hat und welche globalen Symbole gelesen werden. Diese Informationen werden in die Analyse einbezogen. Anschließend wird eine Liste aller Ausdrücke erstellt, die mehr als einmal verwendet werden. Für jeden dieser

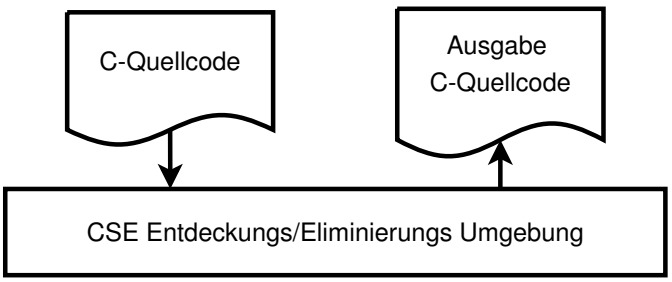

Abbildung 3.1: Übersicht über die ursprüngliche Implementierung

Ausdrücke wird die Dauer der Gültigkeit berechnet (lifetime analysis) und für die weitere Analyse gespeichert. Jetzt beginnt die eigentliche CSE Eliminierung. Jeder Ausdruck, der mehrfach verwendet wird und dessen Operanden unverändert bleiben, wird einer lokalen Variable zugeordnet, und die Vorkommen des Ausdrucks werden durch die lokale Variable ersetzt. Dabei werden die aus der Lifetime Analyse gewonnenen Informationen berücksichtigt.

Alle Vorkommen der neuen CSE Variablen werden in einer Liste zur späteren Verwendung gespeichert. Es wird in der Reihenfolge von kurzen zu längeren CSE Ausdrücken gearbeitet. Dadurch kann eine einfache CSE Teil einer komplexen CSE sein, und es kann vorkommen, daß nach dem Ende des Algorithmus eine solche einfache CSE nur noch einmal verwendet wird (nämlich von der komplexen CSE, die alle anderen Benutzungen überdeckt). CSE-Variablen, die nur noch einmal vorkommen, werden anschließend durch den zugehörigen Ausdruck ersetzt (copy propagation). Betrachten wir das folgende Codebeispiel:

```
if (x>=0 && x<N && y>=1 && y<=M-2) { ... }
if (x>=2 && x<=N-1 && y>=1 && y<=M-2) { ... }
if (y>=1 && y<M+1 && x>=1 && x<N+1) { ... }
```

Dieses Beispiel wird zunächst intern umgewandelt zu:

```
cse24= y>=1;
cse25= y<=M-2;
if (x>=0 && x<N && cse24 && cse25) { ... }
if (x>=2 && x<=N-1 && cse24 && cse25) { ... }
if (cse24 && y<M+1 && x>=1 && x<N+1) { ... }
```

Dann erkennt der Algorithmus, daß `cse24 && cse25` ebenfalls eine CSE ist und erzeugt damit den Code:

```
cse24= y>=1;
cse25= y<=M-2;
cse43= cse24 && cse25;
if (x>=0 && x<N && cse43) { ... }
if (x>=2 && x<=N-1 && cse43) { ... }
if (cse24 && y<M+1 && x>=1 && x<N+1) { ... }
```

Nun sind die CSEs korrekt identifiziert, aber `cse25` kommt nur noch einmal im Quellcode vor und wird dann per „Copy Propagation“ entfernt:

```
cse24= y>=1;
cse43= cse24 && y<=M-2;

if (x>=0 && x<N && cse43) { ... }
if (x>=2 && x<=N-1 && cse43) { ... }
if (cse24 && y<M+1 && x>=1 && x<N+1) { ... }
```

3.2 Struktur der Plattformabhängigen High-Level CSE

Um die in Kapitel 1 vorgenommenen Überlegungen in die Praxis umsetzen zu können, muß dem vorgestellten CSE Eliminierungsalgorithmus ein flexibler Mechanismus zur Klassifizierung der gefundenen CSEs zur Seite gestellt werden. Damit können CSE Eliminierungen vermieden werden, die keine Laufzeitgewinne ergeben. Diese zusätzliche Flexibilität erlaubt es, Kriterien, wie z. B. den Abstand zwischen zwei CSE Nutzungen, in die Betrachtung einzubeziehen.

Die Implementierung wird schematisch in Abbildung 3.2 auf der nächsten Seite dargestellt. Die ursprüngliche Version wird durch die PolicyEngine erweitert. Bei ihr können beliebige Policies registriert werden, die jeweils eine Eigenschaft einer gefundenen CSE untersuchen. Das können plattformabhängige Aspekte wie die Kosten des Ausdrucks sein oder plattformunabhängige wie die Zahl der Vorkommen des Ausdrucks.

Teil der PolicyEngine ist das Entscheidungsmodul, das die konkrete Entscheidung darüber übernimmt, ob eine bestimmte CSE eliminiert wird oder nicht. Dieses Entscheidungsmodul ist mit abgerundeten Ecken dargestellt, um deutlich zu machen, daß dieses Modul in Form von Plugins flexibel und austauschbar gehalten wurde. Derartige Plugins können leicht in der Sprache Lua realisiert werden. Ein Plugin behält während der gesamten Ausführung seinen Zustand. So sind auch Analysen möglich, die ein „Gedächtnis“ voraussetzen. Neben der „ad-hoc“ Implementierung des Entscheidungsmoduls wurde im Rahmen dieser Arbeit ein „intelligentes“ Plugin entwickelt, das auf der Grundlage eines neuronalen Netzes die Entscheidung darüber fällt, ob eine CSE eliminiert werden soll oder nicht.

3.2.1 Ablaufverhalten

Die PolicyEngine bekommt als Eingabe alle gefundenen CSEs der CSE Entdeckungsumgebung. Jede der gefundenen CSEs wird nun der Reihe nach von den registrierten Policies untersucht und bewertet.

Dabei liefern die Policies sowohl Untersuchungsergebnisse in konkreter Form, wie z. B. die Kosten eines Ausdruck oder den Abstand zum nächsten Vorkommen des Ausdrucks, als auch eine abstrakte Bewertung in Form einer Zahl im Intervall $[MinScore, MaxScore]$. Diese Konstanten sind mit den Werten $[-1000, +1000]$ belegt, wobei der Zahl Null der symbolische Wert „DontKnowScore“ zugewiesen wurde. Dieser Wert bedeutet, daß die Policy keine Bewertung für diese CSE abgeben kann. Die Auswertung der Ergebnisse der Policies erfolgt im Entscheidungsmodul.

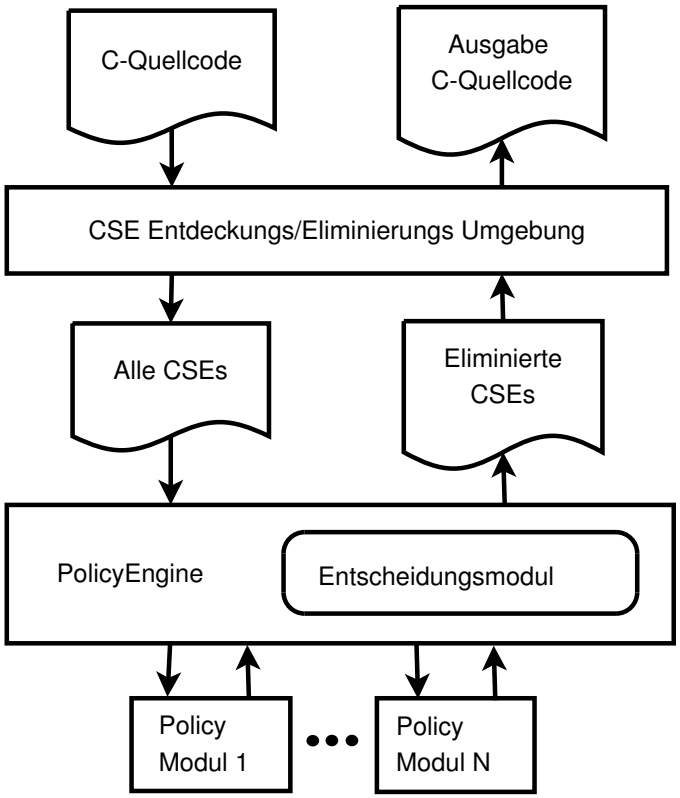

Abbildung 3.2: Übersicht über die Struktur der plattformabhängigen CSE

Die Standardimplementierung des Entscheidungsmoduls addiert die abstrakten Bewertungen und eliminiert nur dann eine CSE, wenn eine bestimmte Punktzahl überschritten wird oder wenn MaxScore erreicht ist. Durch den Abbruch bei MaxScore kann eine Policy die übrigen Policies dominieren, wenn z. B. ein Ausdruck gefunden wurde, der so aufwendig ist, daß eine Eliminierung in jedem Fall sinnvoll ist. Umgekehrt wird die CSE ignoriert, wenn die vorgegebene Punktzahl nicht erreicht wird oder wenn MinScore berechnet wird. Wenn diese Standardimplementierung nicht flexibel genug für eine bestimmte Untersuchung ist, kann sie durch eine eigene Implementierung auf der Basis eines Lua Skriptes ersetzt werden. Dabei stehen alle konkreten Ergebnisse der Policies (wie z. B. die Kosten eines Ausdrucks) zur Verfügung.

Wenn die Entscheidungen, welche CSEs eliminiert werden, abgeschlossen ist, liefert die PolicyEngine das Ergebnis zurück an die CSE-Eliminierungsumgebung, die dann die tatsächlichen Eliminierungen im Quellcode vornimmt und danach den Ergebnisquellcode ausgibt.

3.2.2 Bewertung

Die Policies können gefundene CSEs isoliert untersuchen und zusätzlich auch jedes einzelne Vorkommen eines Ausdrucks betrachten, um so Informationen über den Kontext der Benutzung zu sammeln. Sie sind dabei zustandslos, d. h. sie haben kein Gedächtnis der vorher getroffenen Entscheidungen. Betrachten wir das folgende Beispiel:

```
cse0 = (x-1)%3;
k[cse0] = w;
y = cse0 * z;
// 1000 Zeilen Code, ohne x zu modifizieren
if(cse0>8) { .. }
```

Hier könnte z. B. die isolierte Betrachtung von `cse0=(x-1)%3` die Kosten dieses Ausdrucks auf der untersuchten Plattform berechnen und so abschätzen, ob eine Eliminierung sinnvoll ist oder ob der Ausdruck so einfach ist, daß eine Neuberechnung schneller ist. Eine Betrachtung der Vorkommen liefert zusätzliche Informationen, wie beispielsweise, daß im dritten Vorkommen cse0 in einem `if()` Ausdruck verwendet wird. Auch andere Informationen, wie der Abstand zum nächsten Vorkommen sind nur durch Betrachtung der konkreten Vorkommen zu finden. Im Beispiel ist das erste und das zweite Vorkommen der Variable `cse0` sicher sinnvoll. Das dritte Vorkommen ist fraglich, da hier über eine sehr lange Zeit ein Register belegt wird oder die Variable `cse0` im Speicher abgelegt werden muß. Beide Möglichkeiten sind nicht optimal. Die Eliminierung des dritten Vorkommens von cse0 ist also vermutlich nicht sinnvoll.

Zunächst wird jede CSE immer isoliert betrachtet und dabei werden Aspekte, wie die Anzahl der Vorkommen oder die Kosten des Ausdrucks, betrachtet. In einigen Fällen genügt das, um eine Entscheidung über das Eliminieren zu fällen (wenn z. B. die CSE sehr komplex ist). Wenn es nicht genügt (weder MaxScore noch MinScore wird erreicht), so läßt die PolicyEngine jedes einzelne Vorkommen untersuchen.

Eine Policy verhält sich also sowohl aktiv, indem sie anhand ihrer Konfiguration eine abstrakte Bewertung der untersuchten CSE vergibt, als auch passiv, indem sie die gefundenen Untersuchungsergebnisse zum Entscheidungsmodul zur optionalen genaueren Analyse weiterleitet. Im obigen Beispiel könnte eine Policy, die die Kosten eines Ausdrucks bewertet, als konkretes Ergebnis die Summe der Kosten einer Subtraktion mit Konstanten und einer Modulo Operation mit Konstanten auf dieser Architektur berechnen. Gleichzeitig liefert sie eine abstrakte Bewertung im Intervall $[MinScore, MaxScore]$. Dieser Wert könnte hier z. B. $0.75 \times MaxScore$ sein, da eine Modulo Operation auf den meisten Architekturen aufwendig ist.

3.2.3 Plugin-Struktur des Entscheidungsmoduls

Der Vorteil der Plugin Struktur des Entscheidungsmoduls liegt darin, daß auf diese Weise beliebige Verknüpfungen der konkreten Ergebnisse der Policies möglich sind. Das

Problem der modularen Struktur der Policies ist, daß sie jeweils nur einen Aspekt einer CSE überprüfen. Um eine komplexe Entscheidung zu realisieren, wurde der Plugin-Mechanismus des Entscheidungsmoduls geschaffen. Werden nur einfache Aspekte untersucht, so reicht die Standardimplementierung des Entscheidungsmoduls aus. Hier werden die Bewertungen der einzelnen Policies additiv verknüpft.

Um komplexe Entscheidungen zu ermöglichen, kann eine eigene Implementierung in der Programmiersprache Lua geschrieben werden. Dabei liegen alle Informationen über die jeweils untersuchte CSE in Form von Tabellen vor. So ist maximale Flexibilität möglich, wie die Implementierung eines Entscheidungsmoduls auf der Basis eines neuronalen Netzes zeigt (vgl. Kapitel 3.5).

3.3 Policies

Die Implementierung der Analysen und Bewertungen erfolgt über ein Policy-Framework, das in C++ geschrieben ist. In der globalen systemweiten PolicyEngine können Policies registriert werden. Die PolicyEngine ruft für jede gefundene CSE Methoden der Policies auf, die die CSE isoliert und in dem Kontext ihrer Vorkommen betrachtet. Dabei kann die Policy DontKnowScore zurückliefern, wenn sie den entsprechenden Aspekt nicht untersuchen kann. Eine Betrachtung des Abstands ist z. B. auf der Ebene der isolierten CSE nicht sinnvoll, umgekehrt genügt es, die Kosten einer CSE einmal zu bestimmen und nicht für jedes Vorkommen erneut. Jede Policy trägt die gefundenen Ergebnisse in eine interne Lua-Tabelle ein, die optional von dem Entscheidungsmodul ausgewertet werden kann. Zusätzlich wird eine abstrakte Bewertung der CSE zurückgeliefert. Als Standardrückgabewerte werden die Ganzzahlwerte `MinScore`, `MaxScore` und `DontKnowScore` vordefiniert. Es sind aber beliebige Zwischenwerte möglich.

Mit dem Standardentscheidungsmodul in der PolicyEngine bricht die Untersuchung ab, wenn die Summe der abstrakten Bewertungen MinScore unterschreitet oder MaxScore überschreitet. Bei MinScore wird die CSE ignoriert, der Ausdruck bleibt unverändert. Bei MaxScore wird sie eliminiert. Liefert die Untersuchung auf der Ebene der isolierten CSE kein Ergebnis (DontKnowScore), so wird jedes Vorkommen einzeln untersucht. Wird hier MaxScore zurückgeliefert, so wird die Eliminierung natürlich nur an dieser Stelle durchgeführt.

3.3.1 Konfiguration

Die Policies können durch eine Konfigurationsdatei parametrisiert werden. Dabei leitet der PolicyName in eckigen Klammern einen Konfigurationsblock zur entsprechenden Policy ein. Anschließend können Parameter mittels Schlüssel=Wert Paaren konfiguriert werden.

Jede Policy muß dabei die Schlüssel `active=<bool>` und `dryRun=<bool>` unterstützen. Dabei legt `active` fest, ob die Policy aktiviert ist. Gefundene CSEs werden nur an aktive Policies zur Bewertung übergeben. Der Schlüssel `dryRun` gibt an, ob sich die Policy passiv verhalten soll. In diesem Modus trägt sie nur ihre gefundenen Ergebnisse in die

interne Lua-Tabelle ein, liefert aber keine abstrakte Bewertung an die PolicyEngine. Dies ist sinnvoll, wenn das Entscheidungsmodul per Plugin so verändert wird, daß es alle Entscheidungen anhand der konkreten Ergebnisse der Policies vornimmt und damit keine Empfehlungen in Form von abstrakten Bewertungen benötigt.

3.3.2 Realisierte Policies

Im folgenden werden alle Policies zur Realisierung einer plattformabhängigen CSE sowie ihre Parameter vorgestellt.

UseCountPolicy

Die UseCountPolicy zählt die Anzahl der Vorkommen einer CSE und stellt die einfachste sinnvolle Policy dar. Die Zahl der Vorkommen ist ein wichtiger Parameter, da sie die Zahl der eingesparten Berechnungen angibt.

Es gibt mehrere Arten des Zählens, die flexibel konfiguriert werden können:

1. Normal: Zählt die Zahl der Vorkommen.

2. GetAllUses=1: Addiert die Zahl der Vorkommen durch andere CSEs zu der CSE hinzu. Kommt also die einfache CSE cse0 in der komplexen cse1 vor, so wird zu der Zahl der Nutzungen von cse0 Nutzungen auch die Zahl der Nutzungen von cse1 addiert. Dadurch werden einfache CSEs bevorzugt. Komplexe CSEs dagegen, die aus mehreren einfachen bestehen, werden ignoriert. Wenn z. B. `cse1=cse0+8;` gefunden wird und cse1 sechsmal vorkommt, cse0 zweimal, so hat cse0 in diesem Zählmodus eine Nutzungszahl von $6+2 = 8$, da es ja sechsmal von cse1 mitbenutzt wird.

3. CountSubCSEs=1: Wird diese Zählung aktiviert, so wird die Zahl der Vorkommen von enthaltenen CSEs mitgezählt. Wenn also cse1 die cse0 enthält, so wird die Zahl der Vorkommen von cse0 zur Zahl der Nutzungen von cse1 addiert. Durch diese Zählung werden zwei Arten von CSEs bevorzugt: Einfache mit einem großen Use-Count und komplexe CSEs, die aus einfachen CSEs bestehen, die wiederum einen großen UseCount haben. Wenn z. B. `cse1=cse0+8;` gefunden wird und cse1 sechsmal vorkommt, cse0 zwei Mal, so hat cse1 in diesem Zählmodus eine Nutzungszahl von $2 + 6 = 8$, da die Nutzungen von cse0 mitgezählt werden. `cse0=2` bleibt unverändert.

Mit der Option `MinUseCount` kann die minimale Zahl der Nutzungen festgelegt werden. Wird sie nicht erreicht, so liefert die Policy MinScore zurück, sonst eine positive Bewertung.

Die Zahl der gefundenen Vorkommen wird in die Lua-Tabelle als UseCountPolicy eingetragen. Sie kann damit von einem alternativen Entscheidungsmodul direkt ausgelesen und verwendet werden. Soweit im folgenden nicht anders angegeben, wird die normale Zählweise verwendet.

DistancePolicy

Die DistancePolicy untersucht den Abstand zwischen zwei CSE-Vorkommen. Dabei wird für jedes Vorkommen der Abstand zum nächsten gezählt und bewertet. Diese Policy wird immer im Kontext-Modus von der PolicyEngine betrieben.

Der Abstand ist ein wichtiger Parameter. Wird er zu groß, so belegt die Variable über ihre gesamte Lebenszeit ein Register, das nicht für andere Aufgaben verwendet werden kann. Alternativ muß die Variable in den Speicher ausgelagert werden. Beide Möglichkeiten schaden der Laufzeit. Mit dieser Policy können zu große Abstände korrigiert werden, indem eine weitere CSE-Variable eingeführt wird und damit die Lebenszeit explizit geteilt wird. Beispiel:

```
w = a + b + 1;                          cse0 = a + b;
x = a + b + 2;                          w = cse0 + 1;
/* viel Code, Abstand 200 */    ⇒       x = cse0 + 2;
y = a + b + 3;                          /* viel Code, Abstand 200 */
z = a + b + 4;                          cse0_1 = a + b;
                                        y = cse0_1 + 3;
                                        z = cse0_1 + 4;
```

Abbildung 3.3: Distance Policy

Bei einer konventionellen Eliminierung würde cse0 über das gesamte Codefragment gültig sein. Die DistancePolicy sorgt für das Beispiel aus Abbildung 3.3 dafür, daß stattdessen der Ausdruck `a+b` in zwei lokalen Variablen cse0 und cse0_1 gehalten wird. Hierdurch wird eine Auftrennung der Lebensdauer des Ausdrucks `a+b` erreicht. Infolge dessen hat ein nachgeschalteter Compiler die Möglichkeit zu erkennen, daß der Ausdruck `a+b`, der in cse0 gespeichert ist, nicht über den gesamten durch den Kommentar im Beispiel repräsentierten Code lebendig zu halten ist. Die DistancePolicy reicht auf diese Weise Detail-Informationen über Lebensdauern an den Register-Allokator eines Compilers weiter.

Für die Bewertung des Abstandes ist zunächst ein Abstandsbegriff erforderlich. Dieser wird als die Zahl der SUIF-Instruktionen zwischen zwei Vorkommen einer CSE im abstrakten SUIF-Syntaxbaum definiert. Er wird ermittelt, indem die Instruktionen auf dem kürzesten Pfad im Syntaxbaum zwischen den beiden Vorkommen gezählt werden.

In der Abbildung 3.4 sind zwei Vorkommen einer CSE innerhalb eines abstrakten SUIF Syntaxbaumes dargestellt. Der Algorithmus ermittelt zunächst den ersten gemeinsamen Elter-Knoten. Dazu wird vom ersten Vorkommen an im Baum nach oben iteriert bis zum Knoten, der die gesamte Prozedur repräsentiert und jeder besuchte Knoten markiert. Danach wird vom zweiten Vorkommen aus ebenfalls nach oben iteriert und jeweils geprüft, ob der besuchte Knoten schon im ersten Durchlauf markiert wurde. Ist das der Fall, so ist der erste gemeinsame Elter-Knoten gefunden. Es wird immer ein Elter-Knoten gefunden, dieser kann allerdings die Wurzel (Prozedur Knoten) sein. Vom gemeinsamen

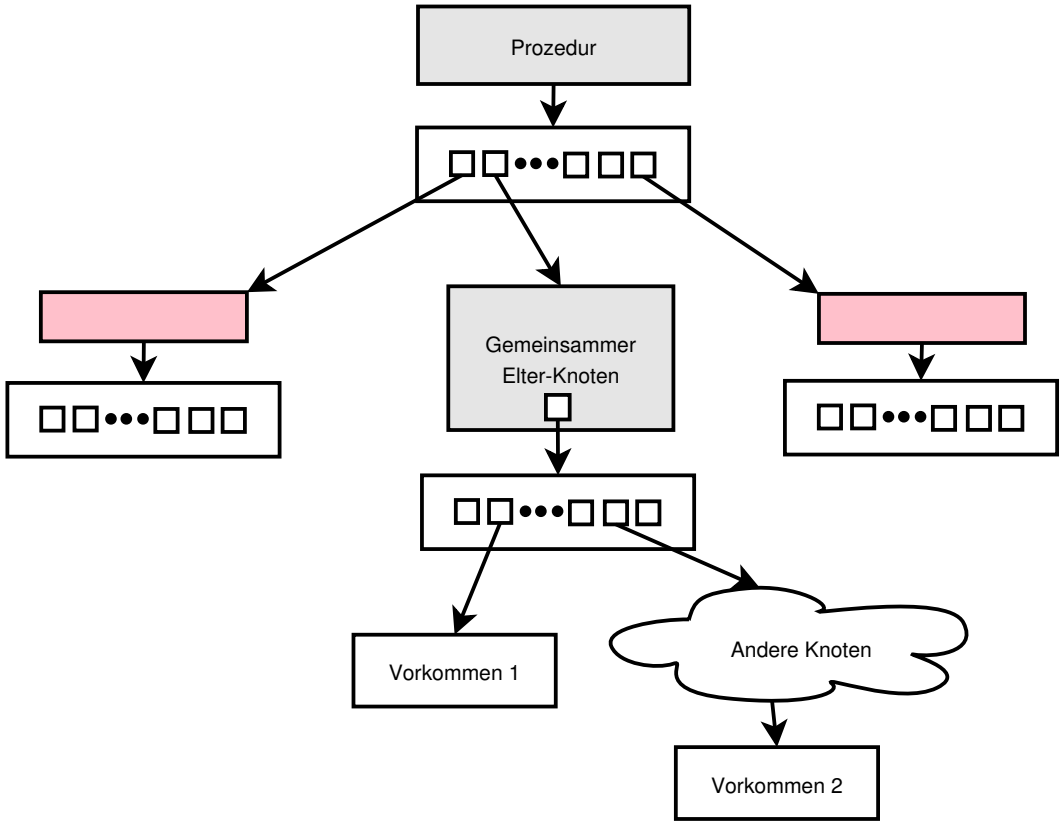

Abbildung 3.4: Berechnung des Abstandes

Elter-Knoten wird nun, rekursiv für jeden Knoten auf dem Pfad zwischen Vorkommen 1 und Vorkommen 2, die Anzahl der Instruktionen gezählt.

Betrachten wir als Beispiel den folgenden Code:

```
int a,b,d,x,y;
x = a + b + 1;
d = x + y;
y = a + b + 2;
```

Er wird zu

```
int a,b,x,y;
cse0 = a + b;
x = cse0 + 1;
d = x + y;
y = cse0 + 2;
```

Der Abstand zwischen den Vorkommen stellt die Zahl der Instruktionen dar. Diese sind in SUIF drei Additionen und zweimal das Laden einer Konstanten (die Werte eins und zwei). Der Abstand beträgt also fünf Instruktionen, die in der Abbildung 3.5 durch Kreise repräsentiert sind.

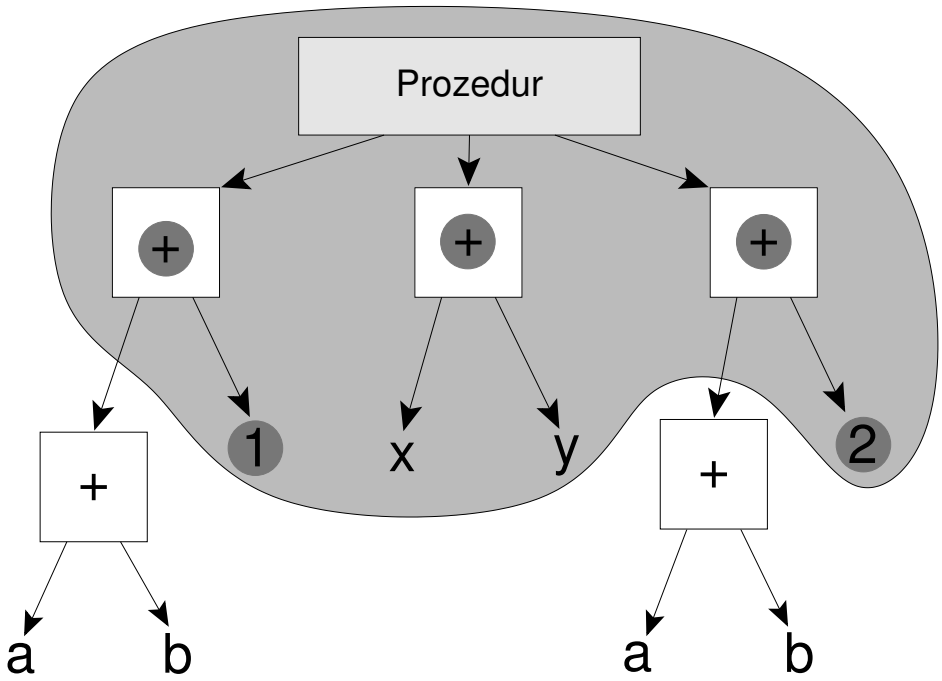

Abbildung 3.5: Beispiel für die Berechnung des Abstandes. Der hervorgehobene Teil wird untersucht.

Im passiven Modus speichert die Policy das Ergebnis als DistanceCountPolicy in der internen Lua Tabelle ab. Im aktiven Modus prüft die Policy, ob der maximal erlaubte Abstand `maxDist` überschritten wurde. Ist das der Fall, so wird an diesen Stellen die Lebenszeit der CSE-Variablen explizit aufgeteilt, und eine neue CSE-Variable wird eingeführt.

Es kann vorkommen, daß ein einziger Lauf der DistancePolicy nicht ausreicht, um korrekte Ergebnisse zu liefern. Das ist dann der Fall, wenn eine CSE eine andere CSE enthält und erstere durch das Aufsplitten der Lebenszeit nur noch einmal vorkommt und dann von der Copy Propagation eliminiert wird. Das folgende Beispiel illustriert diesen Fall, es wird ein maximaler Abstand von 100 Instruktionen gewählt.

```
cse29=1280-y;
cse39=cse29+x;
a=cse39*3;
 // Code, Abstand: 336 Instruktionen
b=cse39*d;
```

wird zu:

```
cse29=1280-y;
cse39=cse29+x;
a=cse39*3;
 // Code, Abstand: 336 Instruktionen
cse39_split_1=cse29+x;
b=cse3_split_19*d;
```

Nun ist der CSE Durchlauf fertig und die Abstände von cse39 sind gültig (maxDist wird nicht überschritten). Da die neu eingefügten lokalen Variablen cse39 und cse39_split_1 lediglich einmal benutzt werden, werden diese durch die nachfolgende Copy Propagation entfernt. Damit ergibt sich der folgende Code:

```
cse29=1280-y;
a=(cse29+x)*3;
 // Code, Abstand: 336 Instruktionen
b=(cse29+x)*d;
```

Hier ist der maximal erlaubte Abstand überschritten, da cse29 nun an den Stellen eingesetzt wird, wo vorher cse39 stand. Es ist also ein weiterer Lauf der DistancePolicy erforderlich, um die Abstände von cse29 zu korrigieren. Fälle wie dieser werden von der Implementierung der Policy korrekt erkannt und gehandhabt.

Parameter: Es werden folgende Parameter unterstützt:

- `MaxDist`: Der maximale erlaubte Abstand. Wird dieser überschritten, so erfolgt eine Auftrennung der Lebensdauer wie oben beschrieben.
- `DoOnlyCompleteCSE`: Ist diese Option aktiviert, so wird der Mittelwert aller Nutzungen berechnet und die ganze CSE entweder eliminiert oder ignoriert. Ist diese Option nicht eingeschaltet, so wird jede Nutzung einzeln untersucht.

InstrEvalPolicy

Die InstrEvalPolicy berechnet die Kosten der gefundenen CSE Ausdrücke. Dazu wird zunächst eine Kostentabelle für die jeweilige Plattform per Benchmark auf der konkreten Hardware mit dem jeweiligen Compiler bestimmt. Die Kostentabelle unterscheidet dabei zwischen SUIF-Operationen mit zwei Variablen und solchen mit Variablen und Konstanten. Es wurden Kosten für die folgenden Operationen bestimmt: add (`+`), sub (`-`), mul (`*`), div (`/`), rem (`%`), and (`&`), or (`|`), not-equal (`!=`), less-than (`<`), less-or-equal (`<=`), logical-and (`&&`), logical-or (`||`).

Die hier angegebenen Operationen bilden die Teilmenge der wichtigsten von der SUIF-Zwischendarstellung zur Verfügung gestellten Operationen. SUIF-Operationen, für die

momentan keine per Benchmark ermittelten Kosten bestimmt wurden, können bei Bedarf leicht hinzugefügt werden und sind für die in Kapitel 4 vorgestellten Ergebnisse ohne Bedeutung.

Eine Unterscheidung zwischen zwei Variablen und einer Variablen mit Konstanten ist notwendig, da ein Compiler bei einer Variablen und einer Konstanten mehr Möglichkeiten bei der Optimierung hat. Der Ausdruck `y=x%3` wird zum Beispiel auf dem Intel ICC in eine Multiplikation mit sehr großer Konstanten, eine Shiftoperation und eine Addition übersetzt. Der Compiler nutzt hier die Tatsache, daß der Intel Pentium Prozessor eine Multiplikation anbietet, die bei einem Überlauf die Berechnung auf zwei Ergebnisregister verteilt.

Der Benchmark zur Bestimmung der Kosten der einzelnen Ausdrücke unterscheidet sich leicht im Hinblick auf teils konstante oder variable Argumente. Für eine Variable mit einer Konstanten hat er das folgende Aussehen:

```
void confuse(int *a)
{
 if(a==42) { printf("Not reached"); }
}

int bench()
{
  for(j=0;j<RUNS;j++) {
    for(i=1;i<LOOPS;i++) {
      res = i OP CONST;
      confuse(&res);
    }
  }
  return res;
}
```

Für zwei Variablen:

```
void confuse(int *a)
{
 if(a==42) { printf("Not reached"); }
}

int bench()
{
  a = random();
  for(j=0;j<RUNS;j++) {
    for(i=1;i<LOOPS;i++) {
      res = a OP i;
```

```
        confuse(&res);
      }
    }
    return res;
}
```

Der jeweils untersuchte Operator OP wird innerhalb zweier geschachtelter Schleifen ausgeführt. Die innere Schleife inkrementiert die Variable i und stellt so der Operation OP ein variables Argument zur Verfügung. Abhängig von der zu messenden Situation wird entweder die Konstante CONST oder die Variable a, welche zufällig definiert wird, als weiteres Argument für OP benutzt.

Die i-Schleife ist in einer weiteren Schleife geschachtelt, die die Gesamt-Anzahl an Ausführungen von OP bestimmt, so daß bei der Ausführung und Zeitmessung dieser Benchmark-Programme auf der verfügbaren Hardware auch meßbare Laufzeiten auftreten.

Der Benchmark selbst ist in der Prozedur bench() untergebracht, der dann mit den in Kapitel 2.3.2 vorgestellten Meßmethoden gemessen wird.

Die jeweils untersuchte Operation wird also in einer Schleife ausgeführt und die Laufzeit gemessen. Die Funktion confuse() ist sehr wichtig, um zu verhindern, daß der Compiler Optimierungen in dieser Schleife durchführt, wie zum Beispiel die Berechnung aus der Schleife entfernen. Durch die Übergabe der Adresse von res an confuse() und den anschließenden if() Test mit diesem Pointer (der nie wahr sein kann) wird verhindert, daß der Compiler die Schleifen optimiert.

Die während der Programmausführung gemessenen Laufzeiten werden nun in eine Tabelle eingetragen. Anschließend wird eine abstrakte Kosteneinheit eingeführt. Diese ist die relative Laufzeit zu einer Addition mit Konstanten. Diese Addition wird mit zehn Kosteneinheiten angesetzt und alle anderen Operationen relativ dazu berechnet. Wenn also eine Addition eine Laufzeit von 20 Sekunden im Benchmark hat und eine Division eine Laufzeit von 100 Sekunden, so ergibt das für die Division die abstrakten Kosten von 50 Einheiten. Die gefundenen Kosteneinheiten sind sehr verschieden. Eine Division einer Variablen kostet zum Beispiel auf der Thumb/gcc Plattform 73 Kosteneinheiten, auf der Intel/icc dagegen nur 13 (da die oben erwähnte Optimierung zum Einsatz kommt).

Die gefundenen CSEs werden nun rekursiv untersucht, und es werden ihnen entsprechende Kosten zugewiesen. Betrachten wir den Ausdruck `cse45=(x-1)%3`. Der Ausdruck entspricht den Kosten einer Subtraktion von einer Variable mit einer Konstanten und einer Modulo Operation mit einer Konstanten. Mit den Kosten einer Division von 30 Einheiten und einer Subtraktion von 10 Einheiten ergeben sich als Gesamtkosten 40 Einheiten.

Parameter: Die Policy trägt die gefundenen Kosten als ExprEvalPolicy in die Lua-Tabelle ein. Zusätzlich kennt sie die folgenden Parameter:

- `CostTableFileName`: Der Name der Datei, die die Kosten der einzelnen Instruktionen enthält

- `minCost`: Gibt die minimalen Kosten an, die ein Ausdruck kosten muß, damit MaxScore zurück geliefert wird
- `showWhatOpcodes`: Debugoption, zeigt welche Opcodes im Ausdruck vorkommen
- `showHowManyUses`: Debugoption, zeigt die Zahl der Nutzungen des Ausdrucks an
- `showHowManySrcs`: Debugoption, zeigt die Zahl der Operanden des Ausdrucks an
- `considerUseCount`: Die Kosten werden mit der Zahl der Nutzungen multipliziert
- `considerNumOfOperands`: Die Zahl der Operanden wird als Kosten in den Ausdruck aufgenommen
- `considerSubCSEs`: Die Kosten für die CSEs, die in dem Ausdruck enthalten sind, werden als Kosten zum Ausdruck hinzuaddiert

TestsInIfPolicy

Die TestsInIfPolicy betrachtet die Kosten der gefundenen CSE Ausdrücke. Sie verhält sich exakt wie die EvalInstrPolicy, vergibt aber „Strafpunkte“ für Test-Ausdrücke, die in `if()` Konstruktionen vorkommen. Dies ist notwendig, da es auf den meisten Architekturen aufwendiger ist, `cse=x<2;if(cse)` zu berechnen als `if(x<2)`.

Aufwand durch Tests in if-Ausdrücken

Neben den Kosten eines Ausdrucks muß auch der Kontext berücksichtigt werden. Ein Ausdruck der Form: `if(expr)`, wobei `expr` ein Vergleich $<, <=, >, >=, ==, != $ mit der Konstanten *const* und der Variablen x ist, wird durch den GCC auf einem Intel Prozessor kompiliert zu:

```
cmpl    $const, %x    ; Vergleich
jle     .L3           ; Sprung je nach flags
```

Demgegenüber wird aus:

```
int a = expr;
if(a) {..};
```

unter den gleichen obigen Bedingungen:

```
cmpl    $const, %eax  ; Vergleich, x ist im Register %eax
setle   %al           ; Register entsprechend flags setzen
movzbl  %al, %eax     ; und in einem allgemeinen
                      ; Register speichern
testl   %eax, %eax    ; Logisches UND, Ergebnis im
                      ; flag Register
jne     .L4           ; Sprung
```

Der Overhead besteht darin, nach dem Vergleich den interessanten Teil des Flagregisters in ein allgemeines Register zu laden. Später muß dann dieses allgemeine Register wieder ins Flagregister geladen werden, da keine gängige Architektur einen Sprung entsprechend eines Registers erlaubt. Das passiert im obigen Beispiel mittels der `test` Assembleranweisung.

Damit kostet jedes nachfolgende `if(a)` zusätzlich die Assembleranweisung, um das Register ins Flag-Register zu laden. Einfache Ausdrücke wie

```
cse0=x<2;
if(cse0) {..};
if(cse0) {..};
```

dürften somit auch für beliebig viele `if(cse0) {..}` Ausdrücke keine Verbesserung bringen. Für komplexe Ausdrücke wie `cse0=(x<2 && y < 17 || ...)` kann die CSE Eliminierung Vorteile bieten, da die Neuberechnung des Wertes aufwendig sein kann.

Generell ist eine Aussage schwierig, da die Programmiersprache C die Berechnung einer Bedingung in if() Ausdrücken abbricht, sobald ein Term in einem `&&` Konstrukt den Wert `false` liefert (lazy evaluation). Umgekehrt bricht C auch bei `true` in `||` Konstrukten ab.

Der Overhead für die verschiedenen Architekturen wurde mit einem speziellen Benchmark bestimmt, der zuerst eine normale `if()` Anweisung in einer Schleife ausführt und danach die entsprechende CSE Version.

Parameter: Diese Policy schreibt die Variable TestsInIfPolicy=1 in die Lua-Tabelle, wenn die CSE in einem Testausdruck gefunden wurde. Zusätzlich werden folgende Parameter unterstützt:

- `PenalityPercent`: Die angegebene Prozentzahl wird von den Kosten eines Ausdrucks in einem `if()` Konstrukt abgezogen.
- `verbose`: Gibt zusätzliche Debuginformationen aus.

LifetimePolicy

Diese Policy untersucht, wieviele Variablen zu jedem Zeitpunkt, in dem ein Vorkommen einer CSE gefunden wurde, aktiv sind. Damit ist eine grobe Abschätzung über den derzeitigen Registerdruck möglich. So können Analysen durchgeführt werden, bei denen z. B. eine Eliminierung immer durchgeführt wird (unabhängig von den Kosten), wenn eine bestimmte Zahl an aktiven Variablen unterschritten wird. Umgekehrt kann bei Überschreiten einer Maximalgrenze ganz von der Eliminierung abgesehen werden, da dann angenommen wird, daß der Registerdruck so hoch ist, daß in jedem Fall in den Speicher ausgelagert werden muß. Es ist zu beachten, daß diese Policy nur grobe Abschätzungen erlaubt, da der Compiler viele Objekte in Registern unterbringen kann. Dazu gehören Konstanten, sofern sie nicht mit ins Befehlswort gepackt werden können,

Zwischenergebnisse, lokale Variablen, globale Variablen und zahlreiche andere Objekte, die zum Teil auf der hohen Abstraktionsebene, auf der die plattformabhängige High-Level CSE Optimierung arbeitet, nicht sichtbar sind.

Arbeitsweise: Die Lebenszeitanalyse arbeitet wie folgt:

1. Finde alle lokalen Variablen
2. Markiere Start und Ende der Lebensdauer der gefundenen Variable (erste und letzte Nutzung)
3. Für jede CSE Nutzung:
   a) prüfe, wie viele Variablen gerade lebendig sind. Dazu wird für jede lokale Variable geprüft, ob die Stelle der Nutzung im aktiven Bereich dieser Variablen liegt.

Es wird zusätzlich geprüft:

1. Ob nach einer Definition irgendwann eine weitere folgt, ohne daß die Variable benutzt wurde. Dies ist der Fall bei:

   ```
   a=18;
   b=12;
   a=b++;
   f(a);
   ```

   Hier beginnt die eigentliche Lebenszeit von a erst im Ausdruck `a=b++`.

2. Es wird geprüft, ob die zweite Definition dieselbe Variable erneut als Quelle enthält. Damit werden Fälle abgedeckt wie:

   ```
   a=12;  a=a+b;
   ```

   Hier beginnt die Lebenszeit schon mit dem `a=12` Ausdruck.

3. Ob die beiden Definitionen auf der gleichen Ebene liegen: Damit werden alle folgenden Fälle abgedeckt:

   ```
   a=12;
   if(b>7)  a=17;
   ```

   oder

   ```
   a=10;
   while(b<7)  a=b;
   ```

   Zwar wird a hier redefiniert, aber der Zweig wird nicht in jedem Fall durchlaufen.

Die Analyse kann noch verbessert werden, indem Fälle wie dieser untersucht werden:

```
a=12;
foo(a);
/* Code, der a nicht modifiziert */
a=b;
bar(a);
```

Die Lebenszeit von a ist zweigeteilt. Im ersten Teil wird a bis foo(a) benutzt und dann ab a=b. Die derzeitige Implementierung kann diesen Fall nicht unterscheiden, aber erkennen und den Benutzer warnen. Der Code ist leicht erweiterbar, um auch solche Fälle abzudecken. Alternativ könnte man die Variablen zu Anfang schon in eine SSA (Static single assignment) Form bringen, um das Problem zu umgehen.

In der SSA Form wird jeder Variable nur genau einmal ein Wert zugewiesen. Wird einer existierenden Variable zweimal ein Wert zugewiesen, so wird für die zweite Zuweisung eine neue Variable eingeführt, und alle folgenden Nutzungen werden mit der neuen Variable durchgeführt ([CFR+89]).

Um das obige Beispielfragment in eine SSA Form zu bringen, müssen lediglich das zweite `a` und alle folgenden Nutzungen von `a` umbenannt werden:

```
a=12;
foo(a);
/* Code, der a nicht modifiziert */
a_1=b;
bar(a_1);
```

Parameter: Diese Policy trägt die Zahl der aktiven Variablen unter dem Namen LifetimePolicy in die Lua-Tabelle ein. Zusätzlich werden folgende Parameter unterstützt:

- `MaxLifeVars`: Maximale Zahl aktiver Variablen. Wird der Wert überschritten, wird MinScore zurück geliefert
- `dryRun`: Nur Informationen ausgeben, keine Entscheidung treffen
- `verbose`: Mehr Informationen zu den einzelnen CSEs ausgeben
- `GoodScore`: Welcher Wert bei Unterschreiten der MaxLifeVars Grenze zurück gegeben werden soll

3.4 Feedback Algorithmus

Um das Potential einer CSE Eliminierung zu untersuchen, die nur mit einer Auswahl der gefundenen CSEs arbeitet statt mit allen CSEs, wurde ein Algorithmus entworfen, der die maximal mögliche Laufzeitverbesserung herausfinden soll. Das Prinzip des Algorithmus ist es, jeweils alle CSEs im Quellcode einzeln zu eliminieren und die Laufzeitänderungen zu messen. Danach wird die beste gefundene CSE permanent eliminiert, und für die

verbleibenden CSEs wird wieder die jeweilige Laufzeit gemessen. Das wird solange wiederholt, bis keine Laufzeitverbesserungen mehr gefunden werden. Pseudocode für diesen Algorithmus findet sich in Kapitel 1.4.

Um den Algorithmus im bestehenden Framework zu realisieren, ist eine neue Policy nötig, mit der gezielt einzelne CSEs eliminiert oder ignoriert werden können. Diese AlwaysIncludeConfFilePolicy erlaubt es, anhand einer Tabelle anzugeben, welche CSE eliminiert werden soll und welche nicht. Es wird dabei immer die gesamte CSE betrachtet, nie die einzelnen Vorkommen.

Der Feedback-Algorithmus hat den Nachteil, daß er den untersuchten Quellcode auf der realen Hardware mit dem realen Compiler kompilieren und ausführen muß. Für das Ausführen sind zudem repräsentative Testdaten nötig. Da zahlreiche Läufe nötig sind, dauert die Ausführung des Feedback Algorithmus u.U. sehr lange. Im schlimmsten Fall müssen:

$$n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2} \tag{3.1}$$

Testprogramme erzeugt und ausgeführt werden. Dieser Fall tritt dann ein, wenn jede CSE zu Laufzeitverbesserungen führt. Aus den genannten Gründen ist der Algorithmus in erster Linie für Forschungszwecke geeignet. Im Rahmen der vorliegenden Arbeit liefert er interessante Aufschlüsse darüber, welches Optimierungspotential noch möglich ist.

3.4.1 AlwaysIncludeConfFilePolicy

Diese Policy wird für den Feedbackalgorithmus benötigt und soll im folgenden kurz beschrieben werden. Mit ihr können CSEs in einer Konfigurationsdatei benannt werden, die positiv bewertet werden sollen. Alle CSEs, die nicht in der Datei vorkommen, werden mit `MinScore` bewertet.

Parameter: Folgende Parameter werden unterstützt:

- `filename`: Name der Datei, in der die CSEs aufgelistet werden
- `PartialMatch`: Wird der Name nur teilweise gefunden, wird dieser Name trotzdem als Treffer betrachtet. Dies ist nützlich im Zusammenhang mit der DistancePolicy, die Namen wie cse0_split_1 erzeugt
- `IncludeScore`: Die Punktzahl für jede CSE, die in der Konfigurationsdatei gefunden wurde

Die Policy erlaubt es, gezielt Experimente mit einzelnen CSEs zu machen.

3.5 Entscheidungsmechanismus mit Neuronalen Netzen

Neben dem Standard-Entscheidungsmodul der PolicyEngine, das auf der Addition der Bewertungen der einzelnen Policies basiert, wurde ein alternatives Entscheidungsmodul

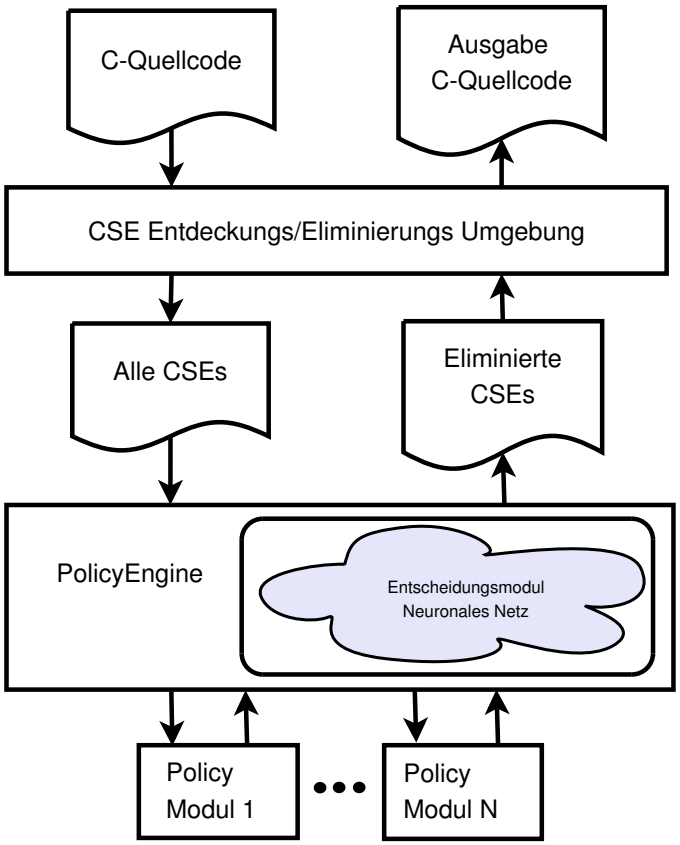

Abbildung 3.6: Implementierung mit neuronalem Netz

auf der Basis eines neuronalen Netzes entwickelt. So werden die Parameter der gefundenen CSE als Ganzes von einem adaptiven Mechanismus untersucht. Ein Netz muß nur einmal auf einer Plattform trainiert werden und kann dann für die Klassifikation der gefundenen CSEs eingesetzt werden. Dabei werden wichtige Kenndaten, wie z. B. ab welchen Kosten eine Eliminierung sinnvoll ist oder welchen Einfluß die Zahl der lebendigen Variablen auf die Laufzeit hat, selbständig aus den Daten gelernt, die die Policies über die CSEs sammeln.

Eine Realisierung durch ein Netz ist möglich, da das Entscheidungsmodul durch ein Plugin verändert werden kann. Damit können alternative Implementierungen realisiert werden, die über alle konkreten Informationen der Policies verfügen.

Die Abbildung 3.6 veranschaulicht die neue Struktur. Das neue Entscheidungsmodul wird in Lua implementiert. Für das neuronale Netz selbst wird die Bibliothek fann (Fast Artificial Neural Network Library)[1] genutzt, die aus Performancegründen in C realisiert ist.

Für das neue Entscheidungsmodul werden die Policies nur noch passiv genutzt, d.h.

[1] `http://fann.sf.net`

ihre abstrakten Bewertungen werden ignoriert. Nur die konkreten Informationen (wie z. B. Kosten, Abstand) werden verwendet. Dabei wird das neue Entscheidungsmodul sowohl für die gesamte CSE als auch für jede einzelne Nutzung aufgerufen.

Eine Entscheidung über die Eliminierung wird nun im Netz selbst gefällt, es werden die folgenden Parameter für die Entscheidung berücksichtigt: Zahl der Vorkommen, Kosten (dabei wird der Abzug durch Tests in if() Ausdrücken mit einberechnet) und Zahl der lebendigen Variablen. Diese Parameter werden dem Netz als Eingabe präsentiert. Die Ausgabe des Netzes ist Grundlage für die Entscheidung, ob die untersuchte CSE eliminiert wird. Die Abstandsdaten werden hier nicht präsentiert, da das Spliten von CSEs dazu führt, daß sich Kenndaten wie die Zahl der Vorkommen ändern und so neu bewertet werden müssten.

Erzeugung der Trainingsdaten

Wie in Kapitel 2.5 erläutert, ist die Erzeugung von Trainingsdaten kritisch bei der Anwendung von neuronalen Netzen. Sie müssen aussagekräftig sein, dürfen aber gleichzeitig nicht das Ergebnis vorwegnehmen. Trainingsdaten, die sich aus den Feedbackläufen ergeben, verbieten sich daher, da das Netz sonst gezielt die Feedbackergebnisse lernen und reproduzieren würde. Eine Methode, die auf einem Zufallselement basiert, ist sinnvoller. Dazu wurde eine neue Policy entwickelt, die die gefundene CSE nur mit einer bestimmten Wahrscheinlichkeit eliminiert. Diese Policy wird weiter unten im Detail beschrieben.

Trainingsdatenerzeugung Zum Erzeugen der Trainingsdaten wurde der folgende Algorithmus eingesetzt:

1. Erzeuge eine zufällige Menge M von eliminierten CSEs (Wahrscheinlichkeit pro CSE=0.5)
   a) Wähle eine CSE C aus M zufällig. Entferne jede Nutzung N einmal und messe die Performance mit und ohne N. Notiere die relative Performanceänderung zusammen mit den Kenndaten der CSE in einer Datei (Zahl-der-Nutzungen, Kosten, Life-Vars, Abstand, Test-In-If)
   b) Wiederhole a) 10 mal
2. Wiederhole 1) und 2) 10 mal

Es werden alle CSE-Parameter erfaßt (Zahl der Nutzungen, Kosten, Life-Vars, Abstand, Test-In-If). Zusätzlich wird die Laufzeitveränderung ermittelt, die sich ergibt, wenn eine CSE mit diesen Daten eliminiert oder ignoriert wird. Dieser Benchmark wird für jede Plattform einmal durchgeführt. Mit den so gewonnenen Daten wird ein neuronales Netz pro Plattform trainiert.

Es zeigt sich, daß die sinnvollsten Eingaben für das Netz das Tupel (Zahl der Nutzungen, Kosten, Lebende Variablen) ist. Das Tests-in-If Kriterium geht bereits in das Kosten Kriterium ein und ist somit redundant. Der Abstand hilft dem Netz wenig, da

keine Möglichkeit besteht, die CSEs mit großem Abstand aufzuspalten, ohne im Allgemeinen Nutzungen zu löschen oder eine zusätzliche Variable einzuführen, was wiederum die Zahl der lebendigen Variablen verändern würde.

Durch die Verwendung eines neuronalen Netzes müssen bis auf die Kosten für Ausdrücke keine plattformabhängigen Parameter angegeben werden. Da das Netz auf realer Hardware mit dem konkreten Compiler trainiert wird, lernt es plattformabhängige Eigenheiten, wie die Auswirkungen des Registerdrucks, selbständig.

Im folgenden werden die beiden Methoden beschrieben, mit denen die neuronalen Netze trainiert werden.

Methode 1 Die Netze werden mit den Parametern (Zahl-der-Nutzungen, Kosten, LifeVars) trainiert. Das gewünschte Ergebnis ist die Laufzeitveränderung, die sich durch diese CSE ergibt. Wenn also eine CSE mit den Kenndaten (Vorkommen=8, Kosten=140, LifeVars=17) die Laufzeit nach Eliminierung auf 97,7% bringt, so soll das Netz nach dem Training eine passende Vorhersage für diese und für ähnliche CSEs machen. Eine Performance kleiner hundert Prozent zeigt dabei an, daß die Laufzeit mit dieser CSE besser geworden ist. Eine Zahl größer hundert Prozent zeigt, daß die Eliminierung dieser CSE die Laufzeit verschlechtert hat.

Aufgabe des neuronalen Netzes ist es, für unbekannte CSEs anhand ihrer Parameter eine Vorhersage über die zu erwartende Performanceänderung zu machen. Das Netz liefert für unbekannte CSEs einen Wert zwischen $[-1, +1]$ zurück. Dieser Wert wird linear zurück auf eine zu erwartende Laufzeitänderung skaliert. Ist die Laufzeitänderung kleiner hundert Prozent, so ist die Vorhersage des Netzes positiv, bei einem Wert größer hundert ist sie negativ. Das Entscheidungsmodul kann nun mit der gewünschten Laufzeitveränderung parametrisiert werden. Es kann z. B. festgelegt werden, daß nur CSEs eliminiert werden, die eine Laufzeit von hundert Prozent oder besser ergeben.

Als Aktivierungsfunktion kommt der $\tanh(x)$ zum Einsatz, die Performanceänderung wird linear auf das Intervall $[-1, +1]$ skaliert.

Methode 2 Die Auswahl der Trainingsdaten erfolgte auf die gleiche Weise wie bei Methode 1, nur die Auswertung wurde verändert. Statt das Netz mit der relativen Performanceänderung zu trainieren, wird es mit einer Ja/Nein Entscheidung als Ergebnis trainiert. Jede CSE in der Trainingsmenge bekommt den Wert Null zugewiesen, wenn die Performance sich verbessert und den Wert Eins, wenn die Performance sich verschlechtert. Bei der Klassifizierung unbekannter CSEs liefert das Netz eine Ausgabe zwischen $[0, 1]$. Je näher sie an dem Wert Eins liegt, desto besser wird die gefundene CSE eingeschätzt, je näher an Null, desto schlechter. Diese Zahl stellt also eine Vorhersage über die Güte der gefundenen CSE dar. Für eine CSE mit den Kenndaten (Vorkommen=8, Kosten=40, LifeVars=17) liefert das Netz auf der Intel/icc Plattform zum Beispiel den Wert $0,85$. Dies ist ein starker Hinweis, daß diese CSE zu Laufzeitverbesserungen führt.

Das Netzwerk wurde so verändert, daß als Aktivierungsfunktion die logistische Funktion $\frac{1}{1+e^{-x}}$ zum Einsatz kommt, da dieses Netz in einem Ergebnisbereich im Intervall $[0, 1]$ arbeitet.

Die Netze selbst unterscheiden sich je nach Plattform in der Zahl ihrer Neuronen. Generell wurden Netze mit drei Schichten bei einem vollständig verknüpften Netzwerk verwendet. Dabei befinden sich drei Neuronen in der Eingabeschicht (pro Parameter einer), 15–35 Neuronen in der verdeckten Schicht und ein Neuron in der Ausgabeschicht. Beim Training wurde mit einer Eingabemenge von etwa 200–300 Trainingsdaten gearbeitet. Es wurden 10 000–50 000 Trainingsiterationen mit dem Back-Propagation Lernverfahren bei einer Lernrate von $0,1$ durchgeführt.

3.5.1 RandomPolicy

Diese Policy entscheidet per Zufall, ob eine CSE eliminiert wird. Sie wird für das Training des neuronalen Netzes benötigt. Die Wahrscheinlichkeit für eine Eliminierung wird per Parameter bestimmt. Es können sowohl komplette CSEs mit allen Vorkommen zufällig eliminiert werden, als auch einzelne Vorkommen.

Mittels dieser Policy wurde eine Testumgebung implementiert, die mit einer zufälligen Verteilung von CSEs Sourcecodes erzeugt und ablaufen läßt. Damit kann automatisch für eine große Zahl von verschiedenen erzeugten Sourcen festgestellt werden, ob es zu Abweichungen der Ausgabe gegenüber der unmodifizierten Version kommt. Auch kann so überprüft werden, ob es zu Programmfehlern oder Abstürzen seitens der CSE Implementierung kommt.

Parameter:

- `PropCompleteCSE`: Wahrscheinlichkeit, daß eine komplette CSE eliminiert wird
- `DoSingleCSE`: Wenn die gesamte CSE nicht eliminiert wird, kann hiermit aktiviert werden, daß einzelne Nutzungen zufällig eliminiert werden
- `PropSingleUse`: Wahrscheinlichkeit, daß eine einzelne CSE Benutzung eliminiert wird

4 Ergebnisse

In diesem Kapitel werden die Meßergebnisse mit der plattformabhängigen High-Level CSE Optimierung für den CAVITY Benchmark vorgestellt.

Der CAVITY Benchmark ist ein Benchmark aus der medizinischen Bildbearbeitung und wird in [BTC89] näher beschrieben. Er eignet sich sehr gut für die Analyse der plattformabhängigen High-Level CSE Optimierung, da er über zahlreiche CSEs verfügt, die von GCC und ICC nicht identifiziert werden.

4.1 Einzelne CSEs

Zunächst wurde überprüft, wie sehr einzelne CSEs die Laufzeit des Gesamtprogramms beeinflussen. Es liegt nahe, daß nicht jede CSE Eliminierung gleich zur Performanceänderung beiträgt. Das Ziel dieser Untersuchung ist eine Qualifizierung, welche von diesen Verbesserungen, keine Änderungen oder Verschlechterungen bringen.

Dazu wurden pro gefundener CSE zwei Quellcodes erzeugt. Im ersten Quellcode wird nur die jeweils betrachtete CSE mit allen ihren Vorkommen eliminiert. Damit kann direkt der Einfluß dieser CSE auf die Laufzeitverbesserung nachvollzogen werden. Wird eine solche erreicht, so hat die CSE positive Auswirkungen auf die Laufzeit.

Im zweiten Quellcode werden alle gefundenen CSEs eliminiert, nur die jeweils betrachtete CSE wird ignoriert. Im Quellcode sind also bei N gefundenen CSEs $N - 1$ Eliminierungen durchgeführt worden. Damit wird überprüft, welche Auswirkungen es hat, wenn eine bestimmte CSE nicht eliminiert wird. Als Vergleich wird nun die Laufzeit gewählt, die sich ergibt, wenn man alle CSEs eliminiert. Wenn die Laufzeit des erzeugten Quellcodes besser ist als die bei Eliminierung aller CSEs, so ist die betrachtete CSE schädlich für die Laufzeit. Ist die Laufzeit dagegen schlechter, so wurde eine CSE ignoriert, die wichtig für Laufzeitverbesserungen ist.

4.1.1 Ergebnisse

Der Einfluß einer einzelnen CSE wurde auf die beiden oben beschriebenen Arten gemessen. Zuerst wurden alle CSEs ignoriert und nur genau eine CSE eliminiert. Das Ergebnis wird in Abbildung 4.1 für alle betrachteten Plattformen zusammengefasst.

Auf der X-Achse ist die relative Laufzeit aufgetragen, wobei 100% für die Laufzeit von CAVITY ohne jegliche eliminierte CSE steht. Auf der Y-Achse findet sich die Anzahl von CSEs, die bei Eliminierung zu dieser relativen Laufzeit führen.

Alle CSEs kleiner als 100% tragen dabei zu Laufzeitgewinnen bei, da das resultierende Programm jetzt schneller läuft als ohne die CSE-Variable. Es ist zu erkennen, daß

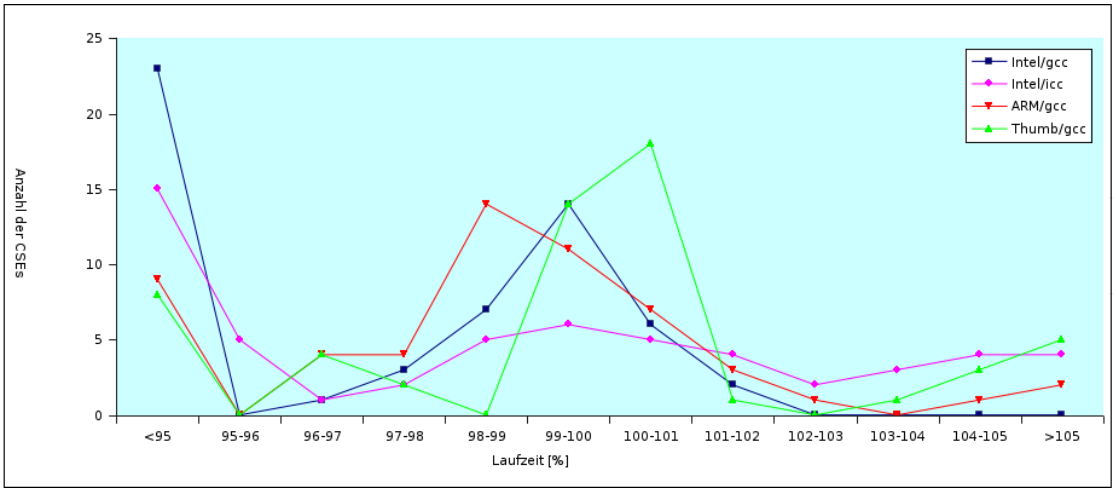

Abbildung 4.1: Nur eine einzelne CSE wird eliminiert

viele CSEs im Bereich 99%–101% liegen, und daß damit eine sehr geringe Veränderung durch diese CSEs zu erwarten ist. Es zeigt sich auch, daß das Eliminieren von bestimmten CSEs Laufzeitgewinne von $> 5\%$ bringt, andere dagegen die Laufzeit um $> 5\%$ verschlechtern. Dabei ist zu erkennen, daß es unabhängig von der Plattform wesentlich mehr CSEs gibt, die zu starken Laufzeitverbesserungen führen als solche, die zu starken Verschlechterungen führen.

Bei der zweiten Methode wurden alle gefundenen CSEs eliminiert und als Laufzeit von 100% betrachtet. Nur genau eine CSE wurde ignoriert. Das Ergebnis wird in Abbildung 4.2 dargestellt. Wenn die Laufzeit des resultierenden Programms besser ist als die Laufzeit bei Eliminierung aller CSEs, so hat die untersuchte CSE negative Auswirkungen. Wird die Laufzeit dagegen schlechter, so handelt es sich um eine CSE, die positiv zu den Laufzeitveränderungen beiträgt.

In der Abbildung 4.2 erkennt man, daß sich sehr viele Änderungen im Bereich von 99%–101% bewegen und damit auf eine eher geringe Rolle der CSE hindeuten. Sie zeigt aber auch, daß das Weglassen von nutzlosen CSEs zu Laufzeitverbesserungen führt. Beim Intel/gcc sind zwei CSEs vorhanden, deren Ignorieren jeweils zu einer Laufzeitverbesserung von 5% gegenüber dem High-Level CSE Ergebnis führt. Beim Intel/icc und Thumb/gcc ist es jeweils eine. Umgekehrt führt das Ignorieren von zwei guten CSEs beim Thumb/gcc zu je 5% Laufzeitverschlechterung. Dies ist auch bei den anderen Plattformen zu erkennen, wenn auch weniger deutlich.

In Anhang A.1.2 finden sich die genauen Zahlen zu den einzelnen CSEs. Alle gefundenen CSEs werden in Anhang A.1.1 aufgelistet.

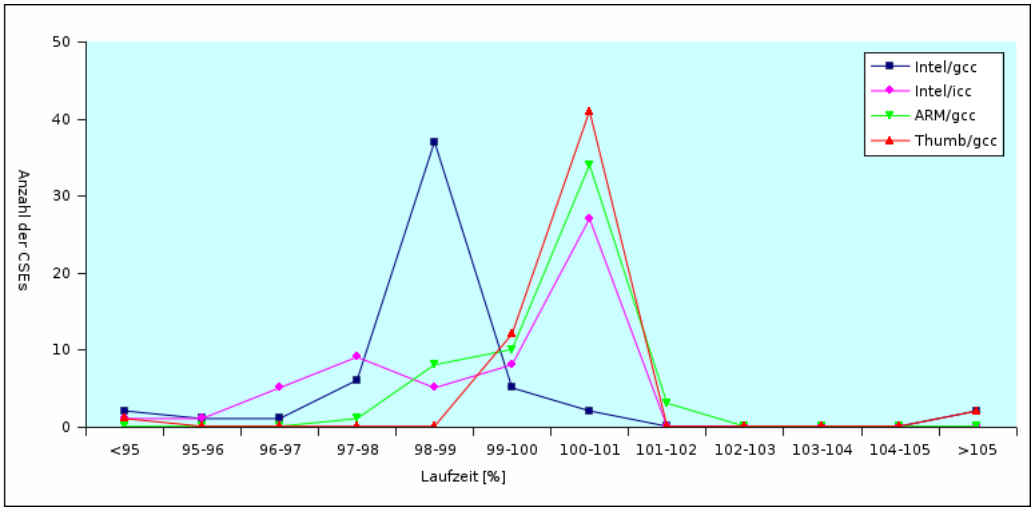

Abbildung 4.2: Alle CSEs bis auf eine werden eliminiert

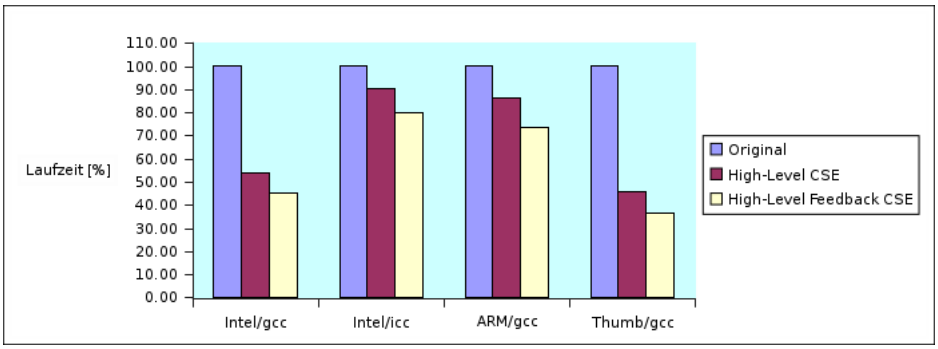

Abbildung 4.3: Zusammenfassung der Feedback Ergebnisse

4.2 Feedback

Nachdem sich gezeigt hat, daß jede CSE einen sehr unterschiedlichen Anteil an der Laufzeit hat, ist es sinnvoll zu untersuchen, welche Laufzeitverbesserung durch eine geeignete Auswahl von CSEs erreicht werden kann. Dazu wurde ein einfacher Feedback optimierender Algorithmus entwickelt, der in Kapitel 3.4 beschrieben wird.

Die Ergebnisse des Experiments für den CAVITY Benchmark sind in Abbildung 4.3 zusammengefaßt. Dabei finden sich auf der X-Achse die jeweils untersuchten Plattformen, auf der Y-Achse die Laufzeit relativ zu der Original-Version. Der erste Balken ist der unmodifizierte CAVITY Benchmark, der zweite Balken die Laufzeit nach der High-Level CSE Optimierung, und der dritte Balken die Laufzeit nach der Feedback-Optimierung.

Man erkennt, daß die High-Level CSE Optimierung sehr gute Ergebnisse erzielt. Diese werden durch den Feedback-Algorithmus noch einmal um 11,7%–20,7% gegenüber der

High-Level CSE Optimierung verbessert. Es bietet sich also noch interessantes Optimierungspotential.

4.2.1 Ergebnisse auf den jeweils untersuchten Plattformen

Im folgenden werden die Ergebnisse der Feedback-Optimierung auf den vier untersuchten Plattformen im Detail vorgestellt. Dabei werden zunächst die wichtigsten Ergebnisse jeweils in einer Tabelle zusammengefasst, dann werden die gefundenen CSEs aufgelistet. Aus ihnen ergeben sich Anhaltspunkte für eine analytische Bestimmung von sinnvollen und weniger sinnvollen CSEs.

In den Tabellen gibt die Spalte „Version“ die jeweilige Code Version des Benchmarks an, „Laufzeit“ gibt die tatsächliche Laufzeit in Sekunden (Intel) oder Maschinenzyklen (ARM) an. „CSEs“ ist die Zahl der insgesamt eingefügten CSE Variablen und „Uses“ die Zahl der Nutzungen der neuen CSE-Variablen. Die Spalte „Laufzeit in Prozent“ wird gegen die Original-Laufzeit berechnet.

Bei der Auflistung der gefundenen CSEs stellt die Zahl hinter dem „//“ die Zahl der Nutzungen dieser CSE-Variablen dar. Die Reihenfolge spiegelt die Laufzeitverbesserung wider, die erreicht wurde. Die CSE mit der größten Laufzeitverbesserung kommt zuerst.

Intel/gcc

Auf dieser Plattform ergab sich eine Laufzeitverbesserung von 15% gegenüber der High-Level CSE Optimierung.

Version	Laufzeit [s]	Anzahl CSEs	Anzahl Uses	Laufzeit [%]
Original	37.05	–	–	100.00
High-Level CSE	19.83	38	105	53.52
Feedback CSE	16.78	9	66	45.29

Tabelle 4.1: Feedback Ergebnisse für Intel/gcc

Es wurden die folgenden CSEs gefunden:

```
cse_053_72= cse_025_47 + 3 - cse_004_26 - cse_023_45;
// 3
cse_023_45= (x - 1) % 3;
// 14
cse_015_37= y % 3;
// 12
cse_012_34= (y - 2) % 3;
// 7
cse_004_26= x % 3;
// 12
cse_055_73= (3-cse_015_37-cse_012_34)*3+3-cse_004_26-cse_023_45;// 3
```

```
cse_025_47= cse_012_34 * 3;
// 6
cse_000_0= Gauss[1];
// 6
cse_050_70= cse_015_37 * 3 + 3 - cse_004_26 - cse_023_45;
// 3
```

Die CSEs cse_023_45, cse_015_37, cse_012_34, cse_004_26 sind relativ einfache Ausdrücke, die allerdings häufig benutzt werden und eine aufwendige Modulo Operation enthalten. Komplex, aber ohne aufwendige Operatoren sind cse_053_72, cse_055_73 und cse_050_70. Die Variable cse_025_47 ist kurz, wird aber häufig benutzt. Die Variable cse_000_0, deutet darauf hin, daß der GCC Compilers kein Scalar Replacement als Optimierung [Muc97] beherrscht.

Vergleicht man die gefundenen CSEs mit den Ergebnissen aus Kapitel 4.1, so zeigt sich, daß zum Beispiel die cse_053_72 bei alleiniger Eliminierung einen Laufzeitgewinn von 20,7% liefert. Allerdings sind die Ergebnisse nicht direkt vergleichbar, da diese CSE bei alleiniger Eliminierung keine weiteren CSEs enthält, die cse_053_72 im Feedback-Ergebnis aber schon. Daher sind direkte Vergleiche nicht möglich, es zeigt sich aber, daß die ersten sechs der gefundenen neun CSEs jeweils deutlich über 10% Laufzeitgewinn liefern, wenn sie alleine eliminiert werden.

Intel/icc

Auf dieser Plattform wurden Laufzeitgewinne von 12% gegenüber der High-Level CSE Version gemessen. Im Vergleich zum GCC sind sämtliche absoluten Laufzeiten des vom ICC generierten Codes deutlich niedriger. Dies zeigt die Stärken des ICC Compilers.

Version	Laufzeit [s]	Anzahl CSEs	Anzahl Uses	Laufzeit [%]
Original	8.82	–	–	100.00
High-Level CSE	7.95	38	105	90.14
Feedback CSE	7.02	4	11	79.59

Tabelle 4.2: Feedback Ergebnisse für Intel/icc

Es wurden die folgenden CSEs gefunden:

```
cse_055_73 = (3-y%3-(y-2)%3)*3+3-x%3-(x-1)% 3; // 3
cse_038_60 = y % 3 * 3 + (x - 1) % 3;          // 3
cse_031_53 = y % 3 * 3 + 3;                    // 3
cse_045_66 = (3-y%3-(y-2)%3)*3+x%3;            // 2
```

Der ICC zeigt ein ganz anderes Profil als der GCC. Hier werden nur komplexe CSEs, die relativ selten vorkommen, aber zahlreiche Operanden haben, gefunden. In diesem Fall spielen vermutlich die Eigenarten der Registerallokation eine Rolle. Wie man im Anhang A.1.2 erkennt, führt die Eliminierung von einfachen, aber häufigen Ausdrücken,

wie cse_023_45 oder cse_015_37 beim ICC zu deutlichen Laufzeitverschlechterungen, während diese CSEs für den GCC nützlich sind.

Die Intel/icc Ergebnisse sind besser mit den Ergebnissen aus Kapitel 4.1 vergleichbar, da hier keine CSE eine andere CSE enthält. Es zeigt sich, daß die gefundenen CSEs einzeln zwischen 2,0%–15,6% Laufzeitgewinn erzielen. Allerdings wird auch deutlich, daß eine direkte Vorhersage der Feedback-Ergebnisse mit den Ergebnissen aus Kapitel 4.1 nicht möglich ist. Die cse_053_72 liefert zum Beispiel einzeln einen Laufzeitgewinn von 12,2%, kommt aber im Feedback-Ergebnis nicht vor.

ARM/gcc

Auf dieser Plattform ergaben sich Laufzeitverbesserungen gegenüber der High-Level CSE Version von 15%.

Version	Laufzeit [cyc]	Anzahl CSEs	Anzahl Uses	Laufzeit [%]
Original	244.177	–	–	100
High-Level CSE	209.848	38	105	85,9
Feedback CSE	179.339	9	53	73.4

Tabelle 4.3: Feedback Ergebnisse für ARM/gcc

Es wurden die folgenden CSEs gefunden:

```
cse_023_45 = (x - 1) % 3;                                  // 18
cse_047_68 = (int)((unsigned int)y & 1u)*298+x-1;          // 3
cse_049_69 = (3-y%3-(y-2)%3)*3+cse_023_45;                 // 3
cse_004_26 = x % 3;                                        // 18
cse_024_46 = 1 <= x && x < 299;                            // 2 (in if)
cse_052_7 = (int)((unsigned int)y & 1u)*298+x-2+5120;      // 2
cse_001_1 = *Gauss;                                        // 3
cse_054_7 = cse_048_6 + 2560;                              // 2
cse_048_6 = 298-(int)((unsigned int)y&1u)*298+x-1;         // 2
```

Auch auf dem ARM findet sich die Klasse der einfachen, aber häufig verwendeten CSEs in Form von cse_023_45 und cse_004_26. Dagegen ist cse_049_69 sehr komplex, kommt aber nur dreimal vor. Die Variablen cse_047_68, cse_052_7 und cse_048_6 sind ebenfalls komplex. Überraschend ist cse_054_7, da sie nur zweimal vorkommt und nur eine Addition mit einer Variablen enthält. Es ist unklar, warum cse_001_1 enthalten ist, nicht aber cse_000_0, die häufiger vorkommt. Ebenfalls interessant ist, daß mit cse_024_46 ein komplexer Test-Ausdruck vorkommt.

Vergleicht man die Ergebnisse des ARM/gcc mit den Ergebnissen aus 4.1, so zeigt sich, daß auch hier jeweils CSEs ausgewählt wurden, die einzeln deutliche Gewinne gebracht haben. So erzielen die ersten drei CSEs einzeln 8,2%, 7,4% und 6,9% Laufzeitgewinne.

Thumb/gcc

Auf dieser Plattform ergaben sich Laufzeitverbesserungen gegenüber der High-Level CSE Version von 20%.

Version	Laufzeit [cyc]	Anzahl CSEs	Anzahl Uses	Laufzeit [%]
Original	671.166	–	–	100.0
High-Level CSE	306.428	38	105	45.66
Feedback CSE	243.030	6	63	36.21

Tabelle 4.4: Feedback Ergebnisse für Thumb/gcc

Es wurden die folgenden CSEs gefunden:

```
cse_023_45 = (x - 1) % 3;       // 20
cse_004_26 = x % 3;             // 18
cse_028_50 = y % 3 * 3;         // 8
cse_025_47 = (y - 2) % 3 * 3;   // 8
cse_001_1 = *Gauss;             // 3
cse_000_0 = Gauss[1];           // 6
```

Beim Thumb ist die Auswahl sehr einfach. Es wurden nur zwei Klassen von Ausdrücken ausgewählt. Einerseits die einfachen, häufig benutzten Modulo Ausdrücke wie cse_023_45, cse_004_26, cse_028_50 und cse_025_47. Andererseits cse_001_1 und cse_000_0, die Werte aus dem Gauss Feld zwischenspeichern und damit das Fehlen der „Scalar Replacement“ Optimierung beim GCC ausgleichen.

Beim Thumb/gcc zeigt sich erneut, daß ein Vergleich mit den Ergebnissen aus Kapitel 4.1 schwierig ist. Zwar liefern cse_023_45 und cse_004_26 Laufzeitgewinne von 30,3% respektive 22,4%, aber cse_028_50 und cse_025_47 liefern leichte Laufzeitverluste. Allerdings enthalten die beiden letzteren CSEs in der Messung aus Kapitel 4.1 jeweils eine weitere CSE, die im Feedback Ergebnis nicht eliminiert wurde. Das erklärt die sehr unterschiedlichen Ergebnisse.

Laufzeit relativ zu den High-Level CSE Ergebnissen

Zur besseren Vergleichbarkeit mit den folgenden Messungen, die alle mit der High-Level CSE als Referenz (100%) rechnen, zeigen die Abbildung 1.5 aus der Einleitung und die folgende Tabelle die Feedback Ergebnisse mit der High-Level CSE Laufzeit als 100% Referenz.

Plattform	Laufzeit (zu High-Level CSE)
Intel/gcc	85.91%
Intel/icc	88.30%
ARM/gcc	85.46%
Thumb/gcc	79.31%

Tabelle 4.5: Laufzeiten relativ zu den High-Level CSE Ergebnissen

Zusammenfassung der Ergebnisse

Die Ergebnisse machen deutlich, daß Laufzeitgewinne von 11,7%–20,7% gegenüber der High-Level CSE Optimierung durch gute Auswahl der CSEs möglich sind. Im CAVITY Benchmark befinden sich alle CSEs in der inneren Schleife und werden etwa gleich oft ausgeführt. Es sind also Faktoren des Ausdrucks selbst, die den Unterschied in der Laufzeitänderung machen. Die gefundenen CSEs liefern Anhaltspunkte dafür, welche CSEs die größten Laufzeitgewinne bringen. Zahl der Vorkommen, Komplexität und verwendete Operationen haben einen Einfluß auf die Laufzeit und lassen sich direkt aus der CSE selbst ableiten. Andere Faktoren wie der Abstand und die Zahl der lebendigen Variablen, die in Kapitel 3 vorgestellt wurden, werden von dem Feedback-Algorithmus nicht berücksichtigt. Es ist auffällig, daß nur vier bis neun CSEs gefunden werden, die für die Laufzeitgewinne verantwortlich sind.

4.3 Ergebnisse der einzelnen Policies

In diesem Abschnitt wird untersucht, welchen Einfluß die einzelnen Faktoren haben, die von den Policies, die in Kapitel 3 vorgestellt wurden, entdeckt werden.

Bei den Untersuchungen traf die PolicyEngine ihre Entscheidungen über eine Eliminierung auf der Basis des Standardentscheidungsmoduls. Damit sind einzelne Policies oder einfache Verknüpfungen von Policies möglich.

Für kompliziertere Gewichtungen der einzelnen Policyergebnisse wird später ein Entscheidungsmodul auf der Basis eines neuronalen Netzes eingesetzt.

4.3.1 Anzahl der identischen Ausdrücke

Die UseCountPolicy (vgl. Kapitel 3.3.2) eliminiert nur Ausdrücke, wenn sie eine bestimmte Zahl an Vorkommen im Sourcecode überschreiten und damit genau diese Anzahl an Berechnungen eingespart wird.

Die Abbildung 4.4 zeigt die Ergebnisse im Überblick. Dabei stellt die X-Achse die minimal nötige Zahl an eingesparten Berechnungen dar. Die „cse“ Spalte, die dort zu finden ist, repräsentiert die normale CSE Eliminierung, d.h. eine Berechnung muß mindestens zweimal vorkommen. Die CSEs im CAVITY Benchmark kommen 2,3,6,8 und 9 mal vor. Daher sind die Spalten 4–6 und 7–8 zusammengefaßt. Wird die aufgetragene Mindestzahl nicht erreicht, so wird die gefundene CSE ignoriert. Auf der Y-Achse findet sich die damit erreichte Laufzeit.

Bei mindestens drei eingesparten Berechnungen erreichen alle Plattformen Laufzeitverbesserungen, die zwischen 5,0% (Intel/gcc) und 20,5% (Thumb/gcc) liegen. Wird die Zahl der geforderten Berechnungen zu groß gewählt, so werden viele CSEs ignoriert. Das führt zu schlechtem Laufzeitverhalten.

Diese Policy zeigt, daß die Zahl der Vorkommen und damit die Zahl der eingesparten Berechnungen wie erwartet eine große Rolle spielt. Als einziges Maß ist diese Zahl aber wenig sinnvoll. In Kombination mit einer Policy, die die Komplexität eines Ausdrucks

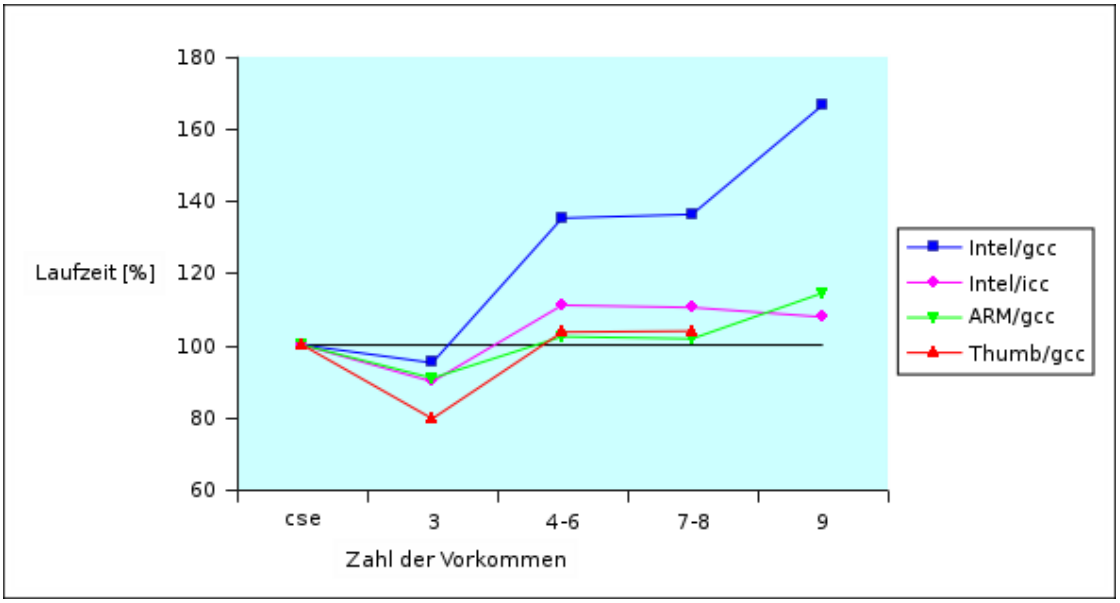

Abbildung 4.4: Ergebnisse der Uses Policy

bewertet, ist sie aber sehr sinnvoll. Es ist auffällig, wie stark die Thumb/gcc Plattform von einer Mindestzahl von drei Vorkommen profitiert.

Die genauen Ergebnisse und die Anzahl der jeweils eliminierten CSEs finden sich in Anhang A.1.3.

4.3.2 Kosten des Ausdrucks

Es ist evident, daß die „Kosten“ eines Ausdrucks einen wichtigen Einfluß auf die Entscheidung haben müssen, ob eine CSE eliminiert werden soll. Eine teure Berechnung einzusparen ist interessanter als eine billige. Um die Kosten zu bestimmen, benötigt man eine plattformabhängige Kostentabelle. Eine solche Tabelle kann durch den Benchmark erstellt werden, der in Kapitel 3.3.2 beschrieben wird. Dabei muß zwischen den Kosten für eine Operation mit zwei Variablen und für eine Operation mit einer Variablen und einer Konstanten unterschieden werden, da im letzteren Fall häufig besondere Optimierungen möglich sind. Mit Hilfe dieser Tabelle können dann leicht die Kosten für einen gefundenen CSE Ausdruck bestimmt werden.

Plattformabhängige Kostentabelle

Es wurden die folgenden Kostentabellen mit dem in Kapitel 3.3.2 beschriebenen Testprogramm gefunden, das die jeweilige Operation in einer Schleife ablaufen läßt. Die jeweilige Laufzeit wird dabei gegen die Laufzeit einer Addition mit einer Konstanten normalisiert. Die Tabelle 4.6 entspricht einer Operation mit einer (kleinen) Konstanten und einer Variablen (z. B. $x + 5$).

Operation	Intel/gcc	Intel/icc	ARM/gcc	Thumb/gcc
add (+)	10	10	10	10
sub (-)	11	10	10	10
mul (*)	9	10	9	9
div (/)	13	13	13	72
rem (%)	31	16	16	90
and (&)	10	11	10	10
or (|)	10	11	10	10
not-equal (!=)	21	11	11	11
less-than (<)	21	12	11	11
less-or-equal (<=)	21	12	11	11
logical-and (&&)	20	11	11	10
logical-or (||)	9	10	9	8

Tabelle 4.6: Kostentabelle für Konstante und Variable

Die zweite Tabelle 4.7 entspricht einer Operation mit zwei Variablen (z. B. $x + y$).

Operation	Intel/gcc	Intel/icc	ARM/gcc	Thumb/gcc
add (+)	10	10	10	10
sub (-)	11	11	10	10
mul (*)	9	12	9	9
div (/)	31	33	21	20
rem (%)	31	33	70	81
and (&)	10	11	10	10
or (|)	10	11	10	10
not-equal (!=)	21	14	11	11
less-than (<)	21	12	11	11
less-or-equal (<=)	21	12	11	11
logical-and (&&)	12	13	12	11
logical-or (||)	21	14	11	11

Tabelle 4.7: Kostentabelle für zwei Variablen

Overhead bei CSE Eliminierung in if()

Durch die Eliminierung von Testausdrücken in `if()` Bedingungen ergeben sich die in Kapitel 3.3.2 vorgestellten Probleme. Mittels eines Benchmarks wurde der Overhead für die untersuchten Plattformen bestimmt. Der Benchmark hat die gleiche Struktur wie der zur Bestimmung der Kosten. Es werden ebenfalls zwei Schleifen eingesetzt. In ihnen wird eine If-Abfrage realisiert, einmal mit einem Testausdruck in der If-Abfrage selbst und einmal mit einer Variablen, die vor der If-Abfrage berechnet wird. Aus den

unterschiedlichen Laufzeiten der beiden Versionen kann der Overhead für das Eliminieren von Tests in If-Ausdrücken bestimmt werden.

Name	Overhead
GCC	50%
ICC	15%
ARM	15%
Thumb	20%

Tabelle 4.8: Overhead bei Eliminierung in if() Ausdrücken

Im Verlauf der Messungen zeigte sich, daß eine noch höhere „Bestrafung“ von durchgehend 80% die besten Ergebenisse liefert. Daher wurde dieser Wert für die folgenden Diagramme verwendet. Im Anhang A.1.5 finden sich die Unterschiede. Umgekehrt erzeugt ein völliger Verzicht auf die Eliminierung von Tests in `if()` Ausdrücken Laufzeitverschlechterungen im Vergleich zu einer Implementierung, die jede CSE eliminiert.

Anwendung auf CAVITY

Die Policy, die für die Bewertung der Kosten zuständig ist, kennt verschiedene Betriebsarten.

Im einfachsten Fall bewertet sie die Kosten eines Ausdrucks und vergleicht, ob die minimal nötigen Kosten erreicht werden, die eine Eliminierung rechtfertigen. Sind die Kosten höher als `minCost`, wird die CSE eliminiert, ansonsten wird sie ignoriert.

In einer zweiten Betriebsart werden die Kosten des Ausdrucks mit der Anzahl der Vorkommen multipliziert. So wird eine lineare Beziehung zwischen Kosten und Zahl der Vorkommen hergestellt. Diese Betriebsart „bevorzugt“ viele billige Ausdrücke gegenüber wenigen teuren Ausdrücken. Es ist zu erwarten, daß die Ergebnisse im Vergleich zu einer reinen Kostenbetrachtung durch die geänderte Bewertung deutlich unempfindlicher gegenüber dem Parameter `minCost` werden.

Eine dritte Meßmethode bewertet die Zahl der Vorkommen und die Kosten als unabhängige Parameter, die jeweils beide erfüllt sein müssen. Hier wurde als minimale Anzahl identischer Ausdrücke drei gefordert und zusätzlich das Kostenkriterium angewendet. Eine CSE wird also eliminiert, wenn mindestens drei Berechnungen eingespart werden und eine bestimmte minimale Kostengrenze überschritten wird.

Unabhängig von der Betriebsart wird jeweils eine „Bestrafung“ für Tests in If-Ausdrücken vorgenommen.

Meßergebnisse

Die Meßergebnisse für die drei untersuchten Methoden werden im folgenden für jede Plattform als Graph dargestellt. Auf der Y-Achse findet sich dabei die Laufzeit relativ zur High-Level CSE Laufzeit, die mit 100% gewertet wird. Auf der X-Achse finden sich die verschiedenen nötigen Kosten, die ein Ausdruck mindestens erreichen muß, damit er

eliminiert wird. Als Referenz finden sich auch „Orig“ für die Originallaufzeit und „CSE“ für die High-Level CSE Laufzeit im Graph.

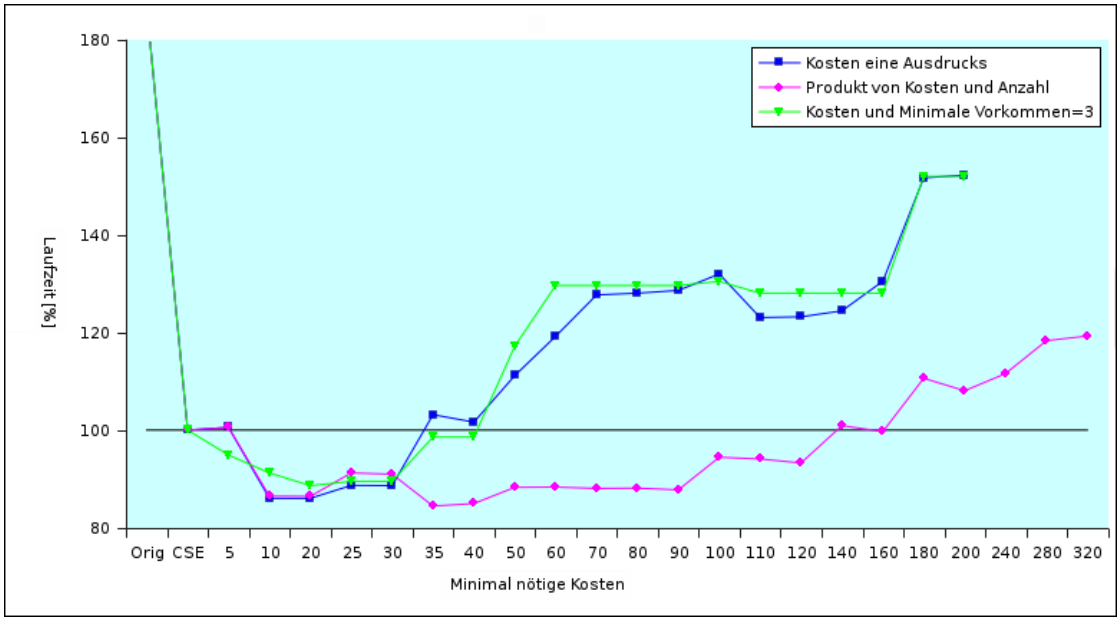

Abbildung 4.5: Meßergebnisse der Kostenbetrachtung für Intel/gcc

Beim Intel/gcc erkennt man in Abbildung 4.5, daß alle Ergebnisse für einen Kostenrahmen von 10–30 ohne jegliche Berücksichtigung der Anzahl der Vorkommen sehr gut sind, es ergeben sich Laufzeitgewinne von 14,0%. Billigere Ausdrücke sind entweder zu einfach und werden schnell neu berechnet, oder es sind Testausdrücke in `if()` Statements, die durch die Strafpunkte unter die Kostengrenze von zehn gedrückt werden. Werden die Kosten auf 35 erhöht, so werden keine einfachen Modulo Ausdrücke mehr eliminiert und die Laufzeit wird schlechter als die Referenzlaufzeit.

Wird die Zahl der Vorkommen mit den Kosten multipliziert, so verändert sich das Bild. Jetzt ist bis zu einer Grenze von 90 Einheiten der Laufzeitgewinn mit 8,8%–15,5% sehr gut. Danach wird auch hier der Laufzeitgewinn mit 5,5%–6,8% schlechter, um erst bei 140 Kosteneinheiten schlechter als die High-Level CSE Version zu werden.

Die unabhängige Betrachtung von Kosten und Vorkommen liefert für diese Plattformen keine interessanten Ergebnisse.

Beim Intel/icc (Abbildung 4.6) zeigt sich ein anderes Bild. Bei reiner Kostenbetrachtung ergeben sich bis zu einer Kostengrenze von 140 Einheiten deutliche Laufzeitgewinne. Die besten Meßwerte zeigen sich aber im Bereich von geringen Kosten zwischen 5 und 25 Einheiten mit einer relativen Laufzeit von 85,4% bei 25 Einheiten.

Wird das Produkt von Kosten und Vorkommen benutzt, so bleibt die Laufzeit unter der Referenzlaufzeit für alle betrachteten Kosten. Ein Minimum von 86,0% wird hier bei 90 Kosteneinheiten erreicht.

Die unabhängige Betrachtung von Kosten und eingesparten Berechnungen liefert keine

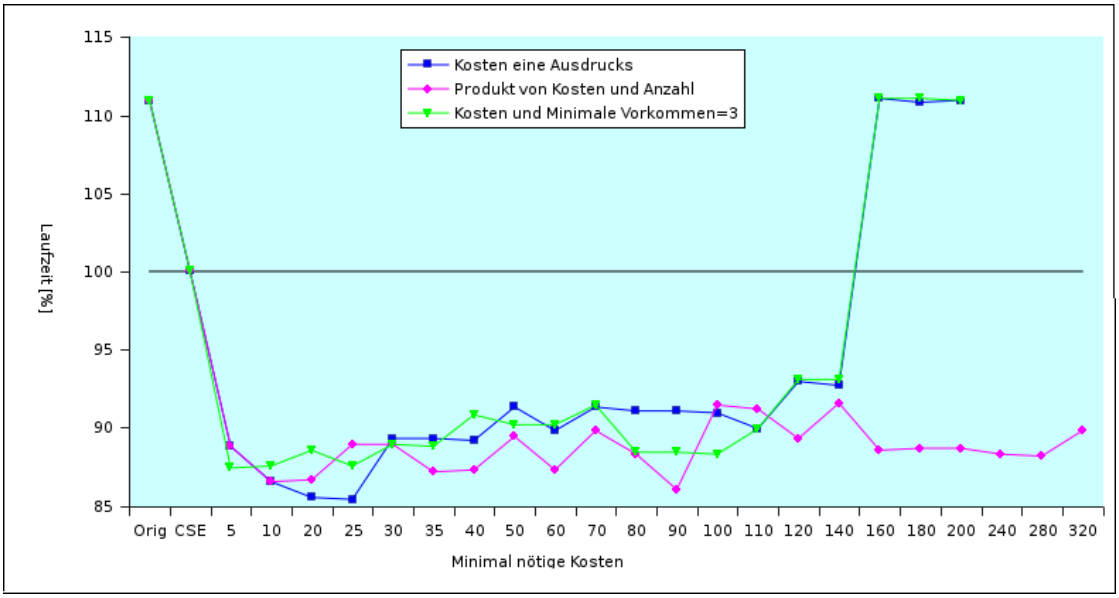

Abbildung 4.6: Meßergebnisse der Kostenbetrachtung für Intel/icc

interessanten Ergebnisse. Dies ist nicht weiter erstaunlich, da ja der Feedback Algorithmus gezeigt hat, daß der ICC sehr hohe Laufzeitgewinne auch durch nur zwei Vorkommen erzielen kann.

Auch beim ARM (Abbildung 4.7) zeigt sich, daß die besten Ergebnisse mit einer kleinen Kostengrenze erreicht werden. Bei der reinen Kostenbetrachtung liegen sie allerdings mit fünf Kosteneinheiten nur bei 92%. Die Laufzeitgewinne sind hier nicht sehr groß.

Wird das Produkt von Kosten und Vorkommen benutzt, so kann ein Minimum von 90,5% erreicht werden. Außerdem bleiben die Laufzeitgewinne bis zu 80 Kosteneinheiten stabil zwischen 6,9-9,5%. Erst ab 100 Einheiten wird die Referenzlaufzeit überschritten.

Die Kosten in Kombination mit dem Kriterium, daß minimal drei Berechnungen eingespart werden müssen, ergeben auf dieser Plattform die besten Ergebnisse. So wird bei den mindestens nötigen Kosteneinheiten 5–10 ein Minimum von 89,5% der Referenzlaufzeit erreicht. Mit der Feedback Analyse wurden 85,4% erreicht.

Werden nur die Kosten betrachtet, so findet sich beim Thumb (Abbildung 4.8) das Minimum von 78,3% bei 10 Kosteneinheiten. Aber schon bei 20 Kosteneinheiten steigt die Laufzeit stark an und ist nur noch 1,6%–3,6% besser als die Referenzlaufzeit, die sie ab 100 Kosteneinheiten überschreitet.

Wird mit dem Produkt von Kosten und Vorkommen gerechnet, so ergibt sich eine stabile Laufzeit von 78,5% bis zu einem Kostenmaß von 30 Einheiten. Interessanterweise gibt es noch einmal sehr gute Ergebnisse von 79,2% für 240 und 280 Kosteneinheiten. Hier werden offensichtlich zum Feedback Ergebnis sehr ähnliche CSEs gefunden.

Die unabhängige Betrachtung von Kosten und eingesparten Berechnungen liefert sehr

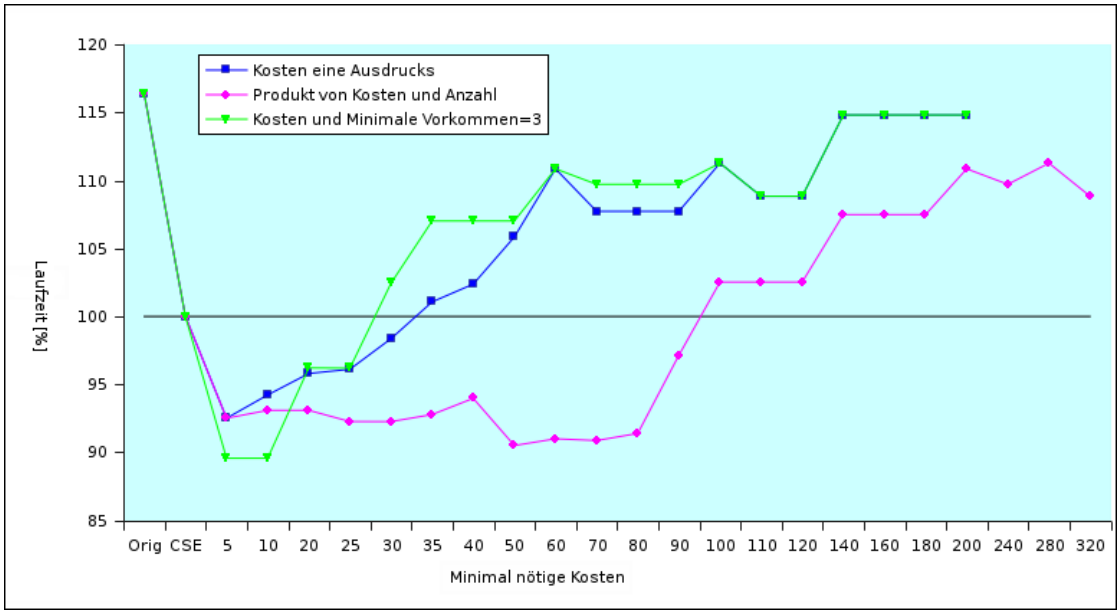

Abbildung 4.7: Meßergebnisse der Kostenbetrachtung für ARM/gcc

stabile Ergebnisse für 5–90 Kosteneinheiten, die jeweils etwa 80% der Referenzlaufzeit haben.

Die gemessenen Benchmarkergebnisse für alle Plattformen und die Zahl der jeweils eliminierten CSEs für die unterschiedlichen Meßmethoden finden sich im Anhang A.1.5.

Zusammenfassung der Kostenergebnisse

Die Kosten eines Ausdrucks spielen eine entscheidende Rolle bei der Frage, ob es sinnvoll ist, einen mehrfach vorkommenden Ausdruck zu eliminieren. In Kombination mit der Zahl der Vorkommen einer CSE ergibt das ein sehr mächtiges Bewertungskriterium. Ein sinnvoller Ansatz bei der Parameterwahl scheint die Wahl eines geringen Wertes für die minimal nötigen Kosten zu sein. Nur Ausdrücke, die sehr einfach sind, werden dann ignoriert, alle anderen werden eliminiert. Dies deckt sich mit der eingangs geäußerten Vermutung, daß nur sehr einfache CSEs oder Tests in `if()` Ausdrücken nicht sinnvoll sind, in den meisten anderen Fällen ist eine Eliminierung sinnvoll.

Nimmt man nur das Kostenkriterium als Grundlage, so ist es sinnvoll, Tests in if() Ausdrücken nur schwach zu bewerten (Kosten nur noch 20% der Ursprungskosten). Minimale Kosten von 10 Einheiten und eine solche Bewertung für If-Ausdrücke führen zu Laufzeiten, wie sie in der Abbildung 4.9 zu sehen sind. In diesem Diagramm finden sich auf der X-Achse die untersuchten Plattformen, auf der Y-Achse die Laufzeit mit der High-Level CSE Version als Referenz mit 100%. Der erste Balken repräsentiert die Laufzeit des unmodifizierten CAVITY Benchmark, der zweite Balken die High-Level CSE

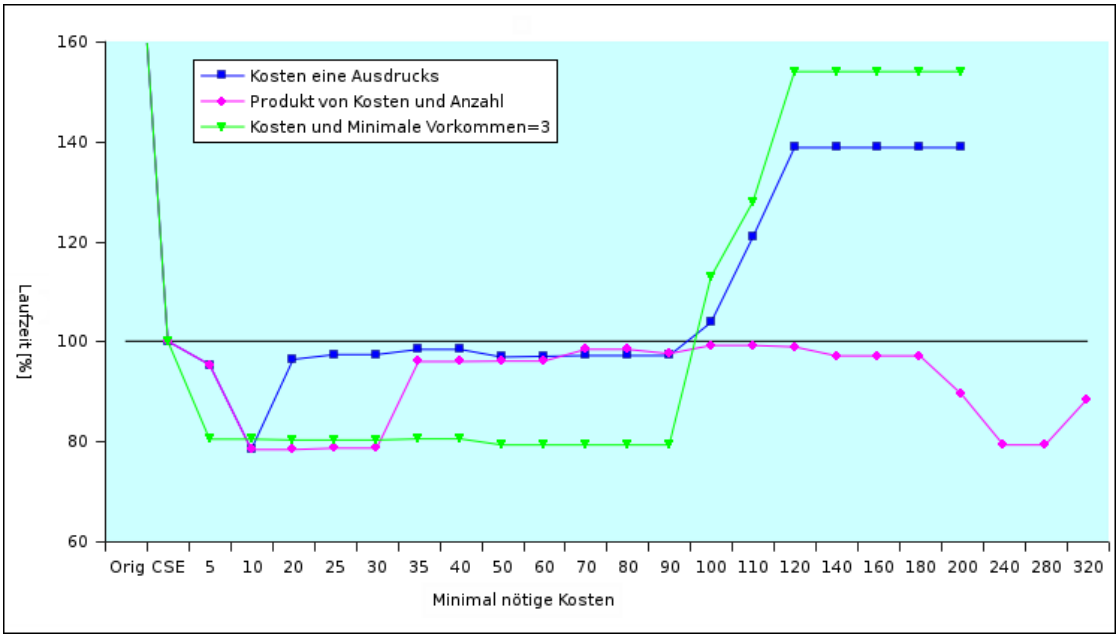

Abbildung 4.8: Meßergebnisse der Kostenbetrachtung für Thumb/gcc

Version, der dritte Balken das Feedback Ergebnis und der letzte Balken die Laufzeit, wenn minimal erreichte Kosten von 10 Einheiten gefordert werden.

Man erkennt, daß bei dieser Parameterwahl die Laufzeit sehr nah an die Ergebnisse der Feedbackoptimierung kommt. Für den Thumb und den ICC Test konnten die Ergebnisse sogar leicht übertroffen werden. Dies ist damit zu erklären, daß die Feedback Optimierung versucht, „gute“ CSEs zu identifizieren und diese hinzuzufügen. Bei einer Eliminierung ab Kosten von 10 Einheiten werden dagegen lediglich „schlechte“ CSEs nicht eliminiert, alle anderen schon. Die Ergebnisse deuten darauf hin, daß dies ein sinnvoller Ansatz ist.

Nicht optimal sind die Ergebnisse der ARM/gcc Plattform. Ein weiteres Problem ist die Empfindlichkeit der Parameter. Wird z. B. beim Thumb mit minimalen Kosten von 20 statt 10 gearbeitet, so verändert sich die Laufzeit stark (von 78,3% auf 96,3%).

4.3.3 Zahl der aktiven Variablen

Diese Policy eliminiert eine gefundene CSE nur dann, wenn nicht zu viele andere Variablen aktiv sind, um so den Registerdruck nicht noch weiter zu erhöhen. Die Implementierung der LifetimePolicy wird in Kapitel 3.3.2 vorgestellt.

Die Ergebnisse stellt die Grafik 4.10 zusammen. Auf der Y-Achse findet sich die Laufzeit in Prozent, 100% stellt dabei die High-Level CSE Version dar. Auf der X-Achse findet sich die maximale Zahl an aktiven Variablen. Wird diese Zahl überschritten, so wird keine CSE Eliminierung an der Stelle des mehrfach vorkommenden Ausdrucks durchgeführt.

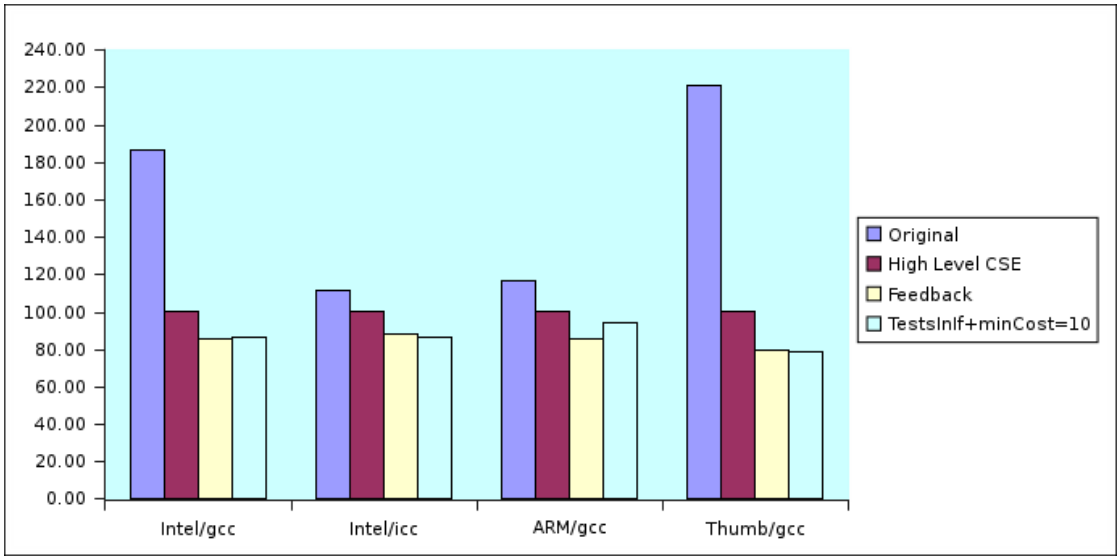

Abbildung 4.9: Zusammenfassung der Kostenergebnisse bei TestInIf und minCost=10

Mit „Orig“ wird dabei noch einmal die Laufzeit des unmodifizierten Referenzquellcodes angegeben.

Eine zu kleine Zahl führt dazu, daß kaum CSEs eliminiert werden, die Laufzeit liegt dann sehr nah an der Originalversion und ist deutlich schlechter als die Referenzlaufzeit der High-Level CSE Optimierung.

Wird die Zahl der aktiven Variablen groß gewählt, so nähert sich die Laufzeit der High-Level CSE Version an, da dann alle gefundenen Ausdrücke eliminiert werden. Die besten Ergebnisse finden sich, wenn maximal 20–25 lebendige Variablen zugelassen werden, der Einfluß ist mit 0,5%–2,7% für die GCC Plattformen und 8,2% für die ICC Plattform allerdings nicht besonders groß.

Die Zahlen zeigen, daß die Anzahl der aktiven Variablen eine Größe ist, die Einfluß auf die Laufzeit hat. Als alleiniges Kriterium ist sie allerdings zu schwach, da es bei einer sehr teuren Berechnung durchaus sinnvoll sein kann, sie zu eliminieren und dadurch den Registerdruck zu erhöhen. Sie muß mit anderen Methoden kombiniert werden.

Ein Problem für diese Policy ist, daß sie nur eine grobe Annäherung an den tatsächlichen Registerdruck darstellt, da die Registerallokation normalerweise zu einem späten Zeitpunkt auf der Low-Level Zwischensprache stattfindet und mehr als nur Variablen in Betracht zieht.

Die exakten Zahlen und die Zahl der jeweils eliminierten CSEs finden sich im Anhang A.1.4.

4.3.4 Abstand zwischen zwei Vorkommen eines Ausdrucks

Bei der Erzeugung von CSE Ausdrücken und dem anschließenden Ersetzen des Ausdrucks durch die neu erzeugte Variable kann es vorkommen, daß zwischen zwei Variablen ein

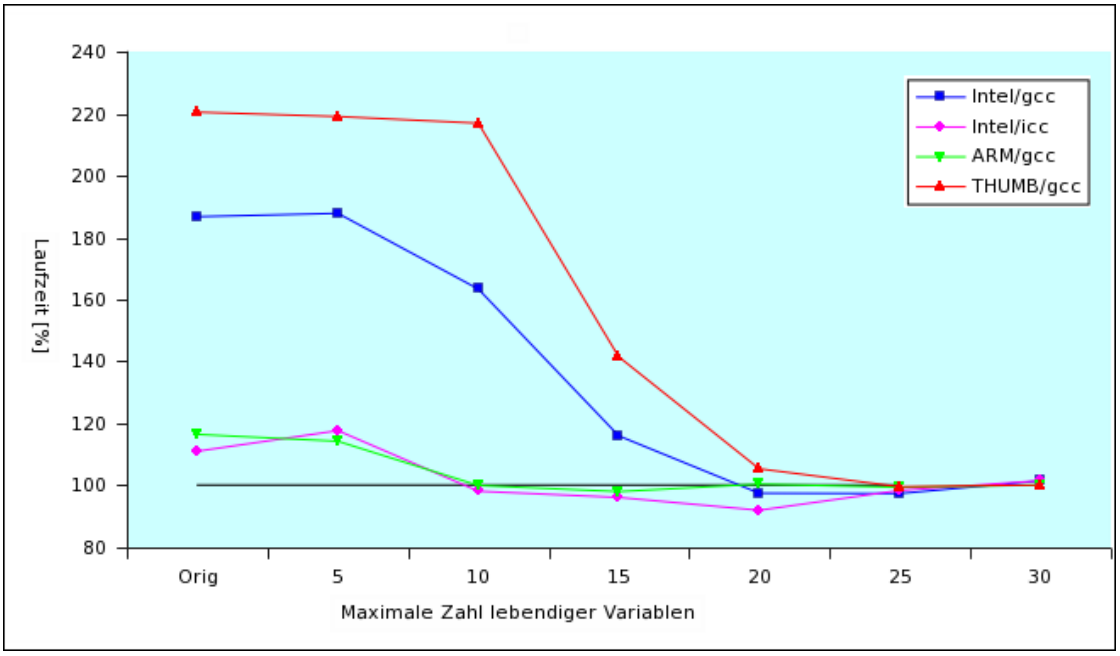

Abbildung 4.10: Ergebnisse der Lifetime Policy

großer Abstand liegt. In diesem Fall ist die CSE Optimierung unter Umständen nicht sinnvoll, da über den gesamten Abstand ein Register für die Variable belegt wird oder ein teures Nachladen aus dem Speicher nötig ist. Die DistanceCountPolicy, die in Kapitel 3.3.2 beschrieben wird, betrachtet die Abstände zwischen zwei Vorkommen einer neuen CSE Variable. Als Abstand wird dabei die Zahl der Instruktionen auf dem kürzesten Pfad im Syntaxbaum betrachtet, die zwischen zwei Vorkommen liegt. Wird dieser Abstand überschritten, so wird die Lebenszeit der CSE-Variablen explizit geteilt.

Ergebnisse

Mit verschiedenen maximal erlaubten Abständen wurde der CAVITY Benchmark gemessen. Die folgende Tabelle zeigt die Charakteristika der erzeugten Quellcodes. Dabei ist der maximale Abstand der Wert, der nicht überschritten werden darf, sonst wird die entsprechende CSE-Variable aufgeteilt. Die Spalte „CSEs“ gibt die Zahl der eingefügten CSE Variablen, die Spalte „Zahl der Nutzungen“ gibt die Zahl der Vorkommen der neuen CSE-Variablen an.

Maximaler Abstand	Anzahl CSEs	Zahl der Nutzungen
–	38	105
50	18	44
100	23	56
150	34	83
200	41	105
250	36	103
300	36	103
350	36	98
400	38	102
450	38	103
500	38	103

Beim Aufteilen kommt es vor, daß die aufgeteilten CSEs nur noch ein Vorkommen haben. Dies ist zum Beispiel bei einer CSE der Fall, die nur zwei Vorkommen hat und den maximal erlaubten Abstand überschreitet. Die beiden neuen CSE-Variablen haben nach der Teilung der Lebenszeit nur noch eine Nutzung und werden jeweils vom copy-propagation Schritt entfernt. Dies erklärt, warum insbesondere bei kleinen Abständen nur ein Teil aller CSEs eliminiert wird.

Die Abbildung 4.11 stellt die Ergebnisse graphisch dar. Auf der Y-Achse findet sich die Laufzeit relativ zur High-Level CSE Version. Auf der X-Achse befindet sich der maximale Abstand. Wird er überschritten, so wird die CSE in zwei unterschiedliche CSEs aufgeteilt. Die exakten Zahlen finden sich im Anhang A.1.6.

Für alle GCC-basierten Plattformen zeigt sich, daß zu kleine Werte für den Abstand zu einer starken Fragmentierung führen und damit keine Verbesserung der Laufzeit erzielt werden kann. Bei großen Werten nähert sich die Laufzeit der unmodifizierten CSE Version an. Da große Abstände seltener sind, sind hier die Veränderungen relativ gering.

Die besten Ergebnisse in allen untersuchten GCC Versionen brachte ein Abstand von 350 Instruktionen, während ein Abstand von 300 keine Verbesserung mehr erzielt. Die Laufzeitverbesserungen sind mit 1,0%–6,7% aber nicht besonders groß. Die komplexen CSEs cse_050_70, cse_053_72 und cse_055_73 sind bei MaxDist=350 noch im Quellcode. Bei 300 dagegen wird je eines von drei Vorkommen von cse_050_70, cse_053_72 und cse_053_72 entfernt, da die Variablen dann in cse_050_70 und cse_050_70split1 aufgeteilt werden. Die erste Variable kommt dann zweimal im Quellcode vor, die letzte nur noch einmal. Diese wird dann durch den abschließenden copy-propagation Schritt entfernt. Damit erklärt sich der relativ deutliche Abfall der Laufzeit bei einem maximal erlaubten Abstand von 300 Instruktionen.

Für die Intel/icc Plattform ergibt sich ein abweichendes Bild. Der ICC profitiert stark von der Aufsplittung in kurze Abstände. Hier ist eine Verbesserung von 13% gegenüber der High-Level CSE Optimierung möglich, und es kommt zu keinen Laufzeitverschlechterungen, egal welcher maximale Abstand gewählt wird.

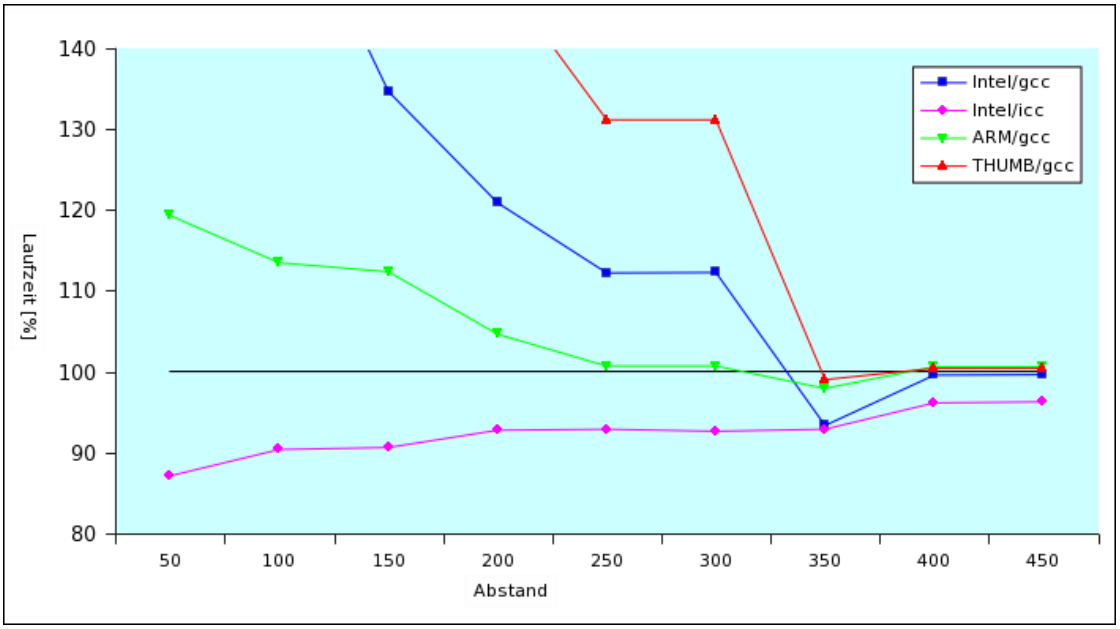

Abbildung 4.11: Einfluß des Abstands auf die Laufzeit

Analyse der Abstandsergebnisse

Eine Frage, die sich aus den Abstandsbetrachtungen ergibt, ist ob sich die Ergebnisse der Feedback Analyse durch diese Abstandsbetrachtung verbessern lassen. Beispielhaft werden die Abstände der gefundenen CSEs betrachtet, die durch die Feedback Methode beim Intel/gcc entdeckt wurden. Hier ergab sich eine Verbesserung von 6,7% bei einem Abstand von 350 Instruktionen.

CSE	Abstände
cse_000_0	10 **812** 34 233 45
cse_004_26	109 34 278 10 13 266 34 31 **398** 54 28
cse_012_34	168 224 266 99 398 59
cse_015_37	101 211 19 266 51 64 51 398 71 42 30
cse_023_45	25 233 55 101 211 10 13 266 34 31 **398** 25 30
cse_025_47	215 270 94 398 40
cse_050_70	214 **398**
cse_053_72	166 **398**
cse_055_73	187 **398**

Tabelle 4.9: Abstände der Feedback-Ergebnisse des Intel/gcc

Es ist auffällig, daß die Abstände meist relativ gering sind, nur ein einziger Abstand mit 812 Instruktionen und fünf Abstände mit 398 Instruktionen sind stechen hervor, alle anderen liegen unter 300 Instruktionen.

Ein maximal erlaubter Abstand von weniger als 398 Instruktionen würde dazu führen, daß die cse_050_70, cse_053_72 und cse_055_73 gesplittet werden und damit das letzte Vorkommen der entsprechenden CSE Variablen vom copy-propagation Schritt entfernt würde. Bei diesen wichtigen CSEs ist ein solches Entfernen aber nicht wünschenswert. Wird es durchgeführt, so führt das zu Laufzeitverschlechterungen.

Bei noch größeren Abständen wird nur cse_000_0 beeinflußt, sonst ergeben sich keine Änderungen. Dies führt zu keinen meßbaren Laufzeitveränderungen.

4.4 Neuronale Netze

Im Verlauf der Untersuchung zeigte sich, daß die Klassifikation der Parameter bei der CSE Analyse entscheidende Bedeutung hat. Nur eine gute Auswahl führt zu Performancesteigerungen. Eine Klassifikation der plattformabhängigen Parameter auf der Basis von neuronalen Netzen bietet sich an. Dazu wird das Netz zunächst mit einer Reihe von Beispieldaten trainiert (wie in Kapitel 3.5 beschrieben). Das so gewonnene Netz kann dann unbekannte CSEs anhand der vorher präsentierten Trainingsbeispiele bewerten.

4.4.1 Anwendung auf den CAVITY Benchmark, Methode 1

Die erzeugten Netze können nun auf den CAVITY Benchmark angewendet werden. Es ist zu beachten, daß hier auf der Ebene der einzelnen Vorkommen einer CSE gearbeitet wird, da Informationen wie die Zahl der aktiven Variablen an jeder konkreten Stelle verschieden sind.

Die Netze machen bei Methode 1 eine Vorhersage über die Performance des Programms nach Eliminierung einer CSE. Die plattformabhängige High-Level CSE Optimierung kann nun mit der mindestens nötigen Performanceänderung parametrisiert werden, die eine CSE erreichen muß, damit sie eliminiert wird. Ein Parameter von 100% bedeutet also, daß nur CSEs eliminiert werden, wenn die Laufzeit des resultierenden Programms kleiner ist als die Ursprungslaufzeit.

In Abbildung 4.12 findet sich eine Zusammenfassung der Werte in einem übersichtlichen Diagramm. Es wird mit der „Bestrafung“ für Tests in If-Ausdrücken gearbeitet, die in Kapitel 4.3.2 (Seite 73) für die jeweils untersuchte Plattform gefunden wurden. Die genauen Meßergebnisse mit den Details zur Zahl der versteckten Neuronen und Menge der Trainingsintervalle finden sich im Anhang A.1.7.

Im Diagramm befindet sich auf der Y-Achse die Laufzeit der erzeugten Quellcodes in Prozent. Diese Quellcodes werden jeweils aus den Parametern der X-Achse erzeugt. Als Referenz auf der Y-Achse ist die High-Level CSE Optimierung mit 100% angesetzt.

Auf der X-Achse wird festgelegt, welche Vorhersage das neuronale Netz für eine einzelne gefundene CSE machen muß, damit sie eliminiert wird. Der Wert 100% auf der X-Achse meint also, daß das Netz eine maximale Laufzeitprognose für jedes gefundene CSE-Vorkommen von 100% oder kleiner abgeben darf. Wird ein größerer Wert vom Netz berechnet, so wird die CSE ignoriert. Es ist zu beachten, daß der Parameter auf der

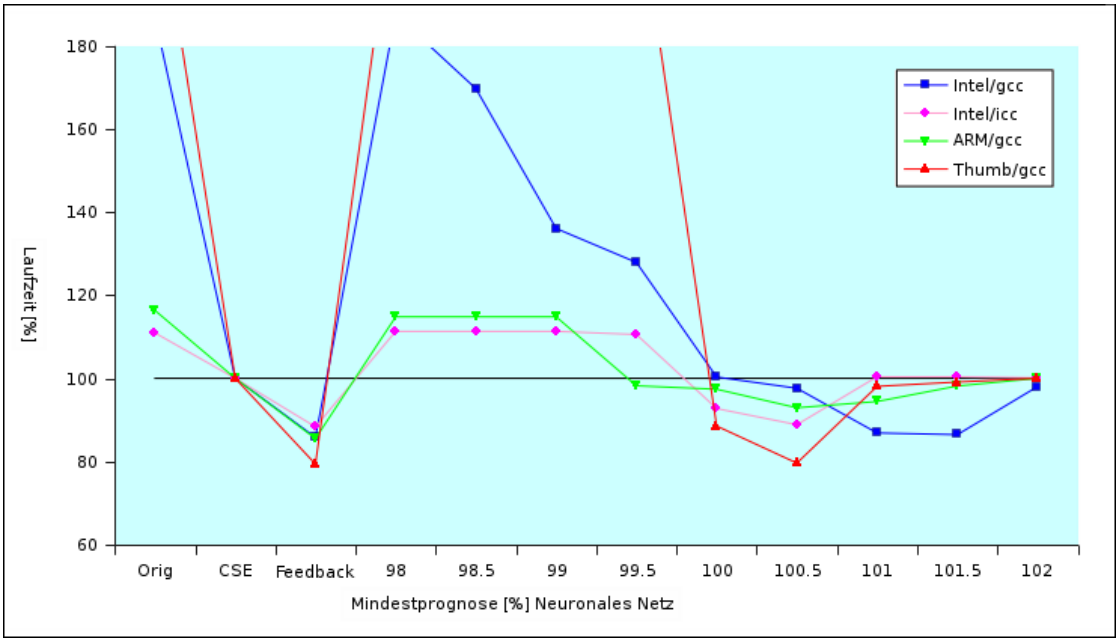

Abbildung 4.12: Methode 1, mit TestInIf Bestrafung

X-Achse die Laufzeitveränderung eines einzelnen CSE-Vorkommens darstellt und keine Vorhersage über die Laufzeit aller CSEs darstellt.

Für den Intel/gcc zeigen sich ab einer Vorhersage der Laufzeit von 100% gute Ergebnisse. Vorher werden zu wenige CSEs eliminiert, die Laufzeit ist durchweg schlechter als die Referenzlaufzeit. Bei einer Vorhersage der Laufzeit von 101% wird ein Laufzeitgewinn auf 86,8% erreicht. Das ist nur ein Prozent von der Feedback-Laufzeit entfernt. Es ist ungewöhnlich, daß dieses Ergebnis bei 101% erreicht wird, passt aber zu der Beobachtung beim Untersuchen der Kosten (vgl. Kapitel 4.3.2). Auch hier war es sinnvoll, nur besonders schlechte CSEs zu ignorieren und alle anderen zu eliminieren. Beim Intel/icc ist die Beobachtung ähnlich. Hier wird das beste Ergebnis bei 100,5% erreicht. An dieser Stelle verbessert sich die Laufzeit auf 88,8% gegenüber der High-Level CSE Laufzeit. Das ist nur 0,5% von der Feedback-Laufzeit entfernt. Beim ARM/gcc wird zwar auch bei 100,5% die beste Laufzeitverbesserung auf 92,9% gemessen, dieser Wert ist allerdings weit von den 85,4% des Feedback-Ergebnisses entfernt. Beim Thumb/gcc ist das Bild dagegen wieder positiver. Auch hier wird bei 100,5% die größte Verbesserung auf 79,5% erreicht, und dies ist nur 0,3% schlechter als das Feedback Ergebnis.

Die Abbildung 4.13 zeigt das Ergebnis, wenn mit einer stärkeren „Bestrafung“ von Tests in If-Ausdrücken gearbeitet wird (nur noch 20% der Ursprungskosten). Hier ändert sich das Bild. Der Intel/gcc hat bei einer Vorhersage der Laufzeit der gefundenen CSE von 100% oder kleiner eine sehr gute Verbesserung der Laufzeit des erzeugten Programms auf 86,8% der High-Level CSE Laufzeit. Sein bestes Ergebnis wird bei einer maximal erlaubten Prognose von 100,5% oder kleiner erreicht. Dann sinkt die Laufzeit auf 85,9%.

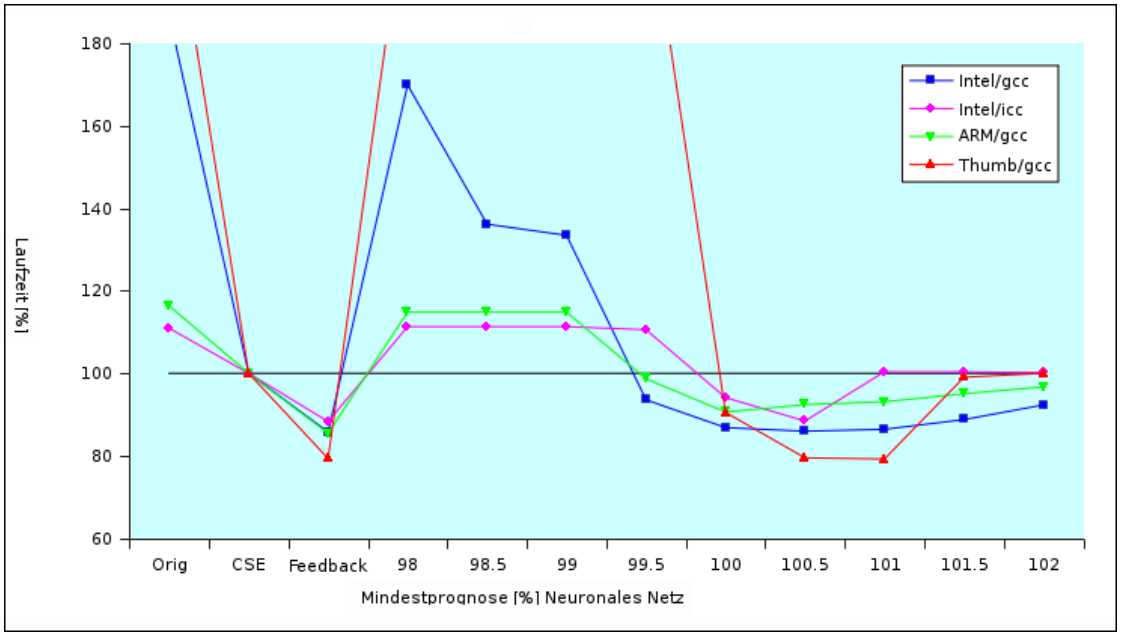

Abbildung 4.13: Methode 1, mit TestInIf Bestrafung von 80%

Beim Intel/icc sind die Veränderungen nur sehr gering, während beim ARM/gcc bei einer Prognose von 100% oder kleiner eine Laufzeitverbesserung auf 90,6% erreicht wird. Beim Thumb/gcc werden nun für die Prognose 100,5% und 101% jeweils sehr gute 79,4% und 79,2% Laufzeit gemessen.

4.4.2 Anwendung auf den CAVITY Benchmark, Methode 2

Bei Methode 2 wird die „Güte“ der CSE als Wert zwischen 0 und 1 angegeben. Je höher der Wert, desto besser bewertet das Netz die CSE, d.h. desto stärker schätzt sie den Laufzeitgewinn ein, der bei Eliminierung erreicht wird. Kleine Werte nahe Null liefern nur wenig oder keinen Gewinn, große Werte nahe Eins liefern große Gewinne.

In Abbildung 4.14 findet sich eine Zusammenfassung der Werte in einem übersichtlichen Diagramm. Die genauen Zahlen finden sich im Anhang A.38. Im Diagramm findet sich auf der Y-Achse die Laufzeiten der erzeugten Quellcodes und auf der X-Achse die Vorhersage über die Güte der CSE, die mindestens erreicht werden muß, damit eine CSE eliminiert wird. Ein Wert von 0,4 bedeutet also, daß für jede gefundene CSE eine Vorhersage von 0,4 oder größer erzielt werden muß, damit die CSE auch eliminiert wird.

Beim Intel/gcc ergeben sich im Bereich 0,1–0,5 sehr gute Ergebnisse, die jeweils deutlich über 10% Laufzeitgewinn liegen. Bei 0,3 ergibt sich ein Minimum, das das Feedback-Ergebnis sogar um 0,9% übertrifft. Ab einer Güte von 0,6 werden zu viele CSEs ignoriert und die Laufzeit übersteigt den Referenzwert von 100%. Beim Intel/icc finden sich die besten Werte bei 0,2 und 0,3 mit jeweils etwas über 11% Laufzeitgewinn. Danach übersteigt auch hier die Laufzeit die Referenzlaufzeit. Beim ARM/gcc liegen die Lauf-

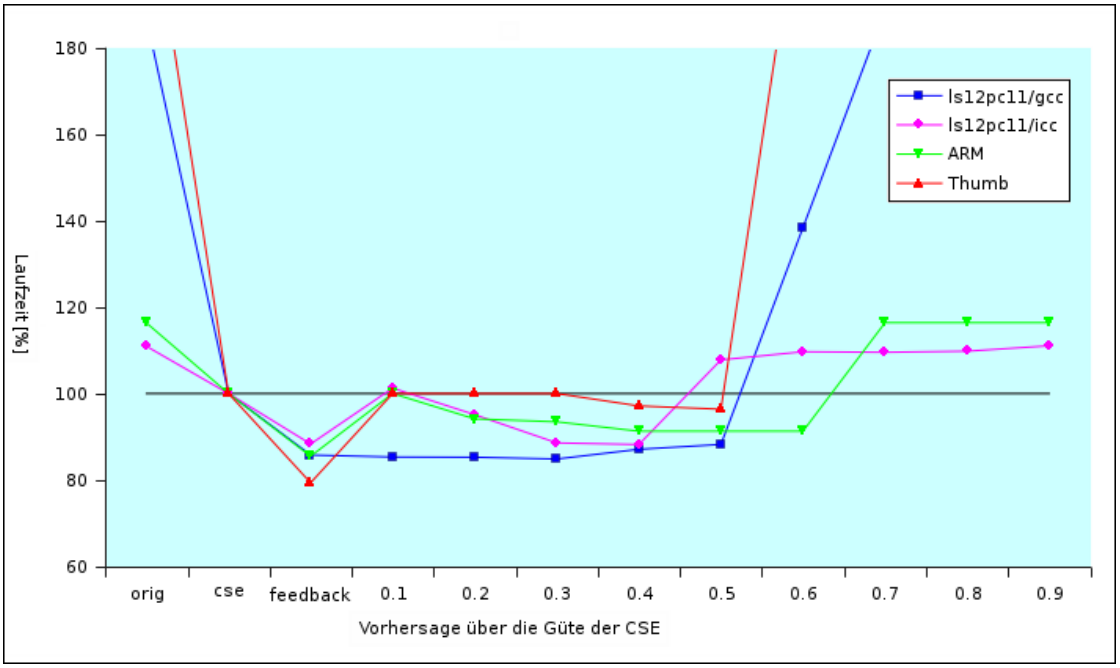

Abbildung 4.14: Methode 2, mit TestInIf Bestrafung

zeitgewinne für 0,4–0,6 bei 8,7% und damit deutlich unter den Feedback Ergebnissen. Beim Thumb/gcc fallen die Ergebnisse nicht gut aus, es ist nur 3,6% Laufzeitgewinn zu messen. Interessant ist, daß die guten Ergebnisse im Bereich von niedrigen Vorhersagen liegen. Auch hier scheint es wichtiger, schlechte CSEs auszuschließen, als nur die besten zu finden.

In Abbildung 4.15 finden sich noch einmal die gleichen Messungen, allerdings mit einer stärkeren „Bestrafung“ von Tests in If-Ausdrücken. Beim Intel/gcc fällt auf, daß die Ergebnisse für den Bereich von 0,1–0,5 mit 13,5% Laufzeitgewinn sehr stabil sind. Auch beim Intel/icc hat sich der Bereich mit sehr guten Ergebnissen vergrößert. Dabei wurden bei 0,2 die Ergebnisse der Feedback-Analyse leicht um 0,7% übertroffen. Beim ARM/gcc haben sich die Ergebnisse dafür leicht verschlechtert. Es werden nur noch maximal 5,5% Verbesserung gefunden. Beim Thumb/gcc läßt sich dagegen wieder eine Vergrößerung der guten Ergebnisse beobachten. Für die Werte 0,1–0,4 werden Laufzeitgewinne von 21,7% gemessen. Das ist 1% besser als die Feedback-Laufzeiten.

Auch bei Methode 2 scheint es wichtiger zu sein, schlechte CSEs zu verhindern, als wenige sehr gute zu eliminieren. Insgesamt liefern beide Methoden für alle Plattformen bis auf den ARM/gcc Ergebnisse, die sehr nah an den Ergebnissen der Feedback Analyse sind.

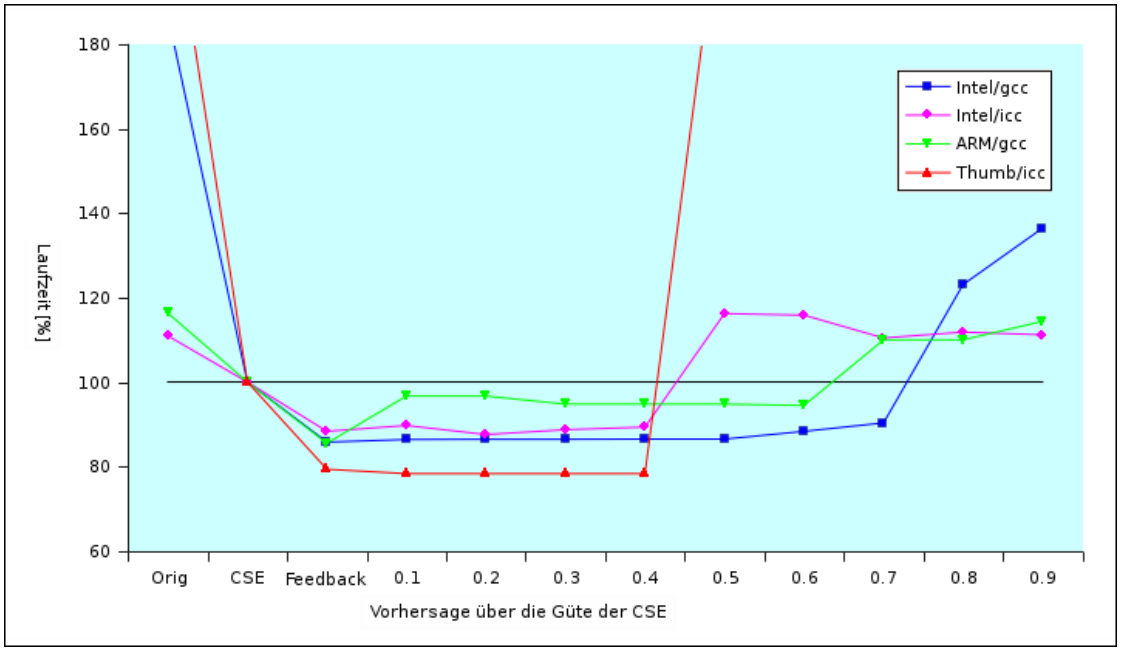

Abbildung 4.15: Methode 2, mit TestInIf Bestrafung von 80%

4.5 Zusammenfassung der Ergebnisse

Werden die Policies isoliert angewendet, so stellt die Kosten-Policy das stärkste Kriterium dar. In Verbindung mit der „Bestrafung“ von Tests in If-Ausdrücken ergeben sich für alle Plattformen deutliche Laufzeitgewinne die sehr nah an den Feedback-Ergebnissen liegen. Nur beim ARM/gcc fehlen etwa 5% zur Feedback-Laufzeit. Die anderen Policies sind isoliert nicht so interessant, sie sind aber zur Entscheidungshilfe wichtig, da sie zusätzliche Informationen, wie die Zahl der aktiven Variablen, liefern.

Um diese zusätzlichen Informationen zu verwenden, wurde eine Methode der Klassifikation auf der Basis von neuronalen Netzen entwickelt. So ist es möglich, deutliche Verbesserungen gegenüber der High Level CSE Methode zu erzielen und Ergebnisse zu erreichen, die sehr nah an den Feedback-Ergebnissen sind oder sie sogar leicht übertreffen. Nur für den ARM/gcc konnten diese Ergebnisse nicht erreicht werden.

Wird mit den Netzen, die nach Methode 1 trainiert wurden, mit einer Prognose von 100,5% oder kleiner für jedes gefundene CSE-Vorkommen gearbeitet, so ergeben sich die folgenden Laufzeiten:

Plattform	Feedback [%]	Neuronales Netz [%]
Intel/gcc	85,9	85,9
Intel/icc	88,3	88,4
ARM/gcc	85,4	92,4
Thumb/gcc	79,3	79,4

Es ist deutlich zu erkennen, daß das neuronale Netz jeweils die Feedback-Ergebnisse erreichen konnte und damit ein sehr gutes Klassifizierungsinstrument darstellt.

5 Abschluß

Dieses Kapitel faßt die gefundenen Ergebnisse noch einmal zusammen und gibt einen Ausblick auf mögliche zukünftige Arbeiten.

5.1 Zusammenfassung

Im Verlauf der Arbeit wurde die Frage untersucht, welche plattformabhängigen Parameter bei einer CSE Einfluß auf die Laufzeit haben. Dazu wurde die Implementierung der High-Level CSE auf Quellcode-Ebene so erweitert, daß plattformabhängige Informationen berücksichtigt werden. Als Parameter wurden die Anzahl der Vorkommen einer CSE, ihre Kosten, die Zahl der lebendigen Variablen in jedem Vorkommen und der Abstand zwischen zwei Vorkommen untersucht. Für die Kosten wurde eine plattformabhängige Kostentabelle erstellt, die zusätzliche Kontextinformationen, wie beispielsweise ob die CSE in einem If-Ausdruck vorkommt, bewertet. Diese Parameter wurden sowohl einzeln untersucht, als auch mit einem neuronalen Netz in ihrer Kombination bewertet und gewichtet.

Die Arbeit konnte zeigen, daß eine plattformabhängige Erweiterung der High-Level CSE Optimierung auf Quellcode-Ebene sinnvoll ist. Durch zusätzliche Bewertungsmöglichkeiten der gefundenen CSEs und unter Einbeziehung von plattformabhängigen Informationen konnten verglichen mit der Original Version von CAVITY ohne jegliche CSE-Optimierung Verbesserungen von 22,1% bis 64,5% erreicht werden. Verglichen mit der bisher vorliegenden High-Level CSE auf Quellcode-Ebene konnten Verbesserungen zwischen 9,4% und 21,7% für den CAVITY Benchmark gemessen werden. Dabei wurde die Entscheidung über die Eliminierung jeweils durch ein neuronales Netz gefällt.

Von den einzelnen untersuchten Kriterien stellen wie erwartet die Kosten das stärkste Werkzeug dar. Wird die Entscheidung über die Eliminierung nur durch ein Kostenkriterium gefällt, so werden Laufzeitgewinne zwischen 9,5% und 21,7% gegenüber der High-Level CSE erreicht. Hier ist die sinnvollste Strategie eine vorsichtige Auswahl der CSEs, die ignoriert werden sollen. Die meisten CSEs scheinen zu Laufzeitverbesserungen beizutragen. Werden nur besonders einfache Ausdrücke oder Test-Ausdrücke in `if()` Abfragen ignoriert, so wirkt sich der geringere Registerdruck und die Einsparung von unnötigen Eliminierungen positiv auf die Laufzeit aus. Negative Effekte durch die Auswahl sind nahezu ausgeschlossen, wie die in dieser Arbeit präsentierten Ergebnisse gezeigt haben.

Eine heuristische Abschätzung des Optimierungspotentials einer plattformabhängigen CSE auf Quellcode-Ebene durch den Feedback-Algorithmus ergab Verbesserungsmöglichkeiten im Bereich von 11,7% bis 21,7%, welche durch die in der vorliegenden Arbeit

präsentierten Techniken auf der Grundlage von neuronalen Netzen ausgeschöpft oder sogar übertroffen werden.

5.2 Ausblick

Es bieten sich eine Reihe von weiteren Untersuchungen an. Zunächst wäre eine Portierung der vorgestellten Techniken auf andere Plattformen und andere Benchmarks interessant. Erste Ergebnisse der vorgestellten Techniken mit dem QSDPCM-Benchmark [Str88], der aus dem Bereich des Video-Coding stammt, sind sehr positiv.

Weiter könnten an den Policies Verbesserungen vorgenommen werden. Für die Kosten-Policy könnte ein weiteres neuronales Netz entwickelt werden, so daß die Kosten ebenfalls über ein neuronales Netz bestimmt werden (statt wie bisher per Benchmark). Eine interessante neue Policy wäre eine Bewertung des Scalar Replacement. Bei dieser Technik werden Zugriffe auf einzelne Array Elemente in CSE-Variablen zwischengespeichert (vgl. Kapitel 1.1.2). Der CAVITY Benchmark und der im folgenden vorgestellte QSDPCM Benchmark bieten einige Möglichkeiten für diese Optimierung. Eine nähere Analyse, an welchen Stellen eine solche sinnvoll ist, wäre wünschenswert.

Auch das neuronale Netz ließe sich verbessern. Die Ergebnisse mit einem anderen Typ von neuronalem Netz wie „Cascade Correlation Learning Architecture“ [Zel94], dessen Struktur nicht im vorhinein festgelegt werden muß, wären interessant. Dieser Netztyp ist, wie die Multilayer-Feedforward Netze, gut für Klassifizierungsaufgaben geeignet. Der Vorteil wäre hier, daß strukturelle Aspekte, wie die Zahl der Neuronen in der verdeckten Schicht, während des Trainings vom Netz selbst erzeugt werden. Beim Multilayer-Feedforward Netz müssen diese Parameter vorher festgelegt werden.

Ein weiteres Problem sind die Abhängigkeiten der CSEs untereinander. Es kommt häufig vor, daß eine CSE eine andere enthält, und es gibt teilweise CSEs, die ausschließlich in anderen CSEs vorkommen und sonst keinen Variablen zugewiesen werden. Eine Untersuchung über die optimale Strategie für diese Klasse von CSE Variablen mit einem Abhängigkeitsgraphen könnte eine weitere interessante Untersuchung sein. Statt die CSEs isoliert zu betrachten, könnte man jeweils Cluster untersuchen.

5.2.1 Ausblick auf QSDPCM

Es wurden einige Messungen mit dem QSDPCM Benchmark durchgeführt. Neben den Feedback-Messungen auf allen Plattformen wurden auf den Intel/gcc und Intel/icc Plattformen die Standardpolicies untersucht. Weitere Untersuchungen waren aus Zeitgründen nicht möglich.

Feedback Ergebnisse

In der Abbildung 5.1 werden die Ergebnisse der Feedback Messungen graphisch dargestellt. Auf der X-Achse finden sich die vier untersuchten Plattformen, auf der Y-Achse die Laufzeit. Dabei ist die Referenz mit 100% die Originallaufzeit des unmodifizierten Quellcodes. Der linke Balken stellt die Originallaufzeit des unmodifizierten Quellcodes

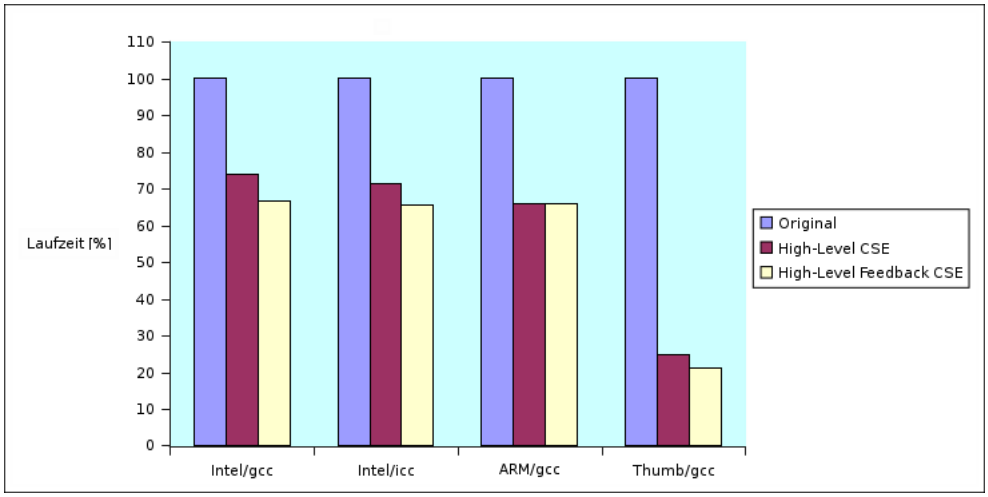

Abbildung 5.1: Feedbackergebnisse für den QSDPCM Benchmark

dar, der mittlere Balken die Laufzeit nach Eliminierung aller CSEs und der rechte Balken das Ergebnis der Feedback-Optimierung.

Es ist zu erkennen, daß auf allen Plattformen deutliche Laufzeitgewinne zwischen 7,9% und 15,0% durch Auswahl der eliminierten CSEs erreicht werden können. Nur für den ARM/gcc können keine Gewinne erzielt werden. Die genauen Laufzeiten werden im Anhang A.2 angegeben.

Ergebnisse der einzelnen Policies

Bei den einzelnen Policies hat nur die Kosten-Policy direkt verwertbare Ergebnisse geliefert. Die anderen Policies haben isoliert nie mehr als 2% Laufzeitveränderung ergeben. Zunächst werden die Ergebnisse für die Intel/gcc Plattform in Abbildung 5.2 graphisch dargestellt. Auf der Y-Achse ist dabei die Laufzeit in Prozent dargestellt. 100% repräsentieren die Referenz-Laufzeit nach der High-Level CSE Optimierung. Auf der X-Achse sind die Kosten eingetragen, die ein Ausdruck mindestens erreichen muß, damit er eliminiert wird.

Es ist deutlich zu erkennen, daß ab Kosten von 80 Einheiten bei isolierter Betrachtung des Ausdrucks die Laufzeit um 8,8% sinkt. Der gleiche Effekt ergibt sich bei 160 Einheiten, wenn das Produkt von Kosten und Vorkommen benutzt wird. In diesem Modus finden sich bei den Kosteneinheiten 50 und 60 sogar Laufzeitverbesserungen von 10,8%. Dieses Ergebnis liegt weniger als 1% entfernt von der Laufzeit der Feedback-Ergebnisse.

Die Ergebnisse des Intel/icc in Abbildung 5.3 sind nicht ganz so deutlich.

Für die isolierte Betrachtung der Kosten ergeben sich Verbesserungen von 4,5% gegenüber der High-Level CSE Laufzeit. Auch das Produkt von Vorkommen und Kosten kommt nicht über diese 4,5% Laufzeitverbesserung hinaus. Das ist gegenüber dem Feedback-Ergebnis von 10,0% noch nicht optimal.

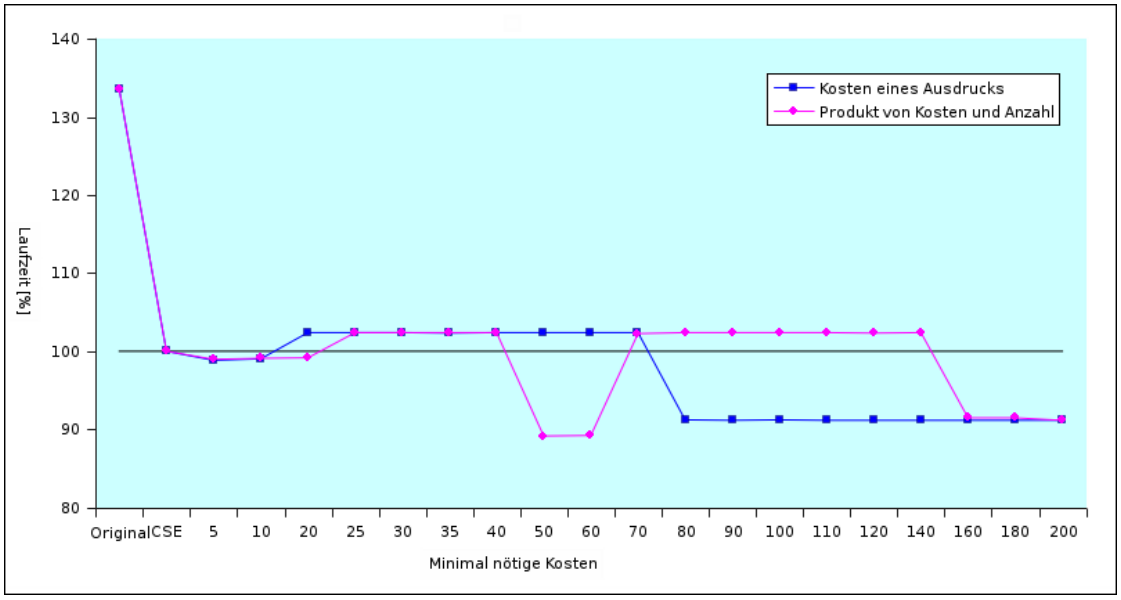

Abbildung 5.2: Kostenergebnisse für den QSDPCM Benchmark auf dem Intel/gcc

5.2.2 Zusammenfassung

Trotz der unvollständigen Messungen für den QSDPCM-Benchmark zeigen sich ermutigende Tendenzen. So zeigt die Feedback-Analyse, daß eine geeignete Auswahl der gefundenen CSEs für drei der vier Plattformen Laufzeitgewinne von 7,9% und 15,0% erzielen kann. Mit der Kosten-Policy können für den Intel/gcc und den Intel/icc 10,8% respektive 4,5% Laufzeitverbesserungen erreicht werden. Eine genauere Analyse der gefundenen Ergebnisse und Anpassungen an die besonderen Gegebenheiten des Benchmarks können die Ergebnisse für den Intel/icc möglicherweise noch verbessern.

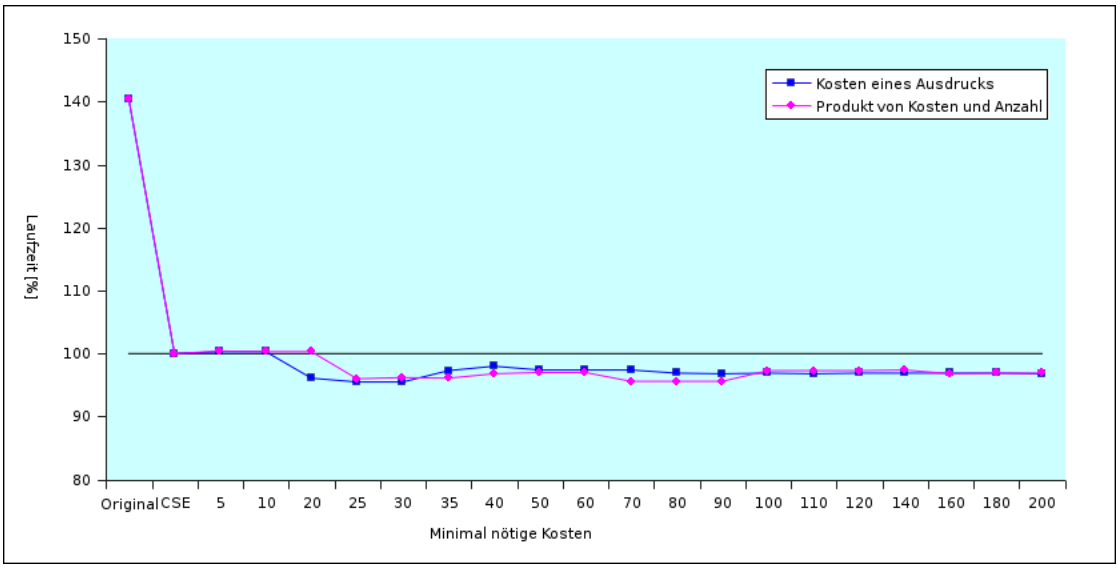

Abbildung 5.3: Kostenergebnisse für den QSDPCM Benchmark auf dem Intel/icc

A Ergebnistabellen

A.1 CAVITY

A.1.1 Liste aller CSEs

```
cse_000_0 = Gauss[1];                    // 6
cse_001_1 = *Gauss;                      // 3
cse_002_24 = 1 <= y;                     // 2
cse_003_25 = y <= 998;                   // 1
cse_004_26 = x % 3;                      // 9
cse_021_43 = cse_002_24 && cse_003_25;   // 2
cse_005_27 = y * 1280;                   // 1
cse_022_44 = cse_005_27 + x;             // 2
cse_006_28 = 2 <= x;                     // 3
cse_007_29 = x <= 1279;                  // 2
cse_008_30 = x - 1;                      // 1
cse_023_45 = cse_008_30 % 3;             // 8
cse_009_31 = 1 <= x;                     // 1
cse_010_32 = x < 1281;                   // 1
cse_011_33 = y - 2;                      // 2
cse_012_34 = cse_011_33 % 3;             // 2
cse_013_35 = (unsigned int)y & 1u;       // 1
cse_014_36 = y - 3;                      // 2
cse_015_37 = y % 3;                      // 2
cse_016_38 = 2 <= y;                     // 2
cse_017_39 = 3 - cse_015_37;             // 1
cse_018_40 = 3 <= x;                     // 2
cse_019_41 = 3 <= y;                     // 2
cse_020_42 = cse_014_36 * 1280;          // 1
cse_024_46 = cse_009_31 && cse_010_32;   // 2
cse_025_47 = cse_012_34 * 3;             // 3
cse_026_48 = (int)cse_013_35 * 1280;     // 2
cse_027_49 = 0 <= cse_014_36;            // 2
cse_028_50 = cse_015_37 * 3;             // 3
cse_029_51 = 1280 - cse_026_48;          // 1
cse_030_52 = cse_017_39 - cse_012_34;    // 1
cse_036_58 = cse_025_47 + cse_023_45;    // 3
cse_037_59 = cse_026_48 + x;             // 2
```

```
cse_038_60 = cse_028_50 + cse_023_45;   // 3
cse_039_61 = cse_029_51 + x;            // 2
cse_040_62 = cse_030_52 * 3;            // 3
cse_047_68 = cse_037_59 - 1;            // 3
cse_049_69 = cse_040_62 + cse_023_45;   // 3
cse_048_6 = cse_039_61 - 1;             // 2
cse_054_7 = cse_048_6 + 2560;           // 2
cse_018_40 = 3 <= x;                    // 2
cse_019_41 = 3 <= y;                    // 2
cse_031_53 = cse_028_50 + 3;            // 1
cse_032_54 = cse_025_47 + 3;            // 1
cse_033_55 = cse_025_47 + cse_004_26;   // 2
cse_034_56 = cse_028_50 + cse_004_26;   // 2
cse_041_63 = cse_031_53 - cse_004_26;   // 1
cse_042_64 = cse_040_62 + 3;            // 1
cse_044_65 = cse_032_54 - cse_004_26;   // 1
cse_045_66 = cse_040_62 + cse_004_26;   // 2
cse_050_70 = cse_041_63 - cse_023_45;   // 1
cse_051_71 = cse_042_64 - cse_004_26;   // 3
cse_053_72 = cse_044_65 - cse_023_45;   // 3
cse_055_73 = cse_051_71 - cse_023_45;   // 3
cse_043_6 = cse_037_59 - 2;             // 1
cse_052_7 = cse_043_6 + 5120;           // 2
cse_020_42 = cse_014_36 * 1280;         // 1
cse_035_57 = cse_020_42 + x;            // 1
cse_046_67 = cse_035_57 - 3;            // 2
```

A.1.2 Einfluß einzelner CSEs

Nur eine CSE eliminiert

In dieser Tabelle ist die angegebene CSE eliminiert, alle anderen werden ignoriert. Die Referenzlaufzeit ist die Laufzeit des unmodifizierten Original Quellcodes.

CSEs	Intel/gcc [%]	Intel/icc [%]	ARM/gcc [%]	Thumb/gcc [%]
cse_000_0	98.68	99.32	98.57	99.91
cse_001_1	99.06	100.34	98.32	99.91
cse_002_24	100.43	94.92	98.30	100.26
cse_003_25	99.43	95.93	98.92	100.10
cse_004_26	88.02	96.84	98.31	77.59
cse_005_27	99.11	99.89	98.69	99.89
cse_006_28	101.08	109.04	99.11	101.13
cse_007_29	101.35	107.68	99.82	100.91
cse_008_30	97.38	109.04	100.67	100.10

CSEs	Intel/gcc [%]	Intel/icc [%]	ARM/gcc [%]	Thumb/gcc [%]
cse_009_31	98.89	101.36	98.34	100.20
cse_010_32	99.97	101.81	98.81	100.55
cse_011_33	99.06	103.73	106.35	99.89
cse_012_34	86.23	95.71	100.73	100.48
cse_013_35	100.30	99.66	100.06	100.38
cse_014_36	99.22	98.53	98.32	99.84
cse_015_37	84.40	104.29	97.81	96.92
cse_016_38	100.49	103.05	105.12	118.74
cse_017_39	90.58	104.52	99.01	104.31
cse_018_40	100.46	101.47	99.91	100.46
cse_019_41	100.59	104.86	99.56	100.48
cse_020_42	99.03	100.11	98.69	99.82
cse_021_43	98.27	95.03	97.45	100.49
cse_022_44	99.22	99.21	98.69	100.60
cse_023_45	83.37	107.12	91.81	69.68
cse_024_46	98.89	99.10	96.71	100.05
cse_025_47	92.42	91.19	102.76	104.75
cse_026_48	97.81	97.97	94.31	99.98
cse_027_49	99.00	102.82	99.43	100.18
cse_028_50	93.31	98.76	100.81	100.34
cse_029_51	99.68	98.98	92.74	100.00
cse_030_52	85.86	92.32	98.81	105.38
cse_031_53	93.90	97.97	100.44	100.93
cse_032_54	91.52	94.35	101.91	105.31
cse_033_55	93.20	89.83	100.57	97.00
cse_034_56	89.61	93.33	101.76	93.08
cse_035_57	99.08	100.45	98.69	99.89
cse_036_58	96.60	94.01	99.52	97.68
cse_037_59	98.16	99.55	94.54	99.81
cse_038_60	94.63	95.59	93.29	97.68
cse_039_61	100.13	100.79	91.90	99.96
cse_040_62	86.53	91.86	97.89	105.10
cse_041_63	90.12	98.31	100.97	92.87
cse_042_64	89.50	95.03	101.34	104.11
cse_043_6	99.33	104.97	96.53	100.10
cse_044_65	87.45	92.77	104.66	93.10
cse_045_66	88.10	91.86	99.62	96.74
cse_046_67	98.89	102.94	98.69	99.89
cse_047_68	99.14	103.50	92.56	103.34
cse_048_6	98.14	101.02	96.92	99.90
cse_049_69	91.28	90.62	93.11	105.53

CSEs	Intel/gcc [%]	Intel/icc [%]	ARM/gcc [%]	Thumb/gcc [%]
cse_050_70	84.91	94.92	99.19	85.24
cse_051_71	85.07	88.93	99.19	96.29
cse_052_7	99.11	98.64	97.43	100.03
cse_053_72	79.30	87.80	99.58	81.71
cse_054_7	97.60	100.79	96.66	99.90
cse_055_73	80.30	84.41	93.54	89.48

Alle bis auf eine CSE eliminiert

In dieser Tabelle werden alle, bis auf die angegebene CSE eliminiert. Die Referenzlaufzeit ist die Laufzeit des High-Level CSE Quellcodes.

CSEs	Intel/gcc [%]	Intel/icc [%]	ARM/gcc [%]	Thumb/gcc [%]
cse_000_0	99.85	95.97	98.84	100.30
cse_001_1	98.84	99.62	99.44	100.30
cse_002_24	96.22	97.74	97.72	99.38
cse_003_25	98.79	100.25	100.00	100.00
cse_004_26	105.24	97.11	101.15	115.25
cse_005_27	98.84	100.25	100.00	100.00
cse_006_28	99.90	96.86	98.24	99.18
cse_007_29	99.45	96.10	98.85	99.30
cse_008_30	98.84	100.25	100.00	100.00
cse_009_31	99.29	100.25	100.00	100.00
cse_010_32	98.84	100.25	100.00	100.00
cse_011_33	98.79	100.25	100.00	100.00
cse_012_34	98.49	97.48	98.53	100.48
cse_013_35	98.59	100.25	100.00	100.00
cse_014_36	98.59	100.13	99.83	100.00
cse_015_37	98.84	97.74	98.09	91.47
cse_016_38	95.76	97.48	98.41	99.23
cse_017_39	98.64	100.38	100.00	100.00
cse_018_40	94.86	97.23	98.60	99.10
cse_019_41	94.30	99.50	98.11	99.33
cse_020_42	98.64	100.00	100.00	100.00
cse_021_43	98.54	98.62	100.47	100.24
cse_022_44	98.69	100.38	100.00	99.77
cse_023_45	107.11	98.11	101.39	117.41
cse_024_46	100.81	98.11	100.50	100.79
cse_025_47	98.08	96.73	100.16	100.05
cse_026_48	97.68	97.23	100.00	99.98

CSEs	Intel/gcc [%]	Intel/icc [%]	ARM/gcc [%]	Thumb/gcc [%]
cse_027_49	98.23	96.98	99.15	99.39
cse_028_50	98.39	99.87	101.04	100.05
cse_029_51	98.59	100.25	100.00	100.00
cse_030_52	98.59	100.25	100.00	100.00
cse_031_53	98.59	100.25	100.00	100.00
cse_032_54	98.64	100.38	100.00	100.00
cse_033_55	98.89	99.25	99.73	99.20
cse_034_56	97.28	97.61	99.90	99.47
cse_035_57	98.64	100.50	100.00	100.00
cse_036_58	100.96	97.74	100.20	100.25
cse_037_59	98.13	100.50	100.00	100.54
cse_038_60	99.85	99.62	100.13	100.23
cse_039_61	98.99	94.97	99.67	100.35
cse_040_62	97.33	96.48	99.45	99.92
cse_041_63	98.64	100.25	100.00	100.00
cse_042_64	98.69	100.25	100.00	100.00
cse_043_6	98.69	100.13	100.00	100.00
cse_044_65	98.64	100.38	100.00	100.00
cse_045_66	98.89	99.75	100.03	100.45
cse_046_67	98.54	100.50	100.00	100.00
cse_047_68	97.68	100.25	99.38	100.15
cse_048_6	98.34	100.13	99.82	100.10
cse_049_69	98.74	99.50	100.60	100.37
cse_050_70	97.48	100.88	99.66	100.69
cse_051_71	98.69	100.38	100.00	100.00
cse_052_7	98.54	100.75	100.39	100.19
cse_053_72	97.48	98.62	100.42	100.65
cse_054_7	98.39	98.87	100.38	100.22
cse_055_73	98.69	99.37	100.81	100.91

A.1.3 UseCount Policy

In den folgenden Tabellen wird eine CSE eliminiert, wenn sie mindestens die angegebene Zahl an Vorkommen im Quellcode hat. Referenzlaufzeit ist die High-Level CSE Laufzeit.

MinUses	CSEs	Uses	Laufzeit [s]	Prozent von CSE
CSE	38	105	19.57	100.00
3	15	65	18.6	95.04
4-6	3	44	26.48	135.31
7-8	2	38	26.68	136.33
9	1	18	32.6	166.58

Tabelle A.3: Ergebnisse der UseCount Policy für Intel/gcc

MinUses	CSEs	Uses	Laufzeit [s]	Prozent von CSE
CSE	38	105	7.95	100.00
3	15	65	7.14	89.81
4-6	3	44	8.83	111.07
7-8	2	38	8.78	110.44
9	1	18	8.57	107.80

Tabelle A.4: Ergebnisse der UseCount Policy für Intel/icc

MinUses	CSEs	Uses	Laufzeit [cyc]	Prozent von CSE
CSE	38	105	209848	100.00
3	15	65	190514	90.79
4-6	3	44	214725	102.32
7-8	2	38	213489	101.74
9	1	18	240058	114.40

Tabelle A.5: Ergebnisse der UseCount Policy für ARM/gcc

MinUses	CSEs	Uses	Laufzeit [cyc]	Prozent von CSE
2	38	105	306428	100.00
3	15	65	243591	79.49
4-6	3	44	317438	103.59
7-8	2	38	318288	103.87
9	1	18	520742	169.94

Tabelle A.6: Ergebnisse der UseCount Policy für Thumb/gcc

A.1.4 Lifetime Analyse

In den folgenden Tabellen wird eine CSE eliminiert, wenn beim entsprechenden Vorkommen höchstens die angegebene Zahl an anderen Variablen lebendig sind. Referenzlaufzeit ist die High-Level CSE Laufzeit.

MaxLifeVars	Laufzeit [s]	CSEs	Uses	Prozent von CSE
Original	36.55	0	0	186.77
CSE	19.57	38	105	100.00
5	36.76	4	8	187.84
10	31.97	9	19	163.36
15	22.68	19	43	115.89
20	19.05	32	85	97.34
25	19.03	36	99	97.24
30	19.83	38	105	101.33

Tabelle A.7: Lifetime Ergebnisse für Intel/gcc

MaxLifeVars	Laufzeit [s]	CSEs	Uses	Prozent von CSE
Original	8.82	0	0	110.94
CSE	7.95	38	105	100.00
5	9.36	4	8	117.59
10	7.8	9	19	97.99
15	7.64	19	43	95.98
20	7.31	32	85	91.83
25	7.82	36	99	98.24
30	8.07	38	105	101.38

Tabelle A.8: Lifetime Ergebnisse für Intel/icc

MaxLifeVars	Laufzeit [cyc]	CSEs	Uses	Prozent von CSE
Original	244177	0	0	116.36
CSE	209848	38	105	100.00
5	239544	4	8	114.15
10	209542	9	19	99.85
15	205376	19	43	97.87
20	210464	32	85	100.29
25	208279	36	99	99.25
30	209848	38	105	100.00

Tabelle A.9: Lifetime Ergebnisse für ARM/gcc

MaxLifeVars	Laufzeit [cyc]	CSEs	Uses	Prozent von CSE
Original	675836	0	0	220.55
CSE	306428	38	105	100.00
5	671538	4	8	219.15
10	664927	9	19	216.99
15	433437	19	43	141.45
20	322502	32	85	105.25
25	304741	36	99	99.45
30	306428	38	105	100.00

Tabelle A.10: Lifetime Ergebnisse für Thumb/gcc

A.1.5 Kosten

In den folgenden Tabellen wird eine CSE eliminiert, wenn sie mindestens die angegebenen Kosten erreicht. Referenzlaufzeit ist die High-Level CSE Laufzeit.

Isolierte Betrachtung der Kosten

MinCost	Laufzeit [s]	CSEs	Uses	Prozent von CSE
Original	36.55	0	0	186.77
CSE	19.57	38	105	100.00
5	19.62	38	105	100.26
10	19.67	39	108	100.51
20	16.84	23	78	86.05
25	18.26	14	76	93.31
30	17.33	12	71	88.55
35	20.25	14	66	103.47
40	19.9	14	66	101.69
50	21.76	12	34	111.19
60	23.32	10	68	119.16
70	25.01	8	25	127.80
80	25	8	25	127.75
90	25.2	7	23	128.77
100	25.84	5	18	132.04
110	24.09	5	14	123.10
120	24.04	5	14	122.84
140	24.33	5	14	124.32
160	25.5	3	9	130.30
180	29.76	1	3	152.07
200	29.74	1	3	151.97

Tabelle A.11: Isolierte Kosten für Intel/gcc, TestsInIfPenalty=50%

Wird stattdessen eine TestsInIf Penalty von 80% gewählt, so ändern sich die ersten Einträge wie folgt (alle übrigen bleiben gleich):

MinCost	Laufzeit [s]	CSEs	Uses	Prozent von CSE
5	19.68	39	108	100.56
10	16.83	29	87	86.00
20	16.83	23	78	86.00
25	17.35	12	71	88.66

Tabelle A.12: Isolierte Kosten für Intel/gcc, TestsInIfPenalty=80%

Distanz	Laufzeit [s]	CSEs	Uses	Prozent von CSE
Original	8.82	0	0	110.94
CSE	7.95	38	105	100.00
5	7.97	38	105	100.25
10	7.95	38	105	100.00
20	7.65	23	76	96.23
25	7.54	16	62	94.84
30	7.31	14	43	91.95
35	7.1	12	39	89.31
40	7.09	12	36	89.18
50	7.27	10	30	91.45
60	7.18	8	25	90.31
70	7.26	5	18	91.32
80	7.24	5	14	91.07
90	7.24	5	14	91.07
100	7.23	5	14	90.94
110	7.15	3	9	89.94
120	7.39	1	3	92.96
140	7.37	1	3	92.70
160	8.83	0	0	111.07
180	8.81	0	0	110.82
200	8.82	0	0	110.94

Tabelle A.13: Isolierte Kosten für Intel/icc, TestsInIfPenalty=15%

Wird stattdessen eine TestsInIf Penalty von 80% gewählt, so ändern sich die ersten Einträge wie folgt (alle übrigen bleiben gleich):

MinCost	Laufzeit [s]	CSEs	Uses	Prozent von CSE
5	7.06	31	91	88.81
10	6.88	29	87	86.54
20	6.8	20	70	85.53
25	6.79	14	58	85.41
30	7.1	12	39	89.31

Tabelle A.14: Isolierte Kosten für Intel/icc, TestsInIfPenalty=80%

Distanz	Laufzeit [cyc]	CSEs	Uses	Prozent von CSE
Original	244177	0	0	116.36
cse	209848	38	105	100.00
5	209848	38	105	100.00
10	209874	38	101	100.01
20	203132	19	70	96.80
25	200661	15	43	95.62
30	206422	12	35	98.37
35	212103	11	32	101.07
40	214847	9	27	102.38
50	222132	7	23	105.85
60	232649	5	18	110.87
70	226085	5	14	107.74
80	226085	5	14	107.74
90	226085	5	14	107.74
100	233528	3	9	111.28
110	228415	1	3	108.85
120	228415	1	3	108.85
140	240968	0	0	114.83
160	240968	0	0	114.83
180	240968	0	0	114.83
200	240968	0	0	114.83

Tabelle A.15: Isolierte Kosten für ARM/gcc, TestsInIfPenalty=15%

Wird stattdessen eine TestsInIf Penalty von 80% gewählt, so ändern sich die ersten Einträge wie folgt (alle übrigen bleiben gleich):

MinCost	Laufzeit [cyc]	CSEs	Uses	Prozent von CSE
5	194135	31	91	92.51
10	197728	26	83	94.22
20	201043	17	66	95.80
25	201694	13	39	96.11

Tabelle A.16: Isolierte Kosten für ARM/gcc, TestsInIfPenalty=80%

MinCost	Laufzeit [cyc]	CSEs	Uses	Prozent von CSE
Original	675836	0	0	220.55
cse	306428	38	105	100.00
5	306428	38	105	100.00
10	306428	38	105	100.00
20	297849	24	85	97.20
25	299389	13	70	97.70
30	298206	11	66	97.32
35	301356	13	66	98.34
40	301356	13	66	98.34
50	296649	7	70	96.81
60	297106	6	67	96.96
70	297577	4	70	97.11
80	297577	4	70	97.11
90	297577	4	70	97.11
100	317901	10	45	103.74
110	370208	9	28	120.81
120	425349	8	25	138.81
140	425349	8	25	138.81
160	425349	8	25	138.81
180	425349	8	25	138.81
200	425349	8	25	138.81

Tabelle A.17: Isolierte Kosten für Thumb/gcc, TestsInIfPenalty=20%

Wird stattdessen eine TestsInIf Penalty von 80% gewählt, so ändern sich die ersten Einträge wie folgt (alle übrigen bleiben gleich):

MinCost	Laufzeit [cyc]	CSEs	Uses	Prozent von CSE
5	291965	31	91	95.28
10	239990	29	87	78.32
20	295235	21	79	96.35
25	298206	11	66	97.32

Tabelle A.18: Isolierte Kosten für Thumb/gcc, TestsInIfPenalty=80%

Produkt der Kosten des Ausdrucks mit der Zahl der Vorkommen

Alle folgenden Tests sind mit TestsInIfPenalty=80% durchgeführt.

MinCost	Laufzeit [s]	CSEs	Uses	Prozent von CSE
Original	36.55	0	0	186.77
cse	19.57	38	105	100.00
5	19.68	38	105	100.56
10	16.92	29	87	86.46
20	16.91	29	87	86.41
25	17.85	22	80	91.21
30	17.8	22	80	90.96
35	16.52	21	79	84.41
40	16.62	21	79	84.93
50	17.27	15	73	88.25
60	17.28	15	73	88.30
70	17.23	13	71	88.04
80	17.24	13	71	88.09
90	17.17	12	68	87.74
100	18.48	9	76	94.43
100	18.41	9	76	94.07
120	18.25	8	73	93.25
140	19.75	6	54	100.92
160	19.51	5	53	99.69
180	21.65	6	45	110.63
200	21.15	5	44	108.07
240	21.84	4	45	111.60
280	23.16	3	46	118.34
320	23.35	4	30	119.32

Tabelle A.19: Produkt von Kosten und Vorkommen für Intel/gcc

Distanz	Laufzeit [s]	CSEs	Uses	Prozent von CSE
Original	8.82	0	0	110.94
CSE	7.95	38	105	100.00
5	7.06	31	91	88.81
10	6.88	29	87	86.54
20	6.89	29	87	86.67
25	7.07	22	80	88.93
30	7.07	22	80	88.93
35	6.93	19	79	87.17
40	6.94	19	79	87.30
50	7.11	16	76	89.43
60	6.94	13	73	87.30
70	7.14	12	70	89.81
80	7.02	13	70	88.30
90	6.84	12	63	86.04
100	7.27	10	68	91.45
110	7.25	10	68	91.19
120	7.1	9	56	89.31
140	7.28	6	45	91.57
160	7.04	4	30	88.55
180	7.05	4	30	88.68
200	7.05	4	30	88.68
240	7.02	4	12	88.30
280	7.01	4	12	88.18
320	7.14	3	9	89.81

Tabelle A.20: Produkt von Kosten und Vorkommen für Intel/icc

Distanz	Laufzeit [cyc]	CSEs	Uses	Prozent von CSE
Original	244177	0	0	116.36
CSE	209848	38	105	100.00
5	194135	31	91	92.51
10	195355	29	97	93.09
20	195355	28	85	93.09
25	193624	22	79	92.27
30	193611	21	76	92.26
35	194708	18	77	92.79
40	197202	16	74	93.97
50	189916	14	73	90.50
60	190960	14	72	91.00
70	190687	13	64	90.87
80	191790	11	59	91.39
90	203853	10	56	97.14
100	215168	7	51	102.54
110	215168	7	51	102.54
120	215168	7	51	102.54
140	225583	4	30	107.50
160	225583	4	30	107.50
180	225583	4	30	107.50
200	232649	5	18	110.87
240	230257	4	12	109.73
280	233528	3	9	111.28
320	228415	1	3	108.85

Tabelle A.21: Produkt von Kosten und Vorkommen für ARM/gcc

MinCost	Laufzeit [cyc]	CSEs	Uses	Prozent von CSE
Original	675836	0	0	220.55
CSE	306428	38	105	100.00
5	291965	31	91	95.28
10	239826	29	87	78.27
20	239826	29	87	78.27
25	240722	22	80	78.56
30	240722	22	80	78.56
35	293915	21	79	95.92
40	293915	21	79	95.92
50	294162	15	73	96.00
60	294162	15	73	96.00
70	301400	13	72	98.36
80	301400	13	72	98.36
90	298740	12	69	97.49
100	303840	12	70	99.16
110	303840	12	70	99.16
120	302913	11	67	98.85
140	297419	5	65	97.06
160	297419	5	65	97.06
180	297419	5	65	97.06
200	273796	5	63	89.35
240	242686	5	62	79.20
280	242686	5	62	79.20
320	270731	6	54	88.35

Tabelle A.22: Produkt von Kosten und Vorkommen für Thumb/gcc

Zahl der Vorkommen und Kosten des Ausdrucks als unabhängige Parameter

MinCost	Laufzeit [s]	CSEs	Uses	Prozent von CSE
Original	36.55	0	0	186.77
CSE	19.57	38	105	100.00
nur MinUse=3	18.57	15	65	94.89
5	18.57	15	65	94.89
10	17.85	14	62	91.21
20	17.34	11	65	88.61
25	17.51	9	56	89.47
30	17.51	9	56	89.47
35	19.29	8	44	98.57
40	19.29	8	44	98.57
50	22.94	8	26	117.22
60-90	25.38	6	21	129.69
100	25.54	5	18	130.51
110-160	25.07	4	12	128.10
180-200	29.76	1	3	152.07

Tabelle A.23: Kosten und Vorkommen als unabhängige Parameter für Intel/gcc

Distanz	Laufzeit [s]	CSEs	Uses	Prozent von CSE
Original	8.82	0	0	110.94
CSE	7.95	38	105	100.00
nur MinUse=3	7.16	15	65	90.06
5	6.95	14	62	87.42
10	6.96	14	62	87.55
20	7.04	10	53	88.55
25	6.96	8	44	87.55
30	7.07	9	29	88.93
35	7.06	9	29	88.81
40	7.22	7	24	90.82
50	7.17	6	21	90.19
60	7.17	6	21	90.19
70	7.27	5	18	91.45
80-100	7.03	4	12	88.43
110	7.15	3	9	89.94
120-140	7.4	1	3	93.08
160-200	8.83	0	0	111.07

Tabelle A.24: Kosten und Vorkommen als unabhängige Parameter für Intel/icc

Distanz	Laufzeit [cyc]	CSEs	Uses	Prozent von CSE
Original	244177	0	0	116.36
CSE	209848	38	105	100.00
nur MinUse=3	190514	15	65	90.79
5	187955	14	62	89.57
10	187955	14	62	89.57
20	201907	10	53	96.22
25	201865	9	29	96.20
30	214987	8	26	102.45
35	224596	6	21	107.03
40	224596	6	21	107.03
50	224596	6	21	107.03
60	232649	5	18	110.87
70	230257	4	12	109.73
80	230257	4	12	109.73
90	230257	4	12	109.73
100	233528	3	9	111.28
110	228415	1	3	108.85
120	228415	1	3	108.85
140	240968	0	0	114.83
160	240968	0	0	114.83
180	240968	0	0	114.83
200	240968	0	0	114.83

Tabelle A.25: Kosten und Vorkommen als unabhängige Parameter für ARM/gcc

MinCost	Laufzeit [cyc]	CSEs	Uses	Prozent von CSE
Original	675836	0	0	220.55
CSE	306428	38	105	100.00
nur MinUse=3	243651	15	65	79.51
5	246237	12	53	80.36
10	246237	12	53	80.36
20-30	245725	9	56	80.19
35	246411	6	65	80.41
40	246411	6	65	80.41
50-90	242686	5	62	79.20
100	345959	7	36	112.90
110	391822	8	26	127.87
120-200	471999	6	21	154.03

Tabelle A.26: Kosten und Vorkommen als unabhängige Parameter für Thumb/gcc

A.1.6 Abstand

In den folgenden Tabellen wird eine CSE aufgeteilt, wenn der angegebene Abstand überschritten wird. Referenzlaufzeit ist die High-Level CSE Laufzeit.

Distanz	Laufzeit [s]	Prozent von CSE
Original	36.55s	–
Normal CSE	19.57s	100
50	35.04s	179.05
100	31.92s	163.11
150	26.36s	134.70
200	23.66s	120.90
250	21.96s	112.21
300	21.97s	112.26
350	18.26s	93.31
400	19.48s	99.54
450	19.49s	99.59
500	19.49s	99.59

Tabelle A.27: Abstandsergebnisse für Intel/gcc

Distanz	Laufzeit [s]	Prozent von CSE
Original	8.83	110.93
Normal CSE	7.96	100.00
50	6.93	87.06
100	7.19	90.33
150	7.21	90.58
200	7.38	92.71
250	7.39	92.84
300	7.37	92.59
350	7.39	92.84
400	7.65	96.11
450	7.66	96.23
500	7.66	96.23

Tabelle A.28: Abstandsergebnisse für Intel/icc

Distanz	Laufzeit [cyc]	Prozent von CSE
Original	244177	-
Normal CSE	209848	100
50	250389	119.32
100	238114	113.47
150	235780	112.36
200	219539	104.62
250	211273	100.68
300	211273	100.68
350	205430	97.89
400	211071	100.58
450	211071	100.58
500	211071	100.58

Tabelle A.29: Abstandsergebnisse für ARM/gcc

Distanz	Laufzeit [cyc]	Prozent von CSE
Original	675836	-
Normal CSE	306428	100
50	674317	220.06
100	556326	181.55
150	594541	194.02
200	458211	149.53
250	401986	131.18
300	401986	131.18
350	303333	98.99
400	307626	100.39
450	307626	100.39
500	307626	100.39

Tabelle A.30: Abstandsergebnisse für Thumb/gcc

A.1.7 Neuronale Netze

Alle folgenden Meßdaten wurden mit Connectionrate=1.0 und Lernrate=0.1 gemacht.

Methode 1

In den folgenden Tabellen wird eine CSE eliminiert, wenn das neuronale Netz für die gefundene CSE die angegebene Maximalprognose unterschreitet. Referenzlaufzeit ist die High-Level CSE Laufzeit.

Intel/gcc

Parameter	Laufzeit [s]	CSEs	Nutzungen	Prozent von CSE
Original	36,55	–	–	186.92
Normal CSE	19,57	38	105	100
Feedback	16,78	9	66	85,74
98	36.74	0	0	187.74
98.5	33.24	1	15	169.85
99	26.64	2	36	136.13
99.5	25.04	8	40	127.95
100	19.63	11	56	100.31
100.5	19.1	11	56	97.60
101	16.99	12	66	86.82
101.5	16.91	18	76	86.41
102	19.18	35	102	98.01

Tabelle A.31: TestsInIf Penality=50%, 10,000 Iteratonen, 15 Neuronen

Parameter	Laufzeit [s]	CSEs	Nutzungen	Prozent von CSE
Original	36,55	–	–	186.92
Normal CSE	19,57	38	105	100
Feedback	16,78	9	66	85,74
98	33.26	0	0	169.95
98.5	26.65	2	35	136.18
99	26.12	3	40	133.47
99.5	18.31	10	57	93.56
100	16.99	12	66	86.82
100.5	16.82	13	69	85.95
101	16.91	19	78	86.41
101.5	17.37	20	79	88.76
102	18.07	28	88	92.34

Tabelle A.32: TestsInIf Penality=80%, 10,000 Iteratonen, 15 Neuronen

Intel/icc

Parameter	Laufzeit [s]	CSEs	Nutzungen	Prozent von CSE
Original	8,82	–	–	–
Normal CSE	7,95	38	105	100
feedback	7.02	4	11	88.30
98	8.85	0	0	111.32
98.5	8.85	0	0	111.32
99	8.85	0	0	111.32
99.5	8.79	2	36	110.57
100	7.37	4	45	92.70
100.5	7.06	13	71	88.81
101	7.98	38	105	100.38
101.5	7.98	38	105	100.38
102	7.96	38	105	100.13

Tabelle A.33: TestsInIf Penalty=15%, 50,000 Iteratonen, 35 Neuronen

Parameter	Laufzeit [s]	CSEs	Nutzungen	Prozent von CSE
Original	8,82	–	–	–
Normal CSE	7,95	38	105	100
feedback	7.02	4	11	88.30
98	8.85	0	0	111.32
98.5	8.85	0	0	111.32
99	8.85	0	0	111.32
99.5	8.79	2	36	110.57
100	7.48	3	46	94.09
100.5	7.03	13	71	88.43
101	7.98	38	105	100.38
101.5	7.98	38	105	100.38
102	7.96	38	105	100.13

Tabelle A.34: TestsInIf Penalty=80%, 50,000 Iteratonen, 35 Neuronen

ARM/gcc

Parameter	Laufzeit [cyc]	CSEs	Nutzungen	Prozent von CSE
Original	244177	–	–	–
Normale CSE	209848	38	105	100
Feedback CSE	179339	9	53	85.46
98	240968	0	0	114.83
98.5	240968	0	0	114.83
99	240968	0	0	114.83
99.5	206058	9	57	98.19
100	204316	13	67	97.36
100.5	194926	24	83	92.89
101	198104	32	97	94.40
101.5	205947	35	99	98.14
102	209848	38	105	100.00

Tabelle A.35: TestsInIf Penalty=15%, 50,000 Iteratonen, 15 Neuronen

Parameter	Laufzeit [cyc]	CSEs	Nutzungen	Prozent von CSE
Original	244177	–	–	–
Normale CSE	209848	38	105	100
Feedback CSE	179339	9	53	85.46
98	240968	0	0	114.83
98.5	240968	0	0	114.83
99	240968	0	0	114.83
99.5	207025	9	56	98.65
100	190124	22	78	90.60
100.5	193915	26	84	92.41
101	195355	33	95	93.09
101.5	199435	33	95	95.04
102	202947	36	101	96.71

Tabelle A.36: TestsInIf Penalty=80%, 50,000 Iteratonen, 15 Neuronen

Thumb/gcc

Parameter	Laufzeit [cyc]	CSEs	Nutzungen	Prozent von CSE
Original	675836	–	–	–
CSE	306428	38	105	100
Feedback	243030	6	63	79.31
98	670447	0	0	218.79
98.5	670447	0	0	218.79
99	670447	0	0	218.79
99.5	669763	4	14	218.57
100	271171	17	81	88.49
100.5	243759	26	87	79.55
101	300469	33	99	98.06
101.5	303671	36	101	99.10
102	306428	38	105	100.00

Tabelle A.37: TestsInIf Penalty=20%, 50,000 Iteratonen, 35 Neuronen

Parameter	Laufzeit [cyc]	CSEs	Nutzungen	Prozent von CSE
Original	675836	–	–	–
CSE	306428	38	105	100
Feedback	243030	6	63	79.31
98	670447	0	0	218.79
98.5	670447	0	0	218.79
99	670447	0	0	218.79
99.5	670580	2	8	218.84
100	277389	15	72	90.52
100.5	243546	25	85	79.48
101	242710	27	87	79.21
101.5	303505	33	97	99.05
102	306428	38	105	100.00

Tabelle A.38: TestsInIf Penalty=80%, 50,000 Iteratonen, 35 Neuronen

Methode 2

In den folgenden Tabellen wird eine CSE eliminiert, wenn das neuronale Netz für die gefundene CSE die angegebene Minimalgüte überschreitet. Referenzlaufzeit ist die High-Level CSE Laufzeit.

Intel/gcc

Parameter	Laufzeit [s]	CSEs	Nutzungen	Prozent von CSE
Original	36.55	–	–	–
Normale CSE	19.57	38	105	100
Feedback CSE	16.78	9	66	85.74
0.10	16.69	27	85	85.28
0.20	16.68	27	85	85.23
0.30	16.6	27	85	84.82
0.40	17.04	23	83	87.07
0.50	17.26	22	82	88.20
0.60	27.1	4	53	138.48
0.70	36.96	2	9	188.86
0.80	36.59	1	4	186.97
0.90	36.56	1	4	186.82

Tabelle A.39: TestsInIf Penalty=50%, 50,000 Iteratonen, 35 Neuronen

Parameter	Laufzeit [s]	CSEs	Nutzungen	Prozent von CSE
Original	36.55	–	–	–
Normale CSE	19.57	38	105	100
Feedback	16.78	9	66	85.74
0.1	16.9	29	87	86.36
0.2	16.91	29	87	86.41
0.3	16.91	29	87	86.41
0.4	16.92	29	87	86.46
0.5	16.92	29	87	86.46
0.6	17.28	27	90	88.30
0.7	17.67	20	78	90.29
0.8	24.09	4	55	123.10
0.9	26.71	3	54	136.48

Tabelle A.40: TestsInIf Penalty=80%, 10,000 Iteratonen, 15 Neuronen

Intel/icc

Parameter	Laufzeit [s]	CSEs	Nutzungen	Prozent von CSE
Original	8,82	–	–	–
Normal CSE	7,95	38	105	100
Feedback	7.02	4	11	88.30
> 0.10	8.05	38	105	101.26
> 0.20	7.56	21	94	95.09
> 0.30	7.04	16	87	88.55
> 0.40	7.01	16	87	88.18
> 0.50	8.57	1	18	107.80
> 0.60	8.72	1	17	109.69
> 0.70	8.71	1	17	109.56
> 0.80	8.73	1	17	109.81
> 0.90	8.83	0	0	111.07

Tabelle A.41: TestsInIf Penalty=15%, 50,000 Iteratonen, 25 Neuronen

Parameter	Laufzeit [s]	CSEs	Nutzungen	Prozent von CSE
Original	8.82	–	–	–
Normal CSE	7.95	38	105	100
feedback	7.02	4	11	88.30
0.1	7.13	28	86	89.69
0.2	6.96	26	81	87.55
0.3	7.05	24	80	88.68
0.4	7.1	24	80	89.31
0.5	9.24	5	62	116.23
0.6	9.21	3	36	115.85
0.7	8.78	2	34	110.44
0.8	8.89	0	0	111.82
0.9	8.84	0	0	111.19

Tabelle A.42: TestsInIf Penalty=80%, 50,000 Iteratonen, 35 Neuronen

ARM/gcc

Parameter	Laufzeit [cyc]	CSEs	Nutzungen	Prozent von CSE
Original	244177	–	–	–
Normal CSE	209848	38	105	100
Feedback	179339	9	53	85.46
> 0.10	209848	38	105	100.00
> 0.20	197501	32	97	94.12
> 0.30	196090	25	87	93.44
> 0.40	191522	25	84	91.27
> 0.50	191522	25	84	91.27
> 0.60	191522	25	84	91.27
> 0.70	244320	1	9	116.43
> 0.80	244320	1	9	116.43
> 0.90	244320	1	9	116.43

Tabelle A.43: TestsInIf Penalty=15%, 50,000 Iteratonen, 35 Neuronen

Parameter	Laufzeit [cyc]	CSEs	Nutzungen	Prozent von CSE
Original	244177	–	–	–
Normal CSE	209848	38	105	100
Feedback	179339	9	53	85.46
0.1	202848	33	97	96.66
0.2	202848	33	97	96.66
0.3	198846	24	84	94.76
0.4	198846	24	84	94.76
0.5	198846	24	84	94.76
0.6	198177	15	73	94.44
0.7	230819	5	71	109.99
0.8	230819	5	71	109.99
0.9	240058	1	18	114.40

Tabelle A.44: TestsInIf Penalty=80%, 20,000 Iteratonen, 35 Neuronen

Thumb/gcc

Parameter	Laufzeit [cyc]	CSEs	Nutzungen	Prozent von CSE
Original	675836	–	–	–
CSE	306428	38	105	100
Feedback	243030	6	63	79.31
> 0.10	306428	38	105	100.00
> 0.20	306428	38	105	100.00
> 0.30	306428	38	105	100.00
> 0.40	297610	25	82	97.12
> 0.50	295291	23	75	96.37
> 0.60	675836	0	0	220.55
> 0.70	675836	0	0	220.55
> 0.80	675836	0	0	220.55
> 0.90	675836	0	0	220.55

Tabelle A.45: TestsInIf Penalty=20%, 20,000 Iteratonen, 15 Neuronen

Parameter	Laufzeit	CSEs	Nutzungen	Prozent von CSE
Original	675836	–	–	–
CSE	306428	38	105	100
Feedback	243030	6	63	79.31
0.1	239826	29	87	78.27
0.2	239826	29	87	78.27
0.3	239826	29	87	78.27
0.4	239826	29	87	78.27
0.5	675836	0	0	220.55
0.6	675836	0	0	220.55
0.7	675836	0	0	220.55
0.8	675836	0	0	220.55
0.9	675836	0	0	220.55

Tabelle A.46: TestsInIf Penalty=80%, 20,000 Iteratonen, 15 Neuronen

A.2 QSDPCM

Alle Messungen beim QSDPCM Benchmark wurden mit `-O3` durchgeführt, da hier das automatische Funktionsinlining ausgenutzt werden soll.

A.2.1 Feedback Ergebnisse

Die folgende Tabelle gibt die Ergebnisse der Feedback-Optimierung an.

Plattform	Original	High-Level CSE	Feedback CSE
Intel/gcc	20.67	15.3	13.76
Intel/icc	14.01	9.98	9.19
ARM/gcc	5,125,580	3,378,981	3,373,779
Thumb/gcc	3,692,494	918,193	780,025

A.2.2 Kostenergebnisse

In den folgenden Tabellen wird eine CSE eliminiert, wenn sie mindestens die angegebenen Kosten erreicht. Referenzlaufzeit ist die High-Level CSE Laufzeit.

Isolierte Kostenbetrachtung

Intel/gcc, considerUseCount=0, TestsInIfPenality=50%

MinCost	Laufzeit [s]	CSEs	Uses	Prozent von CSE
Original	20.67	0	0	133.53
CSE	15.48	65	253	100.00
5	15.33	64	253	99.03
10	15.32	64	253	98.97
20	15.83	41	183	102.26
25	15.85	8	20	102.39
30	15.84	7	18	102.33
35	15.84	3	6	102.33
40	15.85	3	6	102.39
50	15.84	3	6	102.33
60	15.85	3	6	102.39
70	15.85	3	6	102.39
80	14.11	2	4	91.15
90	14.11	2	4	91.15
100	14.11	2	4	91.15
110	14.11	2	4	91.15
120	14.11	2	4	91.15
140	14.11	2	4	91.15
160	14.11	2	4	91.15
180	14.11	2	4	91.15
200	14.11	2	4	91.15

Wird stattdessen eine TestsInIf Penalty von 80% gewählt, so ändern sich die ersten Einträge wie folgt (alle übrigen bleiben gleich):

MinCost	Laufzeit [s]	CSEs	Uses	Prozent von CSE
5	15.3	64	253	98.84
10	15.33	58	241	99.03
20	15.85	40	181	102.39
25	15.85	7	18	102.39

Intel/icc, considerUseCount=0, TestsInIfPenality=15%

MinCost	Laufzeit [s]	CSEs	Uses	Prozent von CSE
Original	14.01	0	0	140.38
CSE	9.98	65	253	100.00
5	10.01	64	253	100.30
10	10	64	253	100.20
20	9.58	41	183	95.99
25	9.53	8	20	95.49
30	9.54	8	20	95.59
35	9.71	3	6	97.29
40	9.72	3	6	97.39
50	9.72	3	6	97.39
60	9.72	3	6	97.39
70	9.72	3	6	97.39
80	9.67	2	4	96.89
90	9.73	2	4	97.49
100	9.67	2	4	96.89
110	9.66	2	4	96.79
120	9.67	2	4	96.89
140	9.67	2	4	96.89
160	9.66	2	4	96.79
180	9.67	2	4	96.89
200	9.67	2	4	96.89

Wird stattdessen eine TestsInIf Penalty von 80% gewählt, so ändern sich die ersten Einträge wie folgt (alle übrigen bleiben gleich):

MinCost	Laufzeit [s]	CSEs	Uses	Prozent von CSE
5	10.01	58	241	100.30
10	10.01	57	239	100.30
20	9.59	40	181	96.09
25	9.53	7	18	95.49
30	9.53	7	18	95.49

Produkt von Kosten und Vorkommen

Intel/gcc, considerUseCount=1, TestsInIfPenality=50%

MinCost	Laufzeit (in sec)	CSEs	Uses	Prozent von CSE
Original	20.67	0	0	133.53
CSE	15.48	65	253	100.00
5	15.28	64	253	98.71
10	15.32	64	253	98.97
20	15.82	58	241	102.20
25	15.85	49	223	102.39
30	15.85	48	221	102.39
35	15.84	48	221	102.33
40	15.81	48	221	102.13
50	13.81	47	221	89.21
60	13.81	47	221	89.21
70	15.85	34	164	102.39
80	15.84	34	164	102.33
90	15.85	34	164	102.39
100	15.85	32	158	102.39
110	15.85	24	113	102.39
120	15.86	5	18	102.45
140	15.85	3	14	102.39
160	14.16	2	12	91.47
180	14.17	2	12	91.54
200	14.11	2	4	91.15

Wird stattdessen eine TestsInIf Penalty von 80% gewählt, so ändern sich die ersten Einträge wie folgt (alle übrigen bleiben gleich):

MinCost	Laufzeit (in sec)	CSEs	Uses	Prozent von CSE
5	15.32	64	253	98.97
10	15.34	58	241	99.10
20	15.35	57	239	99.16
25	15.85	48	221	102.39

Intel/icc, considerUseCount=1, TestsInIfPenality=15%

MinCost	Laufzeit (in sec)	CSEs	Uses	Prozent von CSE
Original	14.01	0	0	140.38
CSE	9.98	65	253	100.00
5	10.01	64	253	100.30
10	10.02	64	253	100.40
20	10.01	58	241	100.30
25	9.58	49	223	95.99
30	9.58	49	223	95.99
35	9.58	48	221	95.99
40	9.57	48	221	95.89
50	9.67	47	221	96.89
60	9.68	47	221	96.99
70	9.54	34	164	95.59
80	9.53	34	164	95.49
90	9.54	34	164	95.59
100	9.71	32	158	97.29
110	9.72	24	113	97.39
120	9.72	5	18	97.39
140	9.71	3	14	97.29
160	9.67	2	12	96.89
180	9.66	2	12	96.79
200	9.66	2	4	96.79

Wird stattdessen eine TestsInIf Penalty von 80% gewählt, so ändern sich die ersten Einträge wie folgt (alle übrigen bleiben gleich):

MinCost	Laufzeit (in sec)	CSEs	Uses	Prozent von CSE
10	10.01	57	239	100.30
20	10.01	57	239	100.30
25	9.58	48	221	95.99
30	9.6	48	221	96.19

Literaturverzeichnis

[Adv01] Advanced RISC Machines Ltd. *ARM7TDMI Technical Reference Manual*, 2001. Document number RM DDI 0029G.

[AEG+02] Tom Vander Aa, Lieven Eeckhout, Bart Goeman, Hans Vandierendonck, Tanja Van Achteren, Rudy Lauwereins, and Koen De Bosschere. Optimizing a 3D image reconstruction algorithm: investigating the interaction between the high-level implementation, the compiler and the architecture. In *Proceedings of the seventh Asia-Pacific conference on Computer systems architecture*, pages 119–126. Australian Computer Society, Inc., 2002.

[ASU96] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1996.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[BKC99] Maarten Boekhold, Ireneusz Karkowski, and Henk Corporaal. Transformation and parallelizing ANSI C Programms using Pattern Recognition. In *High Performance Computing and Networking Conference*, pages 673–682, April 1999. Amsterdam.

[BTC89] M. Bister, Y. Taeymans, and J. Cornelis. Automatic Segmentation of cardiac MR Images. *IEEE Journal on Computers in Cardiology*, pages 215–218, 1989.

[Bun03] Statistisches Bundesamt. Technologie in Haushalten. `http://www.destatis.de/presse/deutsch/pk/2003/iuk_privat.pdf`, 2003.

[Cal03] Robert Callan. *Neuronale Netze im Klartext*. Pearson Studium, 2003.

[CDK+02] Francky Catthoor, Koen Danckaert, Chidamber Kulkarni, Erik Brockmeyer, Per Gunnar Kjeldsberg, Tanja van Achteren, and Thierry Omnes. *Data Access and Storage Management for embedded programmable Processors*. Kluwer Academic Publishers, 2002.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM Press, 1989.

[Coc70] John Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, 1970.

[CWK+99] M. Coors, O. Wahlen, H. Keding, O. Lüthje, and H. Meyr. TI C62x Performance Code Optimization. In R. Ester H. Rogge, editor, *DSP Deutschland '99 - Grundlagen, Architekturen, Tools, Applikationen*, pages 155–164, Poing, September 1999. Design & Elektronik.

[Edw00] Stephen A. Edwards. *Languages for Digital Embedded Systems*. Kluwer Academic Publishers, 2000.

[Fal04] Heiko Falk. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. PhD thesis, University of Dortmund, Dortmund, Germany, June 2004.

[Gre03] Detlef Grell. Rad am Draht. *c't Magazin für Computer Technik*, 14, 2003.

[HP03] J. Hennessy and D. Patterson. *Computer Architekture A Quantative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, 2003.

[Ier03] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, 2003.

[Int02] Intel Corp., Santa Clara. *IA-32 Intel Architecture Software Developer's Manual*, 2002. Volume 3.

[Jak02] Jacek Jakubowski. Architekturunabhängige Quellcodeoptimierung durch Mustererkennung. Master's thesis, Universität Dortmund, 2002.

[Mar03] Peter Marwedel. *Embedded System Design*. Kluwer Academic Publishers, 2003.

[MP69] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, 1969.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, 1997.

[PBSB01] Gilles Pokam, Stephane Bihan, Julien Simonnet, and Francois Bodin. SWARP: A retargetable Preprozessor for Multimedia Instructions. In *Proceedings of Workshop on Compilers for Parallel Computers (CPC)*, 2001.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65, 1958.

[Ros62] F. Rosenblatt. *Principles of Neurodynamics*. Spartan Books, New York, 1962.

[Sta99] Richard Stallman. *Using and Porting the GNU Compiler Collection GCC*. Free Software Foundation, 1999.

[Str88] P. Strobach. A new technique in scene adaptive coding. In *Proceedings of "European Signal Processing Conference" (EUSIPCO)*, pages 1141–1144, Grenoble, September 1988.

[Zel94] Andreas Zell. *Simulation Neuronaler Netze.* Addison-Wesley, 1994.

Dank an ...

... meine Betreuer Dr. Heiko Falk und Prof. Dr. Marwedel für die erstklassige Betreuung

... Patrick Gundlach fürs Korrekturlesen und seine unglaublichen LaTeX Kenntnisse

... meine Schwester Claudia für unermüdliches Korrekturlesen

... Christian Wolder fürs Korrekturlesen

... meine Liebste Caroline für alles

... meine Eltern für ihre Unterstützung

... die Mitarbeiter des Lehrstuhl 12