Performanceanalyse und plattformspezifische Optimierungen am Beispiel des Grid-ALU-Prozessors

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Angewandte Informatik

der Universität Augsburg

eingereicht von
Dipl.-Inf. Ralf Jahr

Erstgutachter: Prof. Dr. Theo Ungerer

Zweitgutachter: Prof. Dr. Bernhard Bauer

Tag der mündlichen Prüfung: 9. Februar 2012

Abstract

Sequentielle Programme können nicht vom Trend zu immer mehr Prozessorkernen profitieren. Zur Beschleunigung dieser Programme bedarf es neuer Architekturen. Ein Beispiel ist der Grid-ALU-Prozessor (GAP). Er ähnelt einer superskalaren Architektur, bei der eine dreidimensionale Struktur aus Funktionseinheiten zur Ausführung verwendet wird.

Um bereits kompilierte Programme für die Ausführung auf dem GAP zu optimieren kann der Post-Link-Optimierer GAPtimize verwendet werden. Dieses Tool unterstützt plattformspezifische Optimierungen, mit denen die Ausführungsgeschwindigkeit auf dem GAP durch Abschwächung negativer Einflussfaktoren erhöht werden kann.

Mit einer automatischen Suche im Parameterraum werden Konfigurationen für den GAP ermittelt, die ein optimales Verhältnis aus Hardwareaufwand und Ausführungsgeschwindigkeit aufweisen. Diese Suche wird auf die Code-Optimierungen ausgedehnt. Durch die Verwendung von GAPtimize können signifikant bessere Ergebnisse erreicht werden.

A trend towards many-core processor designs is evident to make use of the quickly growing resources on a chip available for processor designs. Novel architectures are developed to also accelerate sequential legacy programs not being able to gain any profit from multiple cores. The *Grid ALU Processor* (GAP) resembling a superscalar processor with a three dimensional execution unit is used in this work as example.

To exploit its features without the need of rewriting or even recompiling legacy applications for quicker execution on the GAP the post link optimizer *GAPtimize* is introduced. It supports platform specific optimizations to reduce properties of programs restraining GAP's performance. Each of these restraining forces is addressed by at least one optimization.

In an automatic design space exploration, configurations for the GAP are worked out showing near-optimal effectiveness, so gaining the best performance from the hardware resources. The search is extended to code optimizations and it is demonstrated that GAPtimize can improve both performance and area effectiveness of the GAP.

Keywords: Target-specific optimizations, Adaptive optimization, Feedback directed optimizations (FDO), Post-link optimization (PLO), Iterative optimitation, Analysis of binary files, Static analysis, Whole program optimization (WPO), Automatic design space exploration (ADSE), Genetic algorithms

Vorwort und Danksagung

Die vorliegende Dissertation entstand während meiner Zeit als wissenschaftlicher Mitarbeiter ab 2007 am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme an der Universität Augsburg. Ich war im Rahmen des DFG-Projektes zum Grid-ALU-Prozessor (GAP) zusammen mit Dr. Basher Shehan beschäftigt. Zielsetzung des Projektes war, den von Jun.-Professor Sascha Uhrig entworfenen und patentierten neuartigen Prozessor genau zu untersuchen und zu evaluieren. Meine Aufgabe umfasste sowohl die Code-Generierung für den GAP, als auch jegliche Optimierungen zur Steigerung der Ausführungsgeschwindigkeit.

Mein aufrichtiger Dank gebührt meinem Erstbetreuer Prof. Dr. Theo Ungerer sowie dem Erfinder des GAP und Leiter des DFG-Projektes Jun.-Professor Dr. Sascha Uhrig für ihre stetige Unterstützung und Inspiration. Herrn Prof. Dr. Bauer gebührt mein Dank für seine Tätigkeit als Zweitgutachter.

Mit Mitteln des HiPEAC Network of Excellence wurde mir ein kurzer Forschungsaufenthalt in Rumänien an der „Lucian Blaga“ University of Sibiu ermöglicht, was der Beginn einer sehr angenehmen und ergiebigen Kollaboration mit Prof. Dr. Lucian Vintan und besonders Dr. Horia Calborean war.

Außerdem gilt mein Dank im Speziellen meinen Eltern und meiner Schwester sowie allen weiteren Personen in meinem Umfeld, die mich fortwährend unterstützt haben. Der Rückhalt, den ich von ihnen bekommen habe, war unverzichtbar.

Ralf Jahr

Inhaltsverzeichnis

1. Einführung

Prozessoren werden mit der Zeit immer komplexer. Der Grund dafür ist, dass die Anzahl an Transistoren, aus denen ein Prozessor aufgebaut sein kann, mit den Jahren und entsprechend dem Moorschen Gesetz (vgl. [69], Kap. 1) immer mehr gestiegen ist. Dadurch sind zum einen immer größere Designs möglich, zum anderen wird es immer mehr zur Herausforderung, die verfügbaren Hardware-Ressourcen zielführend einzusetzen. Zur Lösung dieser Herausforderung waren in den letzten Jahren mehrere Ansätze zu beobachten.

Beispielsweise wurden die Zwischenspeicher für Befehle und Daten vergrößert. Zwischenspeicher können meist automatisch erzeugt werden, somit kann zusätzlich zur Verfügung stehende Chipfläche ohne allzu großen Aufwand nutzbar gemacht werden. Allerdings ist es nicht sinnvoll, diese Zwischenspeicher über die Maßen zu vergrößern. Die wesentlichen Gründe dafür sind ein ab einem bestimmten Punkt schlechtes Verhältnis zwischen zusätzlich benötigter Chip-Fläche und der Reduktion der Ausführungsgeschwindigkeit sowie Beschränkungen des Design-Prozesses.

Eine weitere Idee ist es, diejenigen Elemente, die auf einem Prozessor für die Befehlsverarbeitung verantwortlich sind, also die Prozessorkerne, zu vervielfachen. Dadurch begründet ist der Trend hin zu immer mehr Prozessorkernen. Außerdem gibt es hybride Konzepte, bei denen sowohl Zwischenspeicher als auch Prozessorkerne als Einheiten (*Tiles*) repliziert werden. Beide Ansätze können in der Regel aber nur eingesetzt werden, um die parallele Verarbeitung verschiedener Programmfäden zu ermöglichen bzw. zu beschleunigen, nicht aber für einen einzelnen Programmfaden.

Ein anderer Weg wird mit dem Grid-ALU-Prozessor (GAP) beschritten. Die wesentlichen Vorteile dieser Prozessorarchitektur sind (a) die Beschleunigung der Ausführung eines einfädigen Programms, (b) die Skalierbarkeit der Architektur, somit kann sie für kleine und für große Chips eingesetzt werden, und (c) dass Programme nicht explizit für diese Architektur übersetzt werden müssen, was bei vielen anderen neuartigen Architekturen der Fall ist.

Viele Programme, die sich derzeit im Einsatz befinden, bestehen oft nur aus einem einzelnen Ausführungsfaden (Legacy-Software). Die Veränderung dieser Programme, sodass sie mehrfädig ausgeführt werden können, wäre mit hohem wirtschaftlichem Aufwand verbunden. Durch die Ausführung auf einer Mehrkernarchitektur kann deshalb nicht ohne Weiteres eine Beschleunigung erzielt werden. Anders ist das beim GAP; mit ihm kann auch für einfädige Programme die wachsende Transistoranzahl nutzbringend verwendet werden.

Es ist anzunehmen, dass in naher Zukunft noch mehr neuartige skalierbare Prozessorarchitekturen präsentiert werden, deren Fokus auf der Beschleunigung einfädiger Programme gerichtet sein wird. Für diese Architekturen und für den GAP gibt es ei-

nige Herausforderungen zu bewältigen, die im Rahmen dieser Arbeit am Beispiel des GAP untersucht werden. Das wesentliche Ziel ist die automatische Optimierung von Legacy-Programmen, sodass bei optimaler Verwendung der zur Verfügung stehenden Ressourcen die maximale Ausführungsgeschwindigkeit erreicht werden kann.

Basis sind die Analyse der Performance und die Isolierung derjenigen Programmeigenschaften, die negativen Einfluss besitzen. Um diese Effekte zu reduzieren werden Programmoptimierungen für einen Post-Link-Optimierer vorgestellt, die auf bereits vorhandene Programme automatisch angewandt werden können und die Vorteile der jeweiligen Zielplattform nutzbar machen. Die Performance kann dann hinsichtlich mehrerer Zielfunktionen untersucht und optimiert werden. Dazu werden optimale Parameter für die Hardware-Parameter des Prozessors mit einer automatischen Suche im Parameterraum ermittelt. Anschließend wird diese Suche nach optimalen Parametern auf die Code-Optimierungen erweitert. Das für den GAP verwendete Vorgehen ist auf andere Prozessoren, die ähnlich dem GAP sequentiellen Code ausführen können, übertragbar.

1.1. Aufbau der Arbeit

Das übergeordnete Ziel ist die beschleunigte Ausführung einfädiger Programme. Sequentielle Programmteile können die Ausführung mehrfädiger Programme stark bremsen, wie das Amdahlsche Gesetz [7] zeigt. Als Beispiel-Architektur wird der GAP verwendet. Ziel ist für den GAP immer die automatische Anwendung von Optimierungen, wobei bestehende Programme weiterhin ohne erneute Übersetzung ausführbar sein sollen.

Im Kapitel 2 wird der GAP eingeführt. Es wird besonderes Augenmerk darauf gelegt, welche Faktoren gute oder schlechte Auswirkungen auf seine Ausführungsgeschwindigkeit haben. Zur Messung der Ausführungszeit wird ein taktgenauer Simulator verwendet und der Begriff *Performance* definiert.

Um die positiven Faktoren zu verstärken und die negativen abschwächen zu können, eignen sich sowohl reine Code-Optimierungen als auch Kombinationen aus Hardware- und Code-Optimierung. Um Software zu verändern, die bereits als ausführbare Datei vorliegt, wird im Kapitel 3 das Programm GAPtimize vorgestellt.

Anschließend werden im Kapitel 4 einzelne Methoden zur Performance-Steigerung erörtert. Mit verschiedenen Konfigurationen des GAP wird untersucht, wie hoch ihr Einfluss auf die Programmausführungszeiten ist. Jeder der beeinflussbaren Faktoren wird mit mindestens einer Optimierung adressiert.

Einen großen Einfluss auf die Performance des GAP hat die Wahl seiner Hardware-Parameter. Sie stehen in direktem Zusammenhang zur benötigten Chip-Fläche. Die Beziehung zwischen Hardware-Parametern und Performance wird mit einer automatischen Suche im Parameterraum im Kapitel 5 untersucht. Hier wird außerdem die Suche nach optimalen Hardware-Parametern mit der Suche nach optimalen Kombinationen von Code-Optimierungen und Parametern für diese Optimierungen kombiniert. Abschließend wird diese Suche auch adaptiv durchgeführt, wobei Hardware-

und Code-Optimierungs-Parameter optimal für die Situation bzw. das jeweilige Programm gewählt werden.

Kapitel 6 fasst die Ergebnisse zusammen und zeigt mögliche weitere Forschungsthemen auf.

Im Anhang A finden sich zusätzliche Diagramme und Tabellen.

Die Struktur wird durch Abbildung 1.1 zusammengefasst. Da verwandte Arbeiten sich bis auf wenige Ausnahmen auf die in den einzelnen Kapiteln verwendeten Techniken beziehen werden sie direkt dort vorgestellt und nicht an zentraler Stelle.

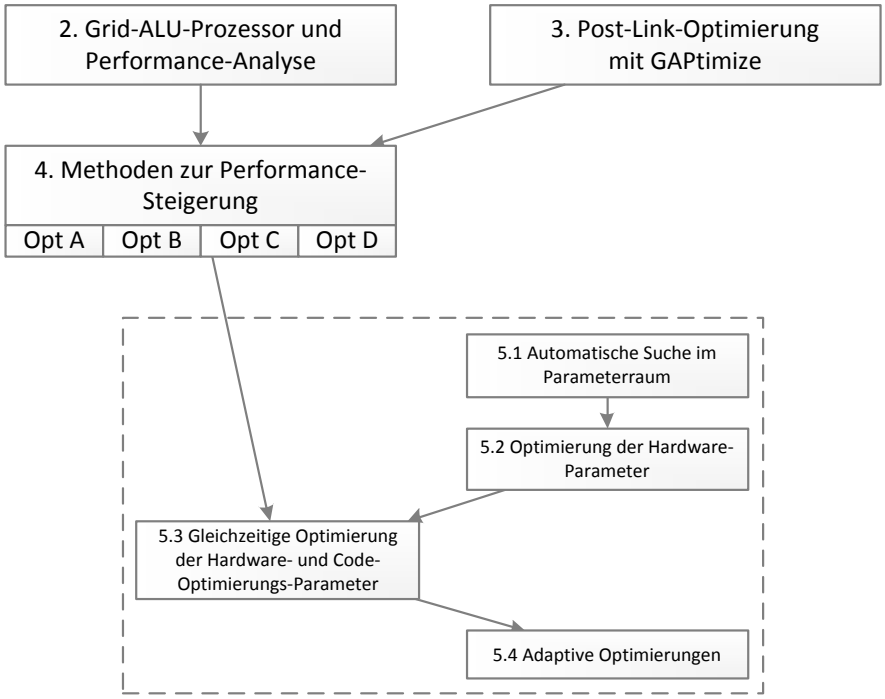


Abbildung 1.1.: Logischer Zusammenhang der wichtigsten Kapitel und Abschnitte

1.2. Beitrag zum Stand der Forschung

Für die GAP-Architekturen werden Stärken und Schwächen dargelegt. Es werden plattformspezifische Optimierungen präsentiert zur Steigerung der Performance des GAP durch Abschwächung der ermittelten negativen Effekte. Sie können auch für andere Architekturen, bei denen der *Instruction Level Parallelism* (ILP) einen starken Einfluss auf die Performance hat, verwendet werden. Als Werkzeug für die Optimierung von Programmen, die nur als ausführbare Dateien vorliegen, wird der Post-Link-Optimierer GAPtimize vorgestellt. Aus dem verwendeten Vorgehen kann für andere Plattformen abgeleitet werden, wie plattformspezifische Optimierungen entwickelt und angewandt werden können.

Mit einem Schätzverfahren kann der nötige Hardware-Aufwand einer GAP-Konfiguration in Relation zu einer anderen Konfiguration bestimmt werden. Diese Vor-

gehensweise kann für andere stark skalierbare Architekturen adaptiert werden. Es wird die Performance, die mit dem GAP erreicht werden kann, in Relation zum nötigen Hardware-Aufwand untersucht. Damit wird verdeutlicht, dass die Untersuchung mehrerer Zielfunktionen für realistische Ergebnisse essentiell ist und es wird dargelegt, wie eine derartige Betrachtung stattfinden kann.

Mit einer automatischen Suche im Parameterraum (*design space exploration*, DSE) werden parallel Hardware- und Code-Optimierungs-Parameter optimiert sowie eine optimale Reihenfolge für die Anwendung von Code-Optimierungen gesucht. Mit den gewonnenen Ergebnissen kann zum einen gezeigt werden, dass GAPtimize die Hardware-Verwendung optimieren und die Performance des GAP steigern kann, also die plattformspezifischen Optimierungen gewinnbringend sind. Zum anderen zeigen die gefundenen Ergebnisse, dass evolutionäre Algorithmen ohne spezielle Anpassung mächtig genug sind, um selbst in einem sehr großen Parameterraum in vertretbarer Zeit sehr gute Konfigurationen zu finden.

Eine weitere Performance-Steigerung für den GAP kann durch adaptive Optimierung der Hardware-Parameter zusammen mit den Code-Optimierungs-Parametern erreicht werden. Adaptive Optimierungen sind auch für andere Prozessoren vielversprechend, wenn die Performance des Prozessors wesentlich von der Struktur oder dem Aufbau des ausgeführten Programms abhängt. Um Optimierungen auszuwählen und anzupassen kann eine automatische Suche im Parameterraum durchgeführt werden.

2. Der Grid-ALU-Prozessor

Zuerst wird die Hardware-Architektur des Grid-ALU-Prozessors vorgestellt, anschließend wie er programmiert wird. Abschließend wird untersucht, welche Faktoren seine Performance wesentlich beeinflussen.

2.1. Hardwarearchitektur

In diesem Kapitel wird der Grid-ALU-Prozessor (GAP) vorgestellt und es wird erklärt, wie er Instruktionen in Konfigurationen umwandelt. Außerdem wird ein kurzer Überblick über verwandte Architekturen gegeben.

2.1.1. Übersicht über die Prozessorarchitektur

Der Grid-ALU-Prozessor (GAP) ist ein neuartiger Forschungsprozessor, der die Stärken von superskalaren Prozessoren und rekonfigurierbaren Systemen vereint. Er wurde von Uhrig et al. vorgestellt [162]. Ziel des GAP ist es, die Ausführung einfädiger sequentieller Befehlsströme zu beschleunigen. Es gibt (bisher) keinen festgelegten Einsatzbereich, er könnte für eingebettete Systeme mit hohen Performance-Anforderungen genauso verwendet werden wie für Desktop-Computer. Man spricht deshalb von einem *general purpose processor*.

Grundlage für den GAP ist ein mehrfach superskalarer Prozessor mit in-order pipeline; Beispiele sind UltraSPARC- oder einfache MIPS-Prozessoren (siehe [59], [69] A.3). Abbildung 2.1 zeigt die Pipeline des SimpleScalar-Simulators [16], mit dem superskalare Prozessoren basierend auf eine einfache MIPS-Architektur simuliert werden können. Die Pipeline verfügt über folgende Phasen:

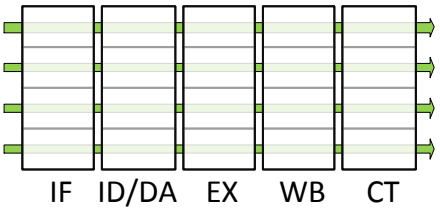


Abbildung 2.1.: Vereinfachte Pipeline eines superskalaren Prozessors

***Instruction fetch* (IF)** Laden von Instruktionen aus dem Befehlszwischenspeicher

***Instruction decode* (ID/DA)** Instruktionen werden dekodiert und Ausführungseinheiten zugewiesen.

***Execute* (EX)** In den Ausführungseinheiten werden die Befehle ausgeführt.

***Write back* (WB)** Die in den Ausführungseinheiten berechneten Ergebnisse werden z. B. in den Speicher zurückgeschrieben.

***Commit* (CT)** Die Befehle und ihre Ergebnisse werden gültig gemacht und die neu berechneten Daten stehen somit endgültig zur Verfügung.

Die Einheiten, in denen die Phasen IF und ID/DA ausgeführt werden, bilden das Frontend des Prozessors, die übrigen Stufen das Backend.

Beim GAP ist das Backend durch eine neue Struktur ersetzt worden, das *Array*. Es besteht aus sehr vielen relativ kleinen *Funktionseinheiten*, die in Zeilen und Spalten angeordnet sind (Kurzschreibweise: `zeilen x spalten`). Um Befehle in dieser Struktur platzieren zu können wird das Frontend um eine weitere Stufe erweitert, die *Konfigurationseinheit*. Zusätzlich befindet sich links neben dem Array eine spezialisierte Einheit zur Sprungbehandlung (*branch control unit*, BCU). Sie führt alle Befehle aus, die den Programmfluss verändern können. Rechts neben dem Array ist pro Zeile des Arrays eine Speicherzugriffseinheit (*memory access unit*, MAU), die auf den Arbeitsspeicher zugreifen kann. Diesen Aufbau zeigt Abbildung 2.2.

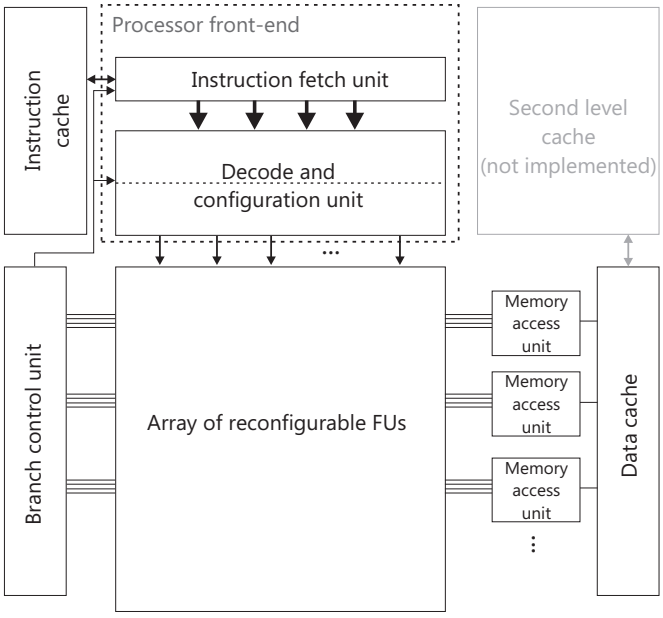


Abbildung 2.2.: Schematischer Aufbau des GAP

Im Array (vgl. Abbildung 2.3) befinden sich viele Funktionseinheiten, die Ganzzahl-Berechnungen durchführen können. Jeder Spalte des Arrays ist ein Architekturregister des Befehlssatzes zugeordnet. Am Anfang jeder Spalte steht ein Register (also Speicher), das den Wert des Architekturregisters bei Start der Berechnungen enthält. In der Spalte können nur Befehle konfiguriert werden, die das Architekturregister der Spalte, das ihr zugeordnet ist, schreiben. Liest ein Befehl die Werte weiterer Register und somit anderer Spalten, so können sie über die horizontalen Verbindungen gelesen

werden. Dazu wird ein Verbindungsnetzwerk verwendet, das die Daten sehr schnell zur Verfügung stellt.

Die Ausführung und die Datenweitergabe im Array erfolgt prinzipiell von oben nach unten. Auf diese Weise können Abhängigkeiten zwischen Befehlen abgebildet werden, indem abhängige Befehle in verschiedenen Zeilen positioniert werden.

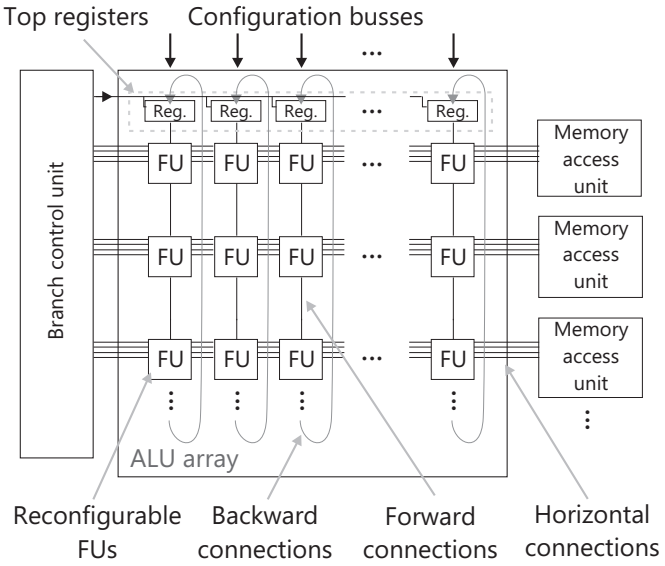


Abbildung 2.3.: Struktur des Arrays im GAP mit Branch Control Unit und den Memory Access Units

In normalen superskalaren Prozessoren wird für jeden Befehl mindestens ein Takt Ausführungszeit veranschlagt. Da einige Befehle, z. B. bitweise Verschiebungen, sehr schnell ausgeführt werden können, ergeben sich dadurch Pausen bis zum Ende des Takts. Diese Pausen werden im GAP durch die asynchrone Ausführung von Befehlen im Array stark reduziert. Abhängige Befehle starten dabei ihre Berechnungen nicht bei Taktflanken sondern sobald spezielle Signale gesetzt sind.

Als Befehlssatz wird für den GAP die *Portable Instruction Set Architecture* (PISA) verwendet, die vom SimpleScalar-Simulator [16] unterstützt wird. Damit kann der GAP dieselben Programmdateien ausführen. Der Befehlssatz verfügt über 32 Architekturregister, deshalb hat der GAP in seiner Basis-Architektur 32 Spalten. Die Anzahl der Zeilen kann zwischen 4 und 32 gewählt werden.

Ist die letzte Zeile des Arrays erreicht oder ein Ereignis aufgetreten, das die Ausführung von unvorhergesehenem Code erfordert, so werden alle Daten aus den Spalten in die Register am Kopf der Spalten kopiert und das Array wird gelöscht. Dann kann die Konfiguration des Arrays wieder in der ersten Zeile beginnen.

Mit den von Shehan et al. [142] vorgestellten Erweiterungen lässt sich die Anzahl der Spalten auf Werte kleiner als 32 setzen. Dazu werden die Register von den Spalten entkoppelt und zur Laufzeit freien Spalten zugeordnet. Ist keine freie Spalte mehr für ein Register vorhanden, so muss zur weiteren Ausführung des Programms das Array gelöscht werden.

Um die Zahl der Neukonfigurationen zu reduzieren kann das Array durch Konfigurationsebenen quasi um eine dritte Dimension erweitert werden (Kurzschreibweise: `spalten x zeilen x ebenen`[1]). Dazu werden alle Funktionseinheiten im Array einschließlich der MAUs und die BCU um kleine Speicher erweitert, in denen nicht nur eine sondern mehrere Instruktionen abgelegt werden können. Die Konfigurationsebenen werden ähnlich zu einem Trace Cache (siehe z. B. [135]) verwendet, in dem Programmteile, also die Konfigurationen, für die spätere Wiederverwendung gespeichert werden. Dadurch kann die Programmausführung stark beschleunigt werden.

Da die Konfigurationseinheit sowohl die Analyse der Befehle auf Abhängigkeiten wie auch darauf basierend die Konfiguration des Arrays erledigt, ist keine spezielle Software nötig, mit der Programme für die Ausführung auf dem GAP vorbereitet werden müssten. Somit ist ein mögliches Einsatzszenario für den GAP, dass er verwendet werden kann, um einen Prozessor mit dem selben Befehlssatz zu ersetzen und so eine höhere Performance ohne großen Aufwand bereitstellen kann. Die auszuführenden Programme müssen nicht speziell für den GAP angepasst werden, ein Zugriff auf den Quellcode ist somit nicht notwendig. Es kann also Legacy-Code direkt ausgeführt werden.

Aktuelle Prozessoren verfügen häufig über die *out-of-order*-Ausführung. Bei der in-order-Ausführung werden alle Befehle in der Reihenfolge, in der sie im Befehlsstrom stehen, ausgeführt. Bei der out-of-order-Ausführung führt der Prozessor Befehle auch außerhalb ihrer Reihenfolge im Befehlsstrom aus, ist also durch die Reihenfolge nicht gebunden. Er kann so seine Ausführungseinheiten wesentlich besser ausnutzen. Aufgrund der Tatsache, dass in-order-Prozessoren als „etwas angestaubt“ gelten, wird für Performance-Vergleiche zwischen GAP und SimpleScalar der SimpleScalar-Simulator immer im out-of-order-Modus konfiguriert.

2.1.2. Konfigurieren des Arrays

Am Beispiel des Programmstücks 2.1, das eine Funktion zur Berechnung des Skalarprodukts $x = \sum_{i=0}^{2048} a[i] * b[i]$ zweier Vektoren zeigt, wird erklärt, nach welchen Regeln die Konfigurationseinheit Befehle im Array des GAP platziert und wie die Ausführung abläuft.

Das Programm besteht aus einigen Zeilen, die nur einmalig beim Aufruf der Funktion aufgerufen werden (`400280 - 4002a8` und `400300 - 400308`). Dazwischen befindet sich eine Schleife, die 2048-mal durchlaufen wird (`4002b0 - 4002f8`). Entsprechend lässt es sich in *Basic Blocks* unterteilen. Ein Basic Block besteht aus Instruktionen, die von einer Startinstruktion aus bei der Ausführung von exakt einer anderen Instruktion erreicht werden können. Hat eine Zeile mehr als einen Vorgänger, so ist sie selbst Startinstruktion eines Basic Blocks.

Den wichtigsten Einfluss auf die Abbildung der Befehle auf das Array haben die Datenabhängigkeiten unter den Befehlen; sie sind für den Programmteil `400280 -`

[1] Ergänzend kann auch die Konfiguration des Befehlszwischenspeichers als `a x b x c` geschrieben werden. Die Größe des Caches wird dann wie folgt berechnet: $a * b * c * 8$ Byte

Quellcode 2.1: Beispielcode zur Berechnung des Skalarprodukts

```
400280:  addu $7,$0,$0
400288:  addu $6,$0,$0
400290:  lui $5,4096
400298:  addiu $5,$5,7552
4002a0:  lui $4,4096
4002a8:  addiu $4,$4,15760

4002b0:  lw $2,0($4)
4002b8:  lw $3,0($5)
4002c0:  mult $2,$3
4002c8:  addiu $6,$6,1
4002d0:  mflo $2
4002d8:  addiu $5,$5,4
4002e0:  addiu $4,$4,4
4002e8:  addu $7,$7,$2
4002f0:  slti $2,$6,2049
4002f8:  bne $2,$0,4002b0

400300:  addu $2,$0,$7
400308:  jr $31
```

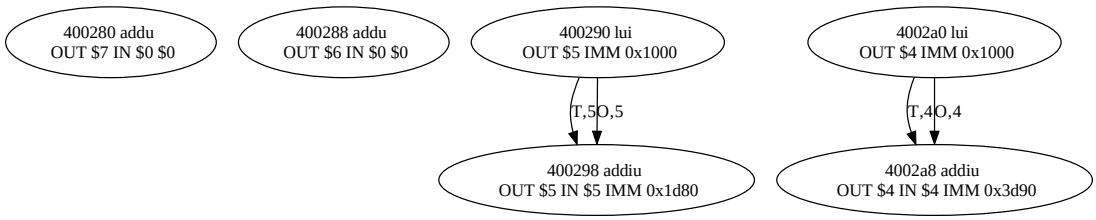


Abbildung 2.4.: Datenabhängigkeiten des Initialisierungscodes vor dem eigentlichen Schleifenkörper

4002a8 in Abbildung 2.4 zu sehen. Abbildung 2.5 zeigt die Abhängigkeiten der Befehle in der Schleife. Es sind nur solche Abhängigkeiten sichtbar, die nicht Schleifeniterationen überspannen (das wären sonst *loop-carried data dependencies*). Diese Datenabhängigkeitsgraphen (*data dependency graph*, DDG) sind gerichtete azyklische Graphen (*directed acyclig graphs*, DAGs) und wie folgt aufgebaut:

Knoten Jede Instruktion wird durch einen Knoten repräsentiert.

Kanten Jede Kante steht für eine Datenabhängigkeit und ist mit zwei Werten beschriftet, z. B. `A,4`. Der Buchstabe steht für den Typ der Datenabhängigkeit (T = echte Datenabhängigkeit, O = Ausgabeabhängigkeit, A = Gegenabhängigkeit[2]). Die

[2] Ausgabeabhängigkeiten und Gegenabhängigkeiten werden auch als falsche Datenabhängigkeiten bezeichnet, da sie vermeidbar wären.

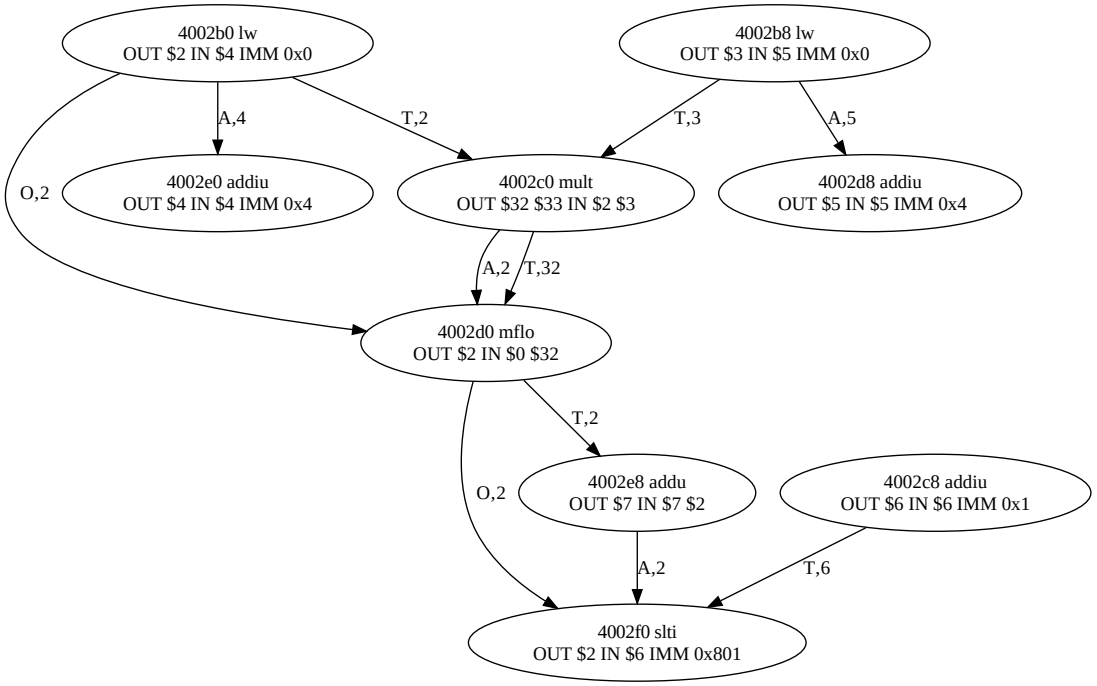


Abbildung 2.5.: Datenabhängigkeiten im Schleifenkörper zur Berechnung des Skalarprodukts

Zahl steht für die Ressource, die von der Datenabhängigkeit betroffen ist. Werte zwischen 0 und 31 stehen für normale Register, 32 und 33 für die Ergebnis-Register der Multiplikations-/Divisionseinheit. Abhängigkeiten bzgl. des Arbeitsspeichers werden als Abhängigkeiten bzgl. Register -1 (das eigentlich nicht existent ist) dargestellt.

Der Vollständigkeit halber ist zu erwähnen, dass der Kontrollflussgraph eines Programms sehr ähnlich aufgebaut ist. Die Instruktionen oder Blöcke des Programms bilden die Knoten des Graphen. Kann nach der Ausführung eines Knotens ein anderer erreicht werden (normaler Kontrollfluss oder Sprung), so muss zwischen beiden Knoten eine Kante vorhanden sein.

Eine gültige Ausführungsreihenfolge für die Instruktionen in einem Basic Block ist dadurch festgelegt, dass Instruktionen nur dann ausgeführt werden können, wenn alle ihre Vorgänger im DDG bereits ausgeführt wurden. So wird sichergestellt, dass keine Konflikte auftreten können. Mögliche Konflikte sind z. B. das Verarbeiten von noch nicht berechneten Daten oder das Überschreiben von Daten, die noch von anderen Instruktionen gelesen werden sollen. Voraussetzung ist dafür natürlich, dass der DDG zyklenfrei ist.

Alle Instruktionen müssen vor ihrer Ausführung durch den GAP erst das Prozessor-Frontend durchlaufen und durch die Konfigurationseinheit auf das Array des GAP abgebildet werden. Dabei gelten die folgenden Regeln:

- Die **Spalte**, in der eine Instruktion abgelegt wird, wird über das Register, das sie

schreibt, festgelegt. Sprunginstruktionen werden in der BCU abgelegt, Speicheroperationen in den MAUs und Multiplikations- oder Divisionsbefehle in der entsprechenden Einheit.

- Die **Zeile** wird vorrangig durch die Datenabhängigkeiten bestimmt. Eine Instruktion muss immer unter allen Befehlen positioniert werden, von denen sie abhängig ist (eingehende Datenabhängigkeiten). Zudem kann sie nur unterhalb einer anderen Instruktion in derselben Spalte positioniert werden.

Nehmen wir an, es wird mit einem Funktionsaufruf `jal` zur Instruktion `400280` gesprungen. Dieser Sprung befinde sich in der zweiten Zeile des Arrays, es sind bereits Spalten für die Register 16 und 17 reserviert. Nach einem Sprung werden prinzipiell erst in der nächsten Zeile des Arrays Befehle abgelegt. Bei einem Array mit sechs Zeilen, sechs Spalten und mehreren Konfigurationsebenen (abgekürzt als `6x6xN`) werden die Instruktionen wie folgt abgebildet:

400280 Die Instruktion schreibt Register 7. Dazu wird eine neue Spalte benötigt, es sind dann noch drei Spalten nicht belegt. 400280 ist die erste Instruktion nach einem Sprung und Kopf eines Basic Blocks, deshalb hat sie keine eingehenden Datenabhängigkeiten und wird somit in die erste verfügbare Spalte positioniert.

400288 Die Instruktion schreibt Register 6, dazu wird eine neue Spalte verwendet. Es bleiben zwei nicht belegte Spalten. Sie hat keine Datenabhängigkeiten und wird somit neben `400280` abgelegt.

400290 Die Instruktion liest Register 5; es wird analog zu `400288` verfahren, es bleibt eine unbenutzte Spalte.

400298 Diese Instruktion schreibt auch Register 5. Außerdem verfügt sie über Datenabhängigkeiten `T,5` und `O,5` zu `400290`. Sie wird deshalb unter `400290` abgelegt.

4002a0 Die letzte freie Spalte wird Register 4 zugewiesen.

4002a8 Diese Instruktion hat Datenabhängigkeiten zu `4002a0` und schreibt dasselbe Register, deshalb wird sie in dieselbe Spalte wie `4002a8` gesetzt unter `4002a0`.

Die entstandene Konfiguration ist in Abbildung 2.6 zu sehen. Jetzt ist allen Spalten ein Register zugewiesen. Die nächste Instruktion, die ein Register schreibt, das nicht in der Menge $\{4, 5, 6, 7, 16, 17\}$ enthalten ist, kann nicht mehr platziert werden. Dazu muss erst (a) bei Verwendung einer Konfigurationsebene das Array gelöscht werden oder (b) bei Verwendung mehrerer Konfigurationsebenen ein Wechsel zu einer anderen Konfigurationsebene stattfinden. Selbiges findet auch statt, wenn das Ende des Arrays erreicht ist oder die gewünschte Funktionseinheit nicht mehr verfügbar ist, wenn z. B. eine zweite Multiplikation stattfindet.

Die weiteren Befehle werden wie folgt konfiguriert:

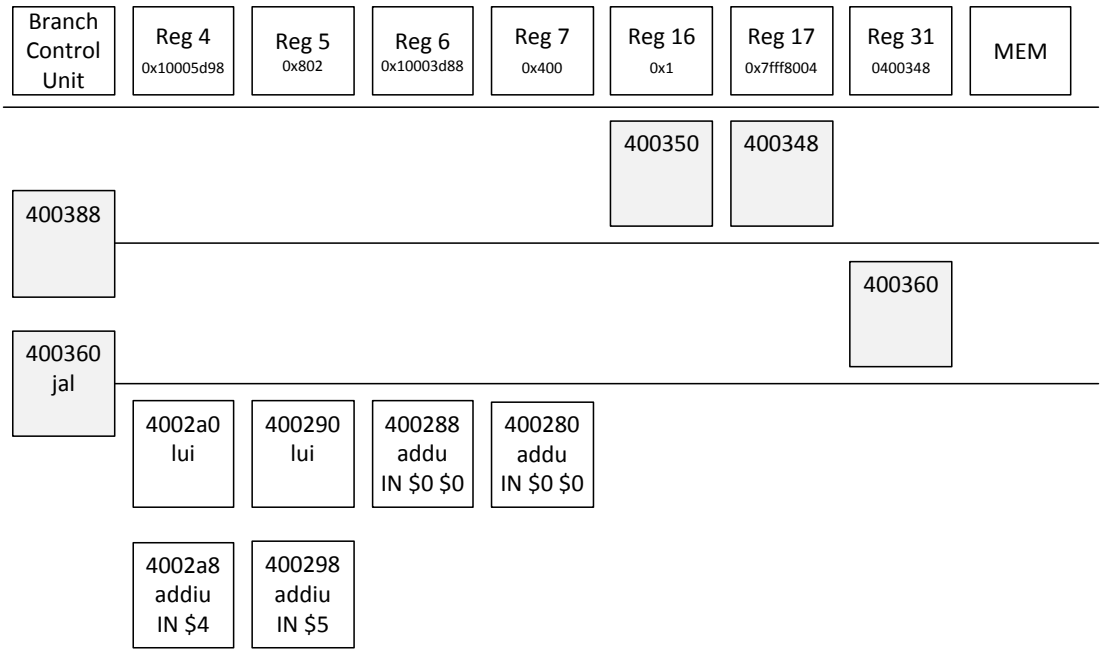


Abbildung 2.6.: Konfiguration für die Befehle `400280` bis `4002a8` für ein Array mit 6x6 Funktionseinheiten. Instruktionen außerhalb von Listing 2.1 sind grau markiert.

4002b0 Die Instruktion schreibt Register 2, für das keine Spalte mehr verfügbar ist. Somit werden alle berechneten Werte in die Register am Anfang der Spalten kopiert und das Array wird gelöscht. Der Befehl `4002b0` wird in die erste Zeile gesetzt.

Da er auf den Arbeitsspeicher zugreift, wird er in zwei Teil-Befehle aufgespalten, die aber sehr eng gekoppelt sind, deshalb wird ihre Interaktion nicht als Datenabhängigkeit modelliert sondern sie werden in dieselbe Zeile gesetzt. Der erste Teil wird in der Spalte mit den MAUs abgelegt, der zweite Teil, der das gelesene Datum aus der MAU in das Zielregister kopiert, wird in der Spalte des Zielregisters, also Register 2, positioniert.

4002b8 Der Befehl wird analog zu `4002b0` behandelt, Zielregister ist aber Register 3. Es bleiben vier freie Spalten. Da er auch eine MAU benötigt und die in der ersten Zeile schon belegt ist, wird er in die zweite Zeile gesetzt.

4002c0 Dieser Befehl ist eine Multiplikation, er schreibt Register 32 und 33 und hat Datenabhängigkeiten zu den Befehlen `4002b0` und `4002b8`. Deshalb muss er in jedem Fall unter diesen Befehlen stehen. Somit wird er in die dritte Zeile gesetzt und belegt die Multiplikations-Divisions-Einheit.

4002c8 Dieser Befehl hat keine eingehenden Datenabhängigkeiten und schreibt das Register 6, dem noch keine Spalte zugewiesen ist. Er wird also in der ersten Zeile positioniert.

Die Befehle 4002d0 bis 4002e8 werden analog verarbeitet.

4002f0 Diese Instruktion ist interessant, da sie den Schleifenrücksprung realisiert. Als Sprung wird sie von der BCU ausgeführt. Trotz Datenabhängigkeit wird sie in dieselbe Zeile wie 4002f0 gesetzt, da die BCU stets etwas versetzt mit den Instruktionen im Array arbeitet. Durch Vergleich der Adresse des Sprungbefehls und des Ziels stellt die Konfigurationseinheit fest, dass es sich um einen Rücksprung handelt. In diesem Fall wird das Array nach dem Sprungbefehl nicht wie bei Vorwärtssprüngen mit dem wahrscheinlichsten Sprungziel konfiguriert sondern stets mit den im Speicher nachfolgenden Instruktionen, also hier die Befehle ab 400300. Für die Ermittlung des „wahrscheinlichsten" Sprungziels wird eine Sprungvorhersagetechnik eingesetzt, beim GAP ist das derzeit ein 2-Bit träger Automat (*bimodal branch predictor*).

Die entstandene Konfiguration ist in Abbildung 2.7 zu sehen.

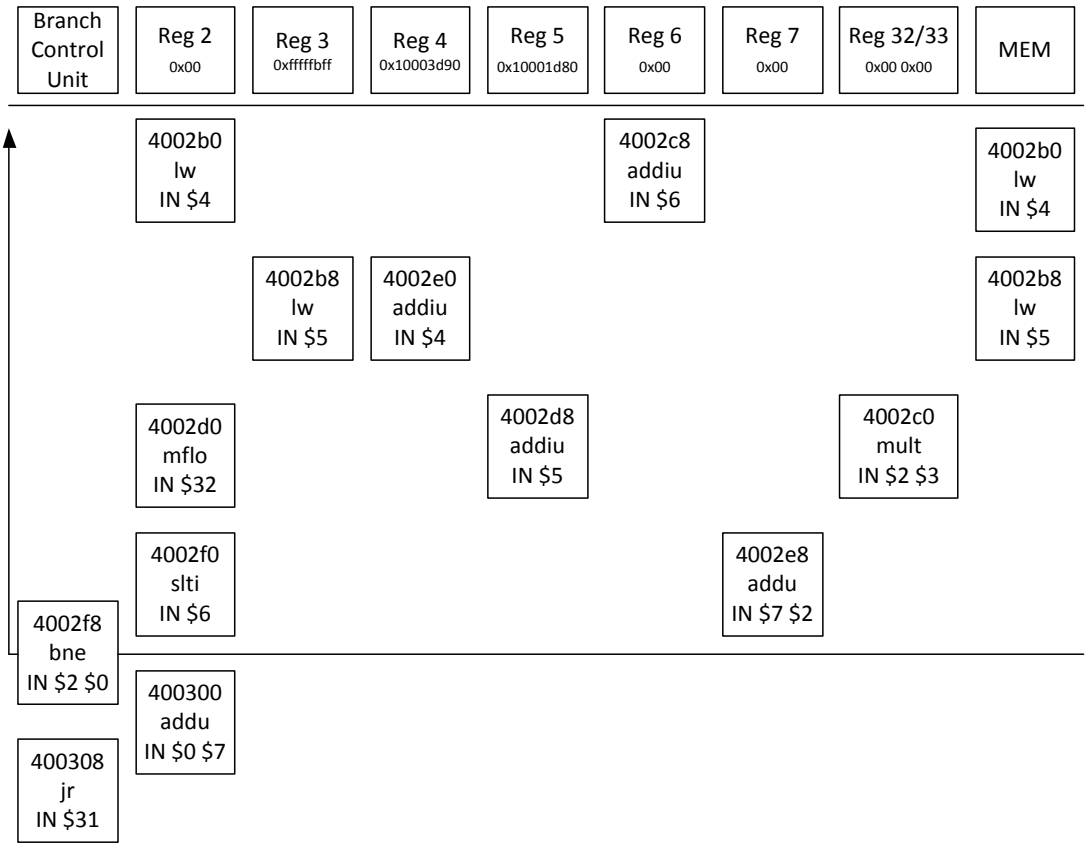


Abbildung 2.7.: Konfiguration für die Befehle 4002b0 bis 400308 für ein Array mit 6x6 Funktionseinheiten.

Bei der Ausführung dieser Konfiguration wird beim Rücksprung überprüft, ob Register 2 und Register 0 den selben Wert besitzen, also ob der Wert in Register 2 gleich 0 ist. Wenn ja, geschieht nichts weiter und die Ausführung wird bei 400300 fortgesetzt. Wenn nein, so wird der Sprung genommen und es wird dazu überprüft, ob

das Sprungziel mit der ersten Instruktion auf der Konfigurationsebene übereinstimmt. Das ist der Fall, dadurch kann die Ausführung direkt fortgesetzt werden ohne die Befehle des Schleifenkörpers neu in das Array einbauen zu müssen. Auf diese Weise kann die Ausführung der Schleife stark beschleunigt werden, da wesentlich weniger zeitintensive Zugriffe auf den Befehlszwischenspeicher oder den Arbeitsspeicher nötig sind.

Ist nur eine Konfigurationsebene vorhanden, so kann das Sprungziel nur mit der ersten Instruktion dieser Ebene verglichen werden. Man spricht dann von der Schleifenbeschleunigung. Sind im Prozessor mehrere Konfigurationsebenen vorhanden, so wird nicht nur der erste Befehl in der aktuellen Ebene überprüft sondern auch die ersten Befehle in allen anderen Konfigurationsebenen. Ist eine Übereinstimmung vorhanden, so kann mit geringer Unterbrechung die Ausführung in dieser Konfigurationsebene fortgesetzt werden. Dann ist die Beschleunigung den Konfigurationsebenen an sich zuzurechnen.

2.1.3. Verwandte Prozessorarchitekturen

Es gibt auf den ersten Blick viele Prozessorarchitekturen, die Ähnlichkeiten mit dem GAP aufweisen. Sie verfügen in der Regel über Architekturelemente, die in Form eines Gitters angeordnet werden können. Beim GAP werden Funktionseinheiten repliziert, sie werden zur Ausführung von Befehlen verwendet und die Konfigurationen für das Array werden zur Laufzeit direkt in der Hardware berechnet. Auf dem Array werden alle Befehle ausgeführt, es gibt keinen nebengeordneten Prozessor.

Anders ist das bei den folgenden Architekturen:

MOLEN [163, 164, 160], GARP [68] und MorphoSys [93] Bei diesen Architekturen werden rekonfigurierbare Elemente analog zu einem Koprozessor verwendet. Sie verfügen dazu über einen Hauptprozessor, der den Koprozessor steuert. Deshalb können nur bestimmte Programmteile auf den rekonfigurierbaren Elementen ausgeführt werden.

Der Morphosys-Prozessor hat einen RISC-Hauptprozesor und ein *Reconfigurable Cell Array*, in dem jede Zelle eigentlich selbständig konfiguriert wird und arbeiten kann. Um Code auf dem Array ausführen zu können muss er zur Übersetzungszeit in spezielle Instruktionen umgeformt werden, mit denen das Array dann konfiguriert wird. Die einzelnen Zellen scheinen ähnlich zu arbeiten wie sehr stark vereinfachte Kerne in einem Many-Core-Prozessor.

Legacy-Code muss für die Morphosys-Architektur eigens aufbereitet und neu übersetzt werden [93]. Alternativ kann auch die Programmiersprache SA-C verwendet werden, aus der dann automatisch Code für den RISC-Prozessor und das Array erzeugt werden kann [165]. Bestehender Code kann also nicht ohne weiteres auf RISC-Prozessor und Array übertragen werden.

XTENSA,CHIMAERA [67] und der PRISC Prozessor [132] verwenden rekonfigurierbare Elemente wie auch der GAP Prozessor direkt in der Pipeline. Allerdings muss hier das Programm speziell für den Prozessor kompiliert sein; es

sind spezielle Tools nötig. Bestehende Software muss somit zumindest neu übersetzt werden.

VEAL [33] und WARP [102] Diese Architekturen können dynamisch Programmteile auf eine separate nebengeordnete Hardwareeinheit abbilden. Allerdings entsteht dabei ein Overhead und es werden nicht alle Programmteile auf der rekonfigurierbaren Hardware ausgeführt.

TRIPS [111] Bei diesem Projekt wird eine vollständig neue Befehlsarchitektur namens EDGE [17] eingeführt. Um ein Programm in EDGE auszudrücken muss spezielle Software verwendet werden. Die Platzierung von Befehlen auf der Hardware wird durch Software berechnet. Bestehender Code muss also neu übersetzt werden.

RAW [150, 151] Die Skalierbarkeit steht im Vordergrund. Allerdings sind die replizierten Kacheln nur sinnvoll verwendbar, wenn mehr als ein Thread ausgeführt wird. Somit wird sequentieller Code – anders als beim GAP – nicht per se beschleunigt.

Für den RAW-Prozessor gibt es wohl einen Software-Simulator [169], der Code ausführen kann, der für einen x86-Prozessor kompiliert wurde. Allerdings zeigt dieser Ansatz, obwohl 16 Elemente ähnlich zu MIPS-Prozessoren verwendet werden, keine Beschleunigung.

Die Abgrenzung von ähnlichen Architekturen findet sich detaillierter in der Dissertation von Basher Shehan [143] und dem Artikel von Sascha Uhrig et al. [162]. Mit obigen Beispielen wird aber klar, dass all diese augenscheinlich verwandten Architekturen anders programmiert werden müssen als der GAP, da Routing und Placement meistens in der Software stattfinden oder nur bestimmte Programmteile aufwändig im Prozessor isoliert und analysiert werden. Aus Sicht der Programmierung sind mit dem GAP am ehesten die superskalaren Architekturen und die VLIW Architekturen vergleichbar.

Eine superskalare Architektur bildet die Basis des GAP Prozessors, deshalb gibt es große Ähnlichkeiten dazu. Auch die Programme, die der GAP ausführen kann, sind tatsächlich für einen superskalaren Prozessor kompiliert worden. Superskalare Architekturen verarbeiten prinzipiell mehrere Befehle (siehe auch Grafik 2.1). Dabei lesen die Stufen IF und ID mehrere Befehle ein. Diese Befehle werden dann den Ausführungseinheiten zugewiesen. Hier wird wie folgt unterschieden:

in-order Ausführung Befehle werden nur in der Reihenfolge, in der sie aus dem Befehlsstrom gelesen werden, den Ausführungseinheiten zugewiesen. Dabei muss auf Datenabhängigkeiten geachtet werden; abhängige Befehle (siehe Abschnitt 2.1.2) können nicht im selben Takt zugewiesen werden. Spekulation kann bzgl. Sprüngen stattfinden. Befehle werden gültig gemacht, wenn Sprünge, von denen sie abhängig sind, wie vorhergesagt, ausgeführt wurden. Andernfalls werden die Ergebnisse dieser Befehle verworfen.

out-of-order Ausführung Bei der out-of-order Ausführung können Befehle in beliebiger Reihenfolge unabhängig von der Anordnung im Befehlsstrom ausgeführt werden. Dazu werden sie vor der Zuordnung in einen Puffer geschrieben, aus dem dann in jedem Takt maximal ein Befehl pro Ausführungseinheiten geholt und zugewiesen wird. Befehle, die ungelöste Datenabhängigkeiten haben, können nicht in einem Takt zugewiesen werden, da ihre Eingabewerte noch nicht zur Verfügung stehen. Wenn alle Abhängigkeiten von Befehlen korrekt vorhergesagt wurden, werden Befehle gültig gemacht, ansonsten werden sie verworfen.

Bei out-of-order Prozessoren sind sowohl die Zuordnungstufe wie auch die Stufen für das Gültigmachen von Befehlen wesentlich komplexer aufgebaut und es sind viele zusätzliche Speicher für Befehle notwendig [44].

Eine genaue Beschreibung von superskalaren Prozessoren ist z. B. in den Büchern [14] und [69] zu finden. Das Frontend des GAP ähnelt am meisten dem eines superskalaren Prozessors mit in-order-Zuweisung. Es sind keine Pufferspeicher für Befehle vorhanden. Allerdings kann die Konfigurationseinheit des GAP auch Befehle mit Abhängigkeiten in einem Takt zuweisen, somit wird wesentlich häufiger die komplette Breite des Frontends effektiv verwendet. Analog zu superskalaren in-order Prozessoren kann die Platzierung von Befehlen auf den Ausführungseinheiten bzw. dem Array nur durch die Reihenfolge der Befehle im Befehlsstrom beeinflusst werden.

Auch VLIW Architekturen zeigen Ähnlichkeiten zum GAP. VLIW steht für *very long instruction word*, zu Deutsch „sehr langes Befehlswort“. Dazu werden mehrere Befehle aus dem Befehlsstrom zu einem „großen Befehl“, dem Befehlswort, zusammengefasst. Alle Befehle in einem Befehlswort können sicher in einem Takt zugeordnet werden, sie müssen dazu unabhängig voneinander sein. Bei einer VLIW Architektur mit in-order Zuordnung muss deshalb keine Prüfung auf Abhängigkeiten in der Zuordnungsstufe stattfinden, da dies der Compiler übernehmen muss. Wird aus den Befehlsworten wieder ein normaler Befehlsstrom gebildet, so wäre er auf dem GAP auch ausführbar. Dadurch lassen sich viele der Code-Optimierungs-Techniken, die für VLIW-Prozessoren entworfen wurden, auf den GAP übertragen. Für weitere Informationen zu VLIW-Architekturen wird wieder auf das Buch [14] verwiesen.

2.2. Programmierung des Grid-ALU-Prozessors

In diesem Kapitel wird darauf eingegangen, wie der Grid-ALU-Prozessor programmiert werden kann. Neben dem Befehlssatz und dem Compiler werden dazu Möglichkeiten vorgestellt, auf die Programmstruktur und somit die Programmausführung Einfluss zu nehmen.

2.2.1. Befehlssatz und Speicherstruktur

Die Grundlage für die Programmierung des GAP stellt der PISA-Befehlssatz ((*Portable Instruction Set Architecture*)) dar, das ist ein Befehlssatz für einen RISC Computer. Er

entstand durch Vereinfachung eines MIPS-Befehlssatzes (siehe z. B. [125]). Der PISA-Befehlssatz wurde mit und für den SimpleScalar-Simulator vorgestellt und ist im Dokument [16] genau erklärt.

Da GAP und SimpleScalar denselben Befehlssatz verwenden, können Programme ohne Modifikation von beiden Prozessoren ausgeführt werden. Sie sind binärkompatibel. Allerdings unterstützt der GAP anders als der SimpleScalar-Simulator keine Double- oder Float-Befehle.

Ein Befehl besteht prinzipiell aus acht Bytes, also 64 Bits. Beim Design des Befehlssatzes wurde nicht darauf geachtet, Befehle möglichst kompakt darstellen zu können, sondern es stand ganz klar die Eignung für die Forschung im Vordergrund. Deshalb wurden zwei Bytes statt dem tatsächlich verwendeten einen Byte für den OpCode (Befehlstyp) vorgesehen und es wurden zwei weitere Bytes hinzugefügt, die für eigene Markierungen verwendet werden können. Da Befehle nur in drei möglichen Formaten dargestellt werden können, ist das Prozessor-Frontend vergleichsweise einfach aufgebaut. In Abbildung 2.8 ist ein Beispiel für eine Addition im PISA-Befehlssatz zu sehen.

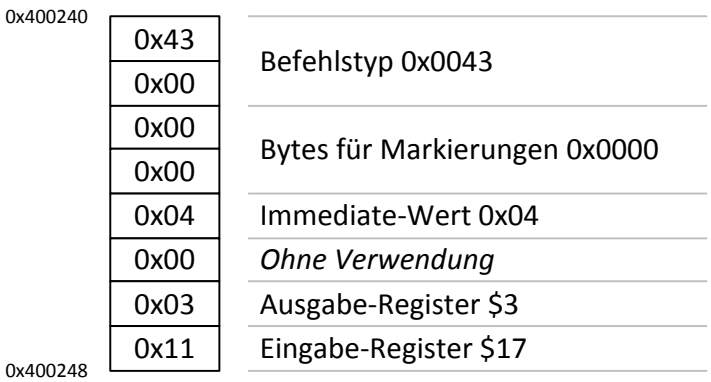


Abbildung 2.8.: Befehl `addiu $3,$17,0x04` im Speicherabbild

Der Befehlssatz unterstützt 31 Bits für die Speicheradressierung. Tabelle 2.1 zeigt die Verwendung des Arbeitsspeichers. Da der GAP momentan nicht über einen eigenen Loader verfügt, wird der SimpleScalar Simulator dafür verwendet. Dazu wird der Programmcode in den Speicher geschrieben und dann ein Speicherabbild ausgegeben, das dann sehr einfach vom GAP geladen werden kann. Um den Zustand des Prozessors vollständig zu beschreiben, werden außerdem der aktuelle Programmzähler und die Belegung der Register ausgegeben.

2.2.2. Compiler und Standardbibliothek: GCC und glibc

Jeder Prozessor kann nur dann sinnvoll verwendet werden, wenn ein passender Compiler und eine passende Standardbibliothek vorhanden sind. Der Compiler wird verwendet, um Programme aus höheren Programmiersprachen, z. B. C, C++ oder Java, in Binärcode zu übersetzen. In der Standardbibliothek sind Funktionen vorhanden, die

Speicheradresse	Programmteil	Beschreibung
0x00000000		ohne Verwendung
0x00400000	`.program`	Programmcode
0x10000000	`.data`	Konstanten etc.
0x7FFFFFFF		Stack (absteigend)

Tabelle 2.1.: Verwendung des Adressbereichs in GAP und SimpleScalar

häufig verwendet werden und deshalb allgemein bereitgestellt werden, z. B. Funktionen für die dynamische Speicherverwaltung (`malloc/free`).

Einen Compiler für einen komplett neuen Befehlssatz anzupassen ist kein einfaches Unterfangen. Da der GAP denselben Befehlssatz wie der SimpleScalar Simulator verwendet, können sowohl Compiler wie auch Standardbibliothek übernommen werden. Zum Einsatz kommen *GCC 2.7*[3] (erschienen im August 1997) als Compiler und *glibc 1.09.1*[4] als Standardbibliothek. Beide Komponenten sind bereits relativ alt, allerdings sind keine neueren Versionen für den PISA-Befehlssatz verfügbar. Das ist aber nicht als Schwierigkeit zu sehen, da der GAP für die Ausführung von Legacy-Code gedacht ist und somit damit gerechnet werden muss, dass nicht die neueste Software-Infrastruktur verwendet wird um Programme zu erstellen. Die gesamte Programmkette zur Code-Erzeugung ist in Abbildung 2.9 zu sehen.

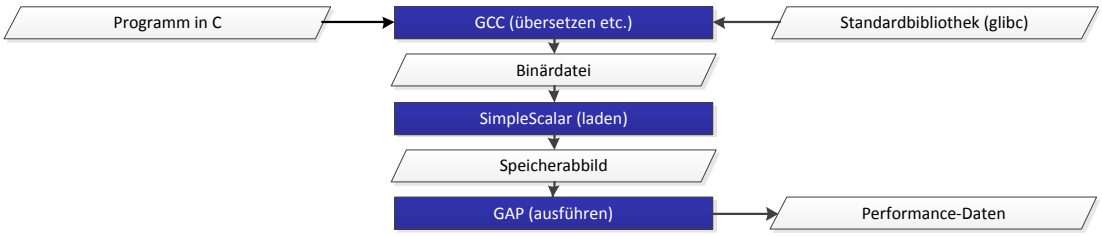


Abbildung 2.9.: Programmkette zur Erzeugung und Ausführung von Programmen auf dem GAP

Mit dem GCC in der verwendeten Version kann derzeit Programmcode in ANSI C oder Assembler übersetzt werden. Für die Ausführung durch den GAP müssen die Programme (wie bei eingebetteten Systemen üblich) statisch gelinkt werden, es muss also das Programm zusammen mit allen abhängigen Bibliotheken zu einer Programmdatei zusammengefasst werden.

Die glibc wurde so erweitert, dass alle Gleitkomma-Berechnungen mit Ganzzahl-Berechnungen emuliert werden. Dazu muss mit dem Parameter `-msoft-float` kompiliert werden. Somit kann auf dem GAP trotz Fehlen einer Hardware-Einheit für die Gleitkomma-Berechnungen mit allen normalerweise verwendeten Datentypen gerechnet werden.

[3] Siehe `http://gcc.gnu.org/`
[4] Siehe `http://www.gnu.org/software/libc/`

Dennoch ist die Software-Emulation relativ langsam. Für ein Beispielprogramm wurde auf dem SimpleScalar-Simulator für die Software-Implementierung im Vergleich zur Verwendung einer Gleitkomma-Einheit pro `float`-Instruktion der folgende durchschnittliche Zusatzaufwand pro Gleitkomma-Befehl ermittelt:

Funktionsaufrufe	$\approx 2,3$
Instruktionen	≈ 156
Takte	≈ 171

Die zusätzlich benötigten ca. 171 Takte pro Gleitkomma-Befehl sind sicherlich nicht „konkurrenzfähig“, allerdings steigt die Zahl der dann ausführbaren Programme stark, da oft nur wenige Gleitkomma-Befehle in Programmen enthalten sind und sie sonst überhaupt nicht ausgeführt werden könnten (Beispiel: SPEC INT 1995). In einem Vergleich würden alle Prozessoren mit emulierten Gleitkomma-Befehlen arbeiten, somit sind Vergleiche wieder fair.

2.2.3. Beeinflussung der Programmausführung im GAP

Anders als viele verwandte Architekturen (siehe Abschnitt 2.1.3), bei denen die Platzierung von Instruktionen oder Prozessen auch durch den Compiler oder sogar den Programmierer beeinflusst werden kann, sind beim GAP die Einflussmöglichkeiten relativ eingeschränkt. Grund dafür ist, dass der Compiler Code erzeugt, der so auch von einem superskalaren Prozessor ausgeführt werden kann. Deshalb gibt es die gleichen Ansatzpunkte zur Performance-Steigerung wie bei superskalaren Prozessoren:

Manuelle Software-Optimierung Sind bei einem Programm bestimmte Eigenschaften (vgl. Abschnitt 2.3) besonders erwünscht oder zu vermeiden, so ist es sinnvoll, schon den Entwickler des Programms darauf hinzuweisen. Ein Beispiel für derartige *Coding Standards* ist das Dokument [98]. Dazu gehört auch das Setzen von Markierungen, sog. Pragmas (Beispiel: OpenMP).

Automatische Software-Optimierung Möchte man das Programm, das ausgeführt werden soll, automatisch modifizieren lassen, sodass es bestimmte Eigenschaften stärker ausgeprägt aufweist, so kann das vor, während oder nach der Übersetzung durch einen Compiler geschehen.

Um das Programm in der Hochsprache, also vor der Übersetzung, zu verändern, werden *Source-to-Source-Compiler* verwendet. Ein Beispiel dafür ist ROSE [128] für die Programmiersprachen C und C++.

Am häufigsten finden Optimierungen bei der Übersetzung statt, also im Compiler. Es handelt sich dann um „klassische“ Compileroptimierungen. Optimierende Compiler sind beispielsweise GCC, ICC, Open64 und LLVM. Sie verfügen über sehr viel Wissen über das Programm und können es deshalb sehr effektiv beeinflussen.

Hat man das Programm schon in Maschinencode vorliegen, so können Optimierungen noch mit einem *Post-Link-Optimizer* vorgenommen werden. Der Vorteil

ist dabei, dass der Quellcode des Programms nicht vorliegen muss. Allerdings sind die Einflussmöglichkeiten nicht so groß wie bei Optimierungen im Compiler.

Hardware-Modifikationen Natürlich kann die Programmausführung auch verändert werden, indem an der Hardware, also dem Prozessor selbst, Änderungen durchgeführt werden. Ein Beispiel ist die Verwendung einer anderen Ersetzungsstrategie in einem Zwischenspeicher (vgl. [83]) oder einer anderen Technik zur Sprungvorhersage (vgl. [171]).

Kombinationen aus Hardware- und Software-Modifikationen Oft findet man Methoden, bei denen manuelle oder automatische Softwaremodifikationen mit Hardware-Modifikationen kombiniert werden. Ein Beispiel dafür ist *Predicated Execution* [122].

Wesentliche Beeinflussungsmöglichkeiten sind die Vergabe der Architekturregister, das Scheduling der Instruktionen, das Vervielfältigen von Instruktionen und die Auswahl der Instruktionstypen. Es ist für den GAP nicht wie bei anderen Architekturen möglich, Instruktionen direkt Funktionseinheiten zuzuweisen. Es ist auch nicht möglich, mehrere Funktionseinheiten zusammenzufassen und sie parallel arbeiten zu lassen.

Die Ausführung im GAP kann man sich vorstellen wie eine Feuersbrunst bei einem Waldbrand; sie beginnt im Array oben und setzt sich nach unten fort. Erst wenn sie das untere Ende erreicht hat startet sie wieder oben. Andere Prozessoren zeigen mehr Parallelität, dort würde das gesamte Array auf einmal „brennen“, also Instruktionen abarbeiten.

Für den GAP bieten sich vor allem ein Post-Link-Optimizer in Verbindung mit Hardware-Modifikationen zur Unterstützung der Code-Optimierung bzw. Performance-Steigerung an. Grund dafür ist, dass davon ausgegangen wird, dass der Quellcode der zu optimierenden Anwendung nicht verfügbar ist.

2.3. Performance-Analyse

In Abschnitt 2.3.1 wird Performance definiert und ihre Messung erklärt. Anhand eines theoretischen Beispiels wird anschließend in Abschnitt 2.3.2 gezeigt, welche maximale Performance der GAP im besten Fall erreichen könnte. In der Praxis können diese Werte leider nicht erzielt werden. Grund dafür sind Verzögerungen, die im Prozessor auftreten. Sie werden in Abschnitt 2.3.3 erörtert und mit Programmeigenschaften, die sie verursachen können, in Verbindung gebracht.

2.3.1. Performance-Definition und Grundlagen der Messung

Performance wird im Rahmen dieser Arbeit wie folgt definiert:

Definition 1. *Performance gibt an, wie schnell ein System (bzw. ein Prozessor) mit einer klar definierten Hardwarekonfiguration ein einzelnes oder mehrere vorgegebene Referenz-Programme ausführen kann.*

Diese Definition legt die Leistungsfähigkeit bzw. Performance ausschließlich über die Ausführungsgeschwindigkeit fest und unterscheidet sich damit von anderen Definitionen, die z. B. auch den Stromverbrauch mit einbeziehen. Der Stromverbrauch ist v. a. bei eingebetteten Systemen wichtig, da hier versucht werden muss, möglichst viel Arbeit pro Watt (also verfügbarer Energie) zu erledigen. Energiebetrachtungen bleiben beim GAP, einem Prozessor für High-Performance-Systeme (siehe Abschnitt 2.1) aber außen vor. Es wird nicht davon ausgegangen, dass der Stromverbauch oder andere Faktoren begrenzend sind.

Die Messung der Performance eines Systems findet durch Ausführung vorgegebener Programme statt. Diese Programme sind soweit fixiert, dass unterschiedliche Implementierungsansätze keinen Einfluss mehr haben, da dann das Programm per Definition nicht mehr dasselbe Programm wäre. Diese vorgegebenen Programme werden Benchmarks genannt. Sie liegen beim GAP entweder als Assembler-Code oder in ANSI C vor.

Der GAP kann Gleitkomma-Befehle nicht in Hardware ausführen. Deswegen ist es nicht sinnvoll, Benchmarks mit hohem Anteil an Gleitkomma-Befehlen zu wählen, da hiermit eher die Implementierung der Gleitkomma-Emulation getestet wird und die Programmstruktur, die ja von Benchmark zu Benchmark unterschiedlich ist, an Bedeutung verliert. Da der Einsatzbereich für den GAP nicht festgelegt ist, sollten Benchmarks aus möglichst vielen unterschiedlichen Gebieten gewählt werden.

Als Benchmark-Suites bieten sich deshalb die MiBench Benchmark Suite [63] und die Integer-basierten SPEC-Benchmarks an. Da der Simulator für den GAP relativ langsam ist (ca. 5.000 bis 35.000 Instruktionen pro Sekunde) dürfen die auszuführenden Programme nicht zu umfangreich sein, da sonst sehr lange Wartezeiten entstehen. Aus diesem Grund wird im Rahmen dieser Arbeit komplett auf die SPEC Benchmarks verzichtet, die Laufzeiten im Bereich von Tagen hätten. Stattdessen werden ca. 17 Benchmarks aus der MiBench Benchmark Suite verwendet, die hauptsächlich aus Integer-Befehlen bestehen. Wenn nötig wurden die Eingabedaten so verändert, dass die Benchmarks ca. 2.000.000 Befehle ausführen. Das ist ein legitimes Vorgehen, da die selben Benchmarks mit den gleichen Eingabedaten auch auf den Plattformen, mit denen verglichen wird, ausgeführt werden. Es findet kein Vergleich mit Performance-Kennzahlen statt, die von Dritten berechnet und bei denen andere Eingabedaten verwendet wurden (vgl. Artikel [74, 54]).

Die Performance des GAP wird durch Ausführung von Programmen mit dem taktgenauen und deterministischen Simulator für den GAP gemessen. Aus der berechneten gesamten Anzahl der benötigten Takte und der Anzahl an ausgeführten Instruktionen lässt sich die Maßzahl *IPC* (*instructions per clock cycle*) berechnen:

$$IPC := \frac{Instruktionen}{Takte}$$

2.3.2. Theoretisch maximale Performance des GAP

Der GAP verfügt über viele Funktionseinheiten. Um ein vollständig konfiguriertes Array mit 12 Zeilen und 12 Spalten komplett auszuführen werden bestenfalls ca. acht Takte benötigt, wobei das von der minimalen Dauer eines Befehles im asynchronen Modell abhängt. In diesen acht Takten könnten auch acht Speicherzugriffe ausgeführt werden. Somit wären in acht Takten maximal $4 + 11 * 12 = 136$ arithmetisch-logische Befehle und acht Speicherzugriffe ausführbar, also insgesamt maximal $136 + 8 = 144$ Befehle. Der maximal erreichbare IPC ist also $144/8 = 18$.

Leider werden in Simulationen wesentlich niedrigere Werte erreicht, Ergebnisse zeigt Abbildung 2.10. Die Gründe für die niedrigere beobachtete Performance sind vielfältig. Teilweise ist die begrenzte Verfügbarkeit an Hardware-Ressourcen hinderlich, teilweise sind Eigenschaften des Programms ursächlich.

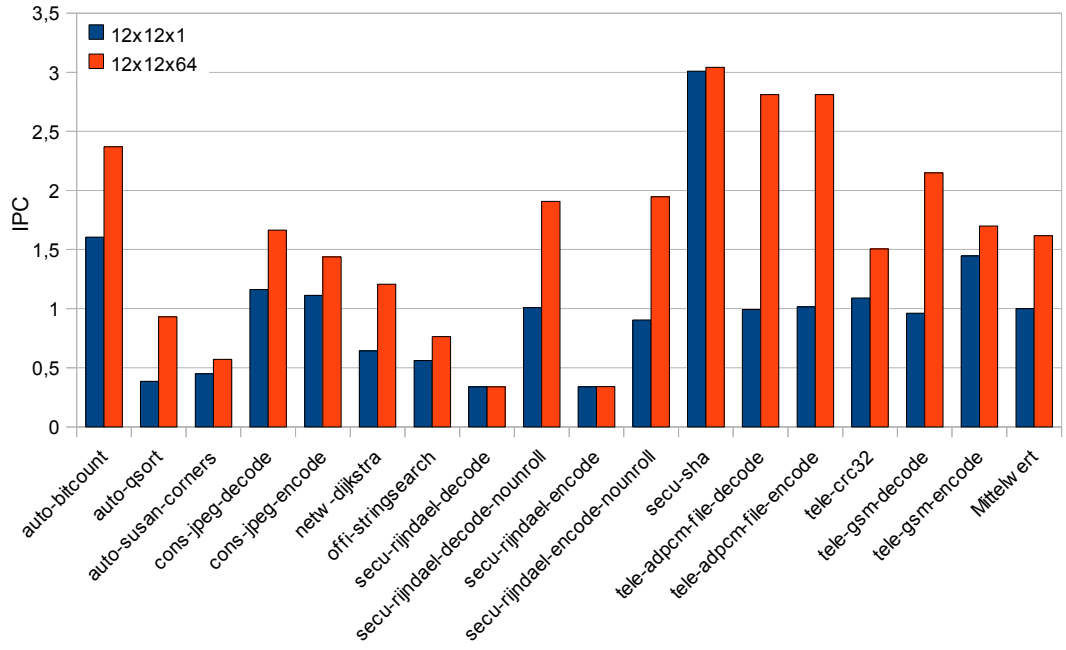


Abbildung 2.10.: IPC-Werte für 17 MiBench-Benchmarks ausgeführt auf GAP mit Array 12x12x1 bzw. 12x12x64 und 8kb Befehls-Cache in der Konfiguration 8x128x1

2.3.3. Einflüsse auf Performance und Ursachen

Die Performance des GAP ergibt sich daraus, wie gut die einzelnen Komponenten des Prozessors arbeiten. Die Performance der Komponenten wird wesentlich durch bestimmte Programmeigenschaften beeinflusst. Diesen Zusammenhang stellt Abbildung 2.11 beispielhaft dar.

Negative Effekte von Programmeigenschaften werden vor allem durch Verzögerun-

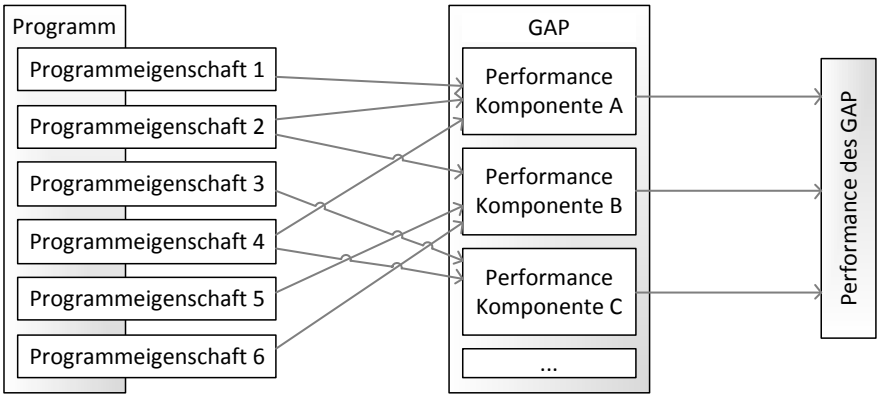


Abbildung 2.11.: Interaktion zwichen Programmeigenschaften (Ursachen), den beeinflussten Komponenten des GAP (Wirkung) und der Gesamtperformance

gen bei der Ausführung sichtbar. Die Gesamt-Performance und die einzelnen Verzögerungen können ermittelt werden, indem die bei der Ausführung des Programms auf dem GAP-Simulator erzeugte Ergebnis-Datei (siehe Anhang A.1) untersucht wird. Darin sind alle relevanten Performance-Counter enthalten.

Einen Überblick über die Beziehungen zwischen Programmeigenschaften und den Gründen für Verzögerungen bei der Ausführung auf dem GAP gibt Abbildung 2.12. Details werden im Folgenden beschrieben und gleichzeitig mit den entsprechenden Performance-Countern des GAP in Zusammenhang gebracht:

Befehls- und Datenzwischenspeicher sowie Frontend

Wird Programmcode auf das Array abgebildet, so muss er erst aus dem Arbeitsspeicher geladen werden. Dies kann der Befehlszwischenspeicher beschleunigen. Befindet sich der Code aber nicht im Zwischenspeicher (*instruction cache miss*), so muss auf den Arbeitsspeicher gewartet werden. Ähnlich ist es, wenn das Programm mit Load-Instruktionen Daten lädt. Sind sie nicht im Datenzwischenspeicher (*data cache miss*), so muss gewartet werden.

Im Allgemeinen gilt: Je mehr Instruktionen durch das Frontend des Prozessors geladen werden müssen, desto häufiger treten Fehlzugriffe im Befehlszwischenspeicher auf. Somit sind Code-Optimierungen, die die Code-Größe erhöhen, vorsichtig zu verwenden. Dazu zählen beispielsweise die Funktionseinbindung und das Abrollen von Schleifen.

Relevante Performance-Counter des GAP: Fehlzugriffe Befehlszwischenspeicher, Fehlzugriffe Datenzwischenspeicher, Anzahl der auf das Array abgebildeten Instruktionen, Anzahl der Takte mit inaktivem Frontend

Parallelität und Konfigurationshöhen

Bei der Beschreibung der Abbildung von Befehlen auf das Array (Kapitel 2.1.2) wurde bereits erwähnt, dass die Datenabhängigkeiten unter den Befehlen einen

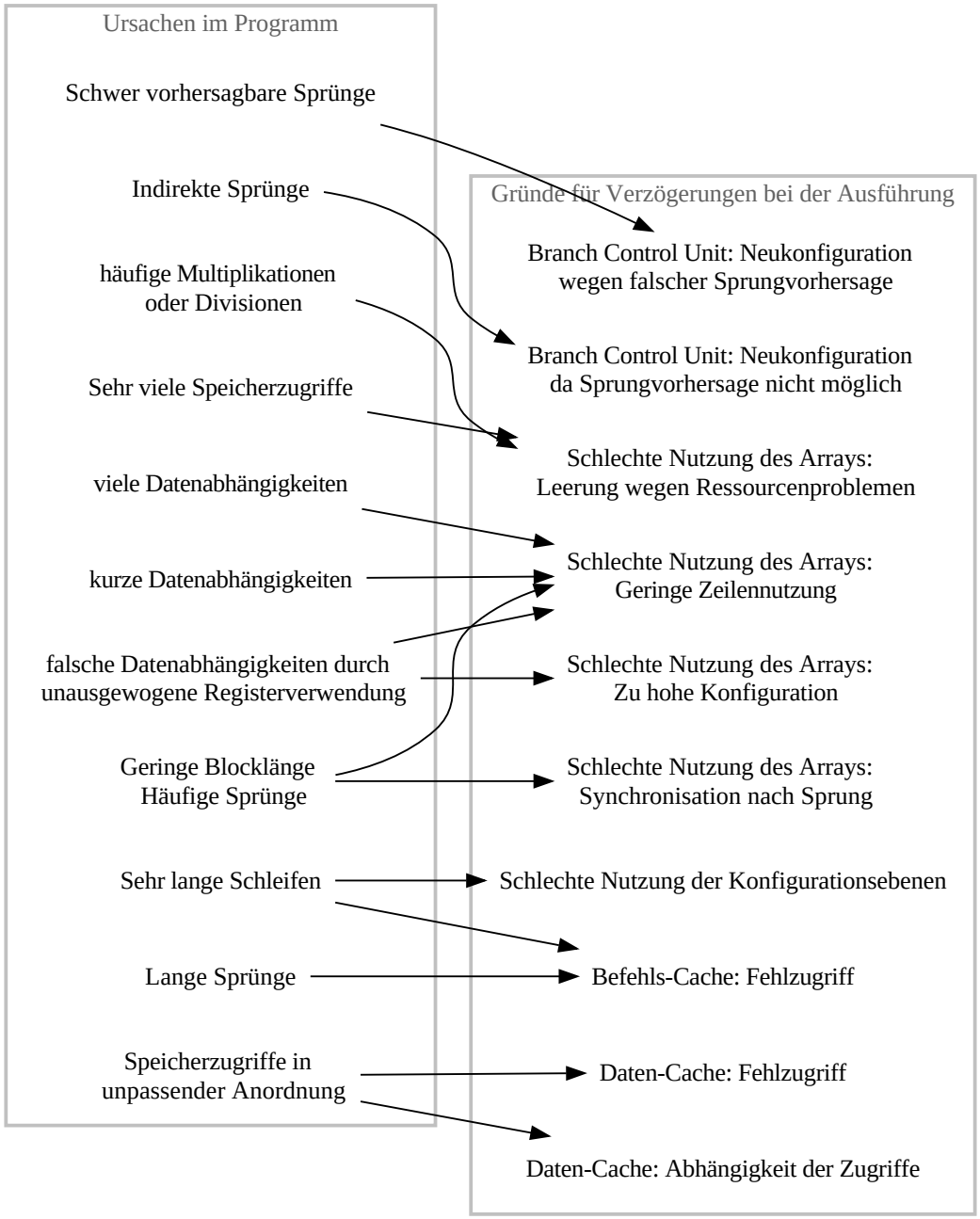


Abbildung 2.12.: Gründe für Verzögerungen bei der Programmausführung und mögliche Ursachen im auszuführenden Programm

sehr hohen Einfluss haben. Das führt dazu, dass selten ein großer Teil der Funktionseinheiten pro Zeile tatsächlich verwendet wird. Im besten Fall sollten die Befehle über sehr wenige Datenabhängigkeiten verfügen oder sie sollten sehr lange sein, also viele Befehle überspannen. Leider ist dieser Wunsch utopisch, da zumindest echte Datenabhängigkeiten die Grundlage für die sinnvolle Verkettung von Befehlen in superskalaren und ähnlichen Prozessoren bilden. Genau aus diesem Grund sind sie meist auch sehr kurz. Falsche Datenabhängigkeiten könnten teilweise reduziert werden, wobei hier oft „parallele“ echte Datenabhängigkeiten dominierend sind.

Außerdem muss, falls die Konfiguration zu hoch für das Array ist, ein Umbruch stattfinden und der Rest der Konfiguration auf eine andere Konfigurationsebene geschrieben werden. Bei jedem Wechsel der Konfigurationsebene (soweit vorhanden, sonst muss das Array komplett neu konfiguriert werden) muss wieder eine kurze Zeit gewartet werden, bis alle Funktionseinheiten benachrichtigt und umgeschaltet sind.

Relevante Performance-Counter des GAP: Anzahl der verwendeten Funktionseinheiten pro Zeile oder Konfiguration, Anzahl der verwendeten Spalten pro Zeile oder Konfiguration, Höhe der Konfigurationen

Branch Control Unit (BCU) und Synchronisation Sprünge im Programmcode haben einen großen Einfluss. Bei jedem Sprung muss eine Synchronisation zwischen den synchronen Einheiten (BCU, MAU) und den asynchron arbeitenden Funktionseinheiten stattfinden. Das erfordert, dass nach einem Sprung stets eine neue Zeile begonnen werden muss. Leider sind Sprünge sehr häufig, die Blocklänge – also (stark vereinfacht) die Anzahl der Instruktionen zwischen zwei Sprüngen – ist meist gering (siehe Grafik A.1 auf Seite 160 und Artikel von Lam und Wison [90]).

Darüber hinaus ist es (auch) für den GAP essentiell, auf eine gute Sprungvorhersagetechnik zurückgreifen zu können. Beim Erstellen von Konfigurationen wird bei jedem Sprung versucht, das wahrscheinlichste Ziel des Sprungs zu finden. Wird der Sprung falsch vorhergesagt (*branch prediction miss*), so kann die vorhergesagte Konfiguration nicht weiter verwendet werden und es muss das Frontend des Prozessors angestoßen werden, damit es eine neue Konfiguration mit den gewünschten Instruktionen erstellt. Das kostet zusätzliche Zeit (vgl. Kapitel 4.2).

Besonders schwierig ist die Sprungvorhersage bei indirekten Sprüngen, also Sprüngen, deren Ziel nicht eindeutig und im Voraus festgelegt ist.

Relevante Performance-Counter des GAP: Anzahl falsch vorhergesagter Sprünge, Anzahl insgesamt ausgeführter Sprünge, Anzahl der Sprünge pro Konfiguration

Verwendung der Konfigurationsebenen

Die Steuerung der Konfigurationsebenen kann zu Performance-Einbußen führen, wenn Konfigurationen, die eigentlich bald wieder benötigt werden, mit an-

deren Daten überschrieben werden. Dann muss dieser Code neu geladen werden. Ein Lösungsansatz wird in Kapitel 4.5 vorgestellt.

Relevante Performance-Counter des GAP: Gesamtanzahl schon geladener Konfigurationen (Schleife/Ebenen), Gesamtanzahl neu zu ladender Konfigurationen

Gesamtperformance

Die Gesamt-Performance des GAP wird durch die Anzahl benötigter Takte und die Anzahl der ausgeführten Instruktionen bestimmt (vgl. Abschnitt 2.3.1).

Als verwandte Arbeit ist z. B. der Artikel von Yi et al. [172] zu nennen; die Autoren stellen ein Verfahren vor, mit dem automatisch ein Zusammenhang zwischen Programmeigenschaften und Performance-Countern ermittelt werden kann. Diese Kennzahlen könnten verwendet werden, um eine Formel zu entwickeln, mit der die Performance des Gesamtsystems vorhergesagt werden kann. Man spricht von *Performance-Prediction* (vgl. [73],[118]).

Ein Ziel dieser Arbeit ist es, die negativen Einflüsse auf die Performance des GAP zu reduzieren, indem auszuführende Programm durch Code-Transformationen aufbereitet wird. Das Werkzeug dafür ist GAPtimize (siehe Kapitel 3), die verwendeten Methoden werden in Kapitel 4 erklärt und in Kapitel 5 abschließend evaluiert.

3. Post-Link-Optimizer GAPtimize

GAPtimize ist der Post-Link-Optimizer für den GAP. Nach einer kurzen Einführung von GAPtimize werden verwandte Ansätze und Projekte vorgestellt und abgegrenzt. Es folgt die Erklärung der Abläufe und Strukturen in GAPtimize.

An mehreren Stellen in diesem und in den folgenden Kapiteln werden Konzepte aus dem Compilerbau verwendet. Für weitere Informationen dazu wird auf die Bücher von Cooper und Torczon [42] sowie Aho et al. [2] verwiesen.

3.1. Einführung

Ein wesentlicher Vorteil des GAP ist, dass Legacy-Code ausgeführt werden kann. Das bedeutet, dass vom auszuführenden Programm kein Quellcode sondern nur die kompilierte Datei vorhanden sein muss. Um in dieser Situation Nutzen aus plattformspezifischen Optimierungen ziehen zu können empfiehlt es sich, einen Post-Link-Optimierer zu verwenden (siehe Kapitel 2.2.3). Er analysiert das kompilierte Programm, führt Veränderungen durch und erstellt eine neue Version mit kompiliertem Code.

Für diese Art von Software gibt es keine Implementierung, die bekannt ist und die sich einfach auf jede Situation anpassen ließe (also keine „Standardlösung“). Für den GAP wurde daher GAPtimize entwickelt. Zweck des Tools ist es, die Anwendung von allgemeinen und plattformspezifischen Code-Optimierungen zu ermöglichen. Allgemeine Code-Optimierungen sind nicht speziell auf den GAP zugeschnitten, sondern führen auch bei anderen Prozessoren zu Performance-Gewinnen. Plattformspezifische Optimierungen wurden speziell für den GAP entwickelt und führen entweder durch die Nutzung spezieller Eigenschaften des GAP zu Vorteilen oder sind nur mit spezieller Hardware-Unterstützung verwendbar.

Optimierungen können mit verschiedenen Anwendungsbereichen (*Scopes*) realisiert werden. Das Maximum ist das ganze Programm, d. h. es werden Informationen über das gesamte Programm herangezogen um Optimierungen durchzuführen. Diese Optimierungen werden *whole program optimizations* genannt. Beim Design von GAPtimize wurde besonderes Augenmerk darauf gelegt, möglichst wenig Einschränkungen für Analysen und Optimierungen zu schaffen. Analysen wurden so implementiert, dass sie einen möglichst großen Bereich des Programms abdecken.

In Abbildung 3.1 ist zu sehen, wie GAPtimize in den Prozess zum Vorbereiten eines Programms zur Ausführung auf dem GAP eingebunden wird. Das Programm muss erst ganz regulär mit einem Compiler, z. B. dem GCC, übersetzt werden. Dann wird es, da der GAP derzeit keinen Loader besitzt und deshalb nur vorbereitete Speicherabbilder ausführen kann, mit einer modifizierten Version des SimpleScalar-Simulators

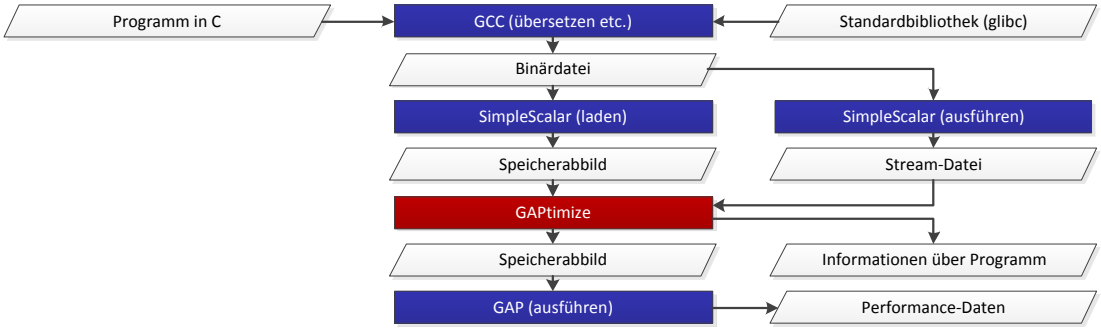


Abbildung 3.1.: Ablauf vom Programm im Quellcode bis zur Ausführung auf dem GAP mit optionaler Optimierung durch GAPtimize (vgl. Abb. 2.9, Seite 28)

in ein solches umgewandelt. Es kann dann mit GAPtimize eingelesen, verarbeitet und anschließend wieder ausgegeben werden. Da GAPtimize ein optionaler Schritt ist, könnte das Abbild auch direkt ausgeführt werden. Ein Speicherabbild besteht aus Binärdateien, die die einzelnen Befehle des Programms und den Daten-Bereich genau so beinhalten, wie er bei der Ausführung im Speicher des Prozessors vorhanden ist.

GAPtimize kann als Eingabe auch weitere Informationen über die Programmausführung verwenden (Stream-Datei), sie werden durch Ausführung des unveränderten Programms auf dem GAP oder SimpleScalar-Simulator gewonnen. So stehen weitere Informationen zur Verfügung, die in traditionellen Compilern nicht vorhanden sind, da zum Zeitpunkt der Optimierung üblicherweise noch keine Informationen über das Laufzeitverhalten verfügbar sind. Optimierungen, die diese Informationen verwenden können, werden als *feedback directed optimizations* bezeichnet. Dadurch kann eine Feedback-Schleife (Regelkreis) gebildet werden.

Somit sind beispielsweise adaptive Optimierungen (vgl. Kapitel 5) möglich, bei denen durch zusätzliche Steuerkomponenten die Code-Optimierungen so variiert oder konfiguriert werden, dass damit gute Ergebnisse in der jeweiligen Situation (Hardware, Software) erzielt werden können.

Neben der Code-Optimierung sind die Erzeugung von Statistiken über das Programm (vgl. Abbildung A.1 auf Seite 160) und die Darstellung des Programms in einer lesbaren bzw. schnell erfassbaren Form weitere Einsatzgebiete, die GAPtimize abdeckt. Somit fördert es das Verständnis für die Performance von Programmen und ermöglicht, die Ursachen für besonders gute oder schlechte Performance bei der Ausführung auf dem GAP zu analysieren.

Der vom GCC erzeugte Code, der mit GAPtimize optimiert wird, ist nicht per se „schlecht“, er nutzt aber Eigenschaften des GAP nicht aus bzw. berücksichtigt einige Schwächen des GAP nicht im erforderlichen Maß, sodass es hier Optimierungsmöglichkeiten gibt. Dieses Potenzial soll mit GAPtimize so gut wie möglich ausgeschöpft werden. Es findet aber keine „Übersetzung“ für die GAP-Architektur statt. Deshalb ist die Optimierung mit GAPtimize optional.

3.2. Verwandte Projekte

GAPtimize ist das Werkzeug, mit dem Optimierungen für den GAP entwickelt und getestet werden können. Ziel dieser Optimierungen ist es, die Performance von Programmen (siehe Kapitel 2.3.1) zu erhöhen. Dazu werden statische Post-Link-Optimierungen durchgeführt. Programme mit ähnlicher Zielsetzung sind die Folgenden:

FDPR Mit FDPR[1] (siehe z.B. [64]) hat IBM für mehrere Plattformen (AIX/Power, Linux/Power, Cell BE, Windows/IA32 etc.) ein Werkzeug zur Post-Link-Optimierung vorgestellt. Dazu wird zuerst die auszuführende Datei des zu optimierenden Programms so erweitert, dass bei der Ausführung dieser Datei zusätzliche Daten über die Ausführung gesammelt werden. Dieser Prozess heißt *Instrumentation*. Diese Version des Programms wird dann ausgeführt und es werden Daten über das Laufzeitverhalten aufgezeichnet. Mit diesen Daten wird dann eine optimierte Version des Programms erzeugt.

Die wesentlichen Optimierungen sind „inter-procedural register re-allocation, improving instruction scheduling, data prefetching, function inlining, global data reordering, global code reordering, along with various tuning options for the Power architecture“. FDPR ist allgemein verfügbar und wird, beispielsweise für die Cell Architektur, auch im produktiven Betrieb eingesetzt.

SOLAR project: ALTO, PLTO, ILTO Im Rahmen des SOLAR-Projekts[2] werden „Techniken für flexible Code-Optimierungen zur Link- und Laufzeit“ entwickelt. Ziel ist die Optimierung der Ausführungszeit, der Programmgröße und von Security-Eigenschaften. Teilprojekte sind ALTO [116, 117] für Alpha-Architekturen, PLTO [138] für IA32/Pentium und ILTO [146] für Itanium.

Diablo [46] Diablo[3] ist ein optimierender *Linker*, es ist also für den Prozess des Zusammenfügens von Objekt-Dateien zum kompletten ausführbaren Programm verantwortlich. Es wurde ursprünglich entwickelt um Programme zu beschleunigen. Allerdings stellte sich bald heraus, dass auf Standard-Plattformen das durch optimierende Compiler ungenutzte Potenzial relativ gering ist. Deshalb wurde bald die Optimierung der Code-Größe, die v. a. bei eingebetteten Systemen eine sehr wichtige Rolle spielt, in den Vordergrund gestellt. Als Nebeneffekte ergeben sich dadurch geringe Geschwindigkeitsvorteile.

Diablo unterstützt auch die Analyse von Code. Es wurde auf verschiedene Plattformen portiert (z. B. Itanium [8]); derzeit werden ARM [45], x86 und PowerPC unterstützt.

Spike [36, 101] Ein weiterer Post-Link-Optimierer ist Spike, der von HP Tru64 Unix entwickelt wurde. Spike kann Informationen über die Programmausführung

[1] `https://www.research.ibm.com/haifa/projects/systems/cot/fdpr/index.html`
[2] `http://www.cs.arizona.edu/solar/`
[3] `http://diablo.elis.ugent.be/publications`

verwenden und somit bessere Optimierungsentscheidungen treffen.

Außerdem werden Optimierungen zur Link-Zeit auch von den Compilern GCC[4] ab Version 4.5.0 (erschienen am 14.04.2010) und von LLVM[5] unterstützt. Eine Übersicht über weitere Link-Zeit- und Post-Link-Optimierer ist auf der Internetseite von Diablo[6] zu finden.

Darüber hinaus gibt es Tools, die Programme während ihrer Ausführung modifizieren und somit *dynamische Optimierungen* vornehmen können (z. B. [15], [103]). GAPtimize kann das nicht, es verfolgt (wie die oben genannten Programme) ein statisches Vorgehen.

GAPtimize ist das einzige Programm, das für den GAP mit dem PISA-Befehlssatz verfügbar ist. Alle anderen hätten erst mit hohem Aufwand portiert werden müssen. Zudem ist fraglich, inwieweit bei ihnen Erweiterbarkeit gegeben ist, die für die Implementierung und Evaluierung plattformspezifischer Optimierungen unabdingbar ist.

Mit adaptiver Kompilierung, die Feedback über die Ausführung des zu optimierenden Programms verwendet, haben sich im Wesentlichen Keith D. Cooper und Lisa Czornton [41, 38, 43] beschäftigt. Einen Schwerpunkt bilden dabei einzelne Optimierungen, z. B. die Einbindung von Funktionen (siehe Kapitel 4.3) und die Suche nach einer optimalen Anordnung von Optimierungen (*Phase Ordering Problem*, Details und Referenzen siehe Kapitel 5.3).

3.3. Struktur von GAPtimize

Beim Entwurf von GAPtimize standen Erweiterbarkeit und Klarheit im Vordergrund. Entsprechend den verschiedenen Use-Cases (Optimierung und Analyse) und den benötigten Schnittstellen (Kommandozeile, Java-GUI, Skript-Schnittstelle) wurden mehrere Module angelegt (vgl. Grafik 3.2):

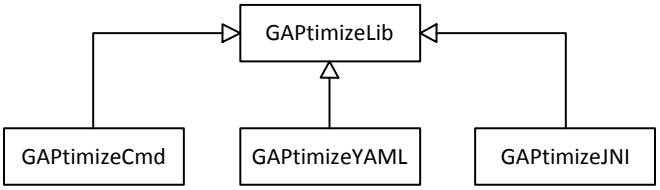


Abbildung 3.2.: Modulstruktur von GAPtimize.

- Die Bibliothek **GAPtimizeLib** ist die gemeinsame Code-Basis aller Frontends. Es enthält die Systemoperationen, also z. B. alle Funktionen zum Laden, Speichern, Analysieren und Optimieren von Programmen. Alle Frontends rufen Funktionen aus GAPtimizeLib auf.

[4] `http://gcc.gnu.org/wiki/LinkTimeOptimization`
[5] `http://llvm.org/docs/LinkTimeOptimization.html`
[6] `http://diablo.elis.ugent.be/?q=links`

- Das einfachste Frontend ist **GAPtimizeCmd**, das v. a. zum Testen von GAPtimize während der Entwicklung verwendet wird. Alle Parameter, z. B. das zu optimierende Programm und Analyseschritte, werden vor der Kompilierung von GAPtimizeCmd im Quellcode festgelegt.

- **GAPtimizeYAML** ist ähnlich wie GAPtimizeCmd aufgebaut, liest aber alle Parameter aus einer Datei im YAML-Format[7], einem lesbaren Format zur Datenserialisierung (ähnlich XML, Beispiel siehe Anhang A.3). YAML-Dateien können z. B. mit einer Java-Anwendung geschrieben werden. Somit kann GAPtimize einfach von anderen Programmen aus gestartet werden, diese Anbindung wird z. B. von FADSE (siehe Kapitel 5) verwendet.

- **GAPtimizeJNI** stellt die wesentlichen Funktionen von GAPtimize als DLL-Datei über das *Java Native Interface* für Java bereit. Dadurch kann innerhalb einer Java-Anwendung direkt auf Funktionen von GAPtimize zugegriffen werden. Diese Schnittstelle wird v. a. bei der Visualisierung des Kontrollflussgraphen stark genutzt; Abbildung 3.3 zeigt einen Screenshot.

Die in der gemeinsam verwendeten Bibliothek GAPtimizeLib vorhandenen Klassen lassen sich nach Funktion unterteilen. Die Datenklassen werden zur Repräsentation des Programms, das verändert oder analysiert werden soll, verwendet. Analysen sind teilweise direkt in den Datenklassen und großteils in separaten Klassen untergebracht. Ein weiterer Teil stellt Optimierungen zur Verfügung, wobei hier oft eine starke Verzahnung mit den Analysen vorhanden ist. Alle Module sind in C++ programmiert.

[7] `http://www.yaml.org/`

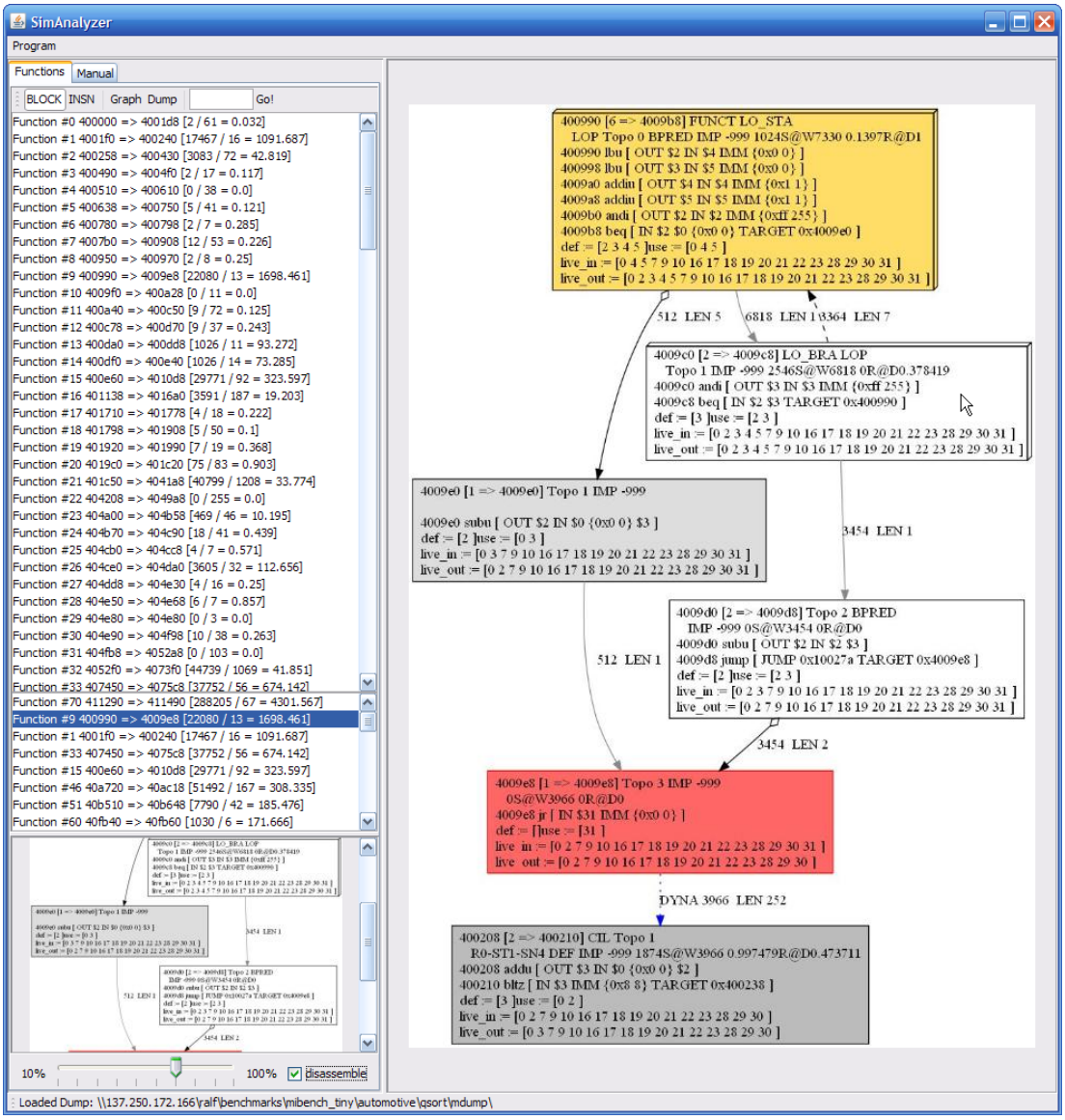


Abbildung 3.3.: Screenshot für die Visualisierung des Kontrollflussgraphen. Links werden alle Funktionen in zwei unterschiedlichen Sortierungen angezeigt, bei Klick auf eine wird rechts der mit den Ausführungshäufigkeiten annotierte Kontrollflussgraph auf Block-Ebene dargestellt. In den einzelnen Blöcken werden unter anderem die enthaltenen Instruktionen sowie Daten aus der Liveness-Analyse gezeigt.

3.4. Datenstrukturen zur Programmrepräsentation

Das Programm, das analysiert oder optimiert werden soll, wird in GAPtimize in bestimmten Datenstrukturen gespeichert, die in diesem Abschnitt vorgestellt werden.

Die wichtigsten Informationen über das Programm werden in einer Instanz der Klasse **program** abgelegt.

Die interne Repräsentation des Programmcodes basiert auf einer doppelt verketteten Liste. Sie besteht aus Instanzen der Klasse **instruction**. Jede Instanz repräsentiert genau eine Instruktion und hat je einen Zeiger zur nächsten und zur vorhergehenden Instruktion (siehe Abbildung 3.4(a)). Die Reihenfolge der Instruktionen in der Liste ist identisch zu ihrer Anordnung im Speicherabbild. Die Adressen der Instruktionen lassen sich somit berechnen. Das erste und letzte Glied der verketteten Liste sind in der **program**-Instanz hinterlegt.

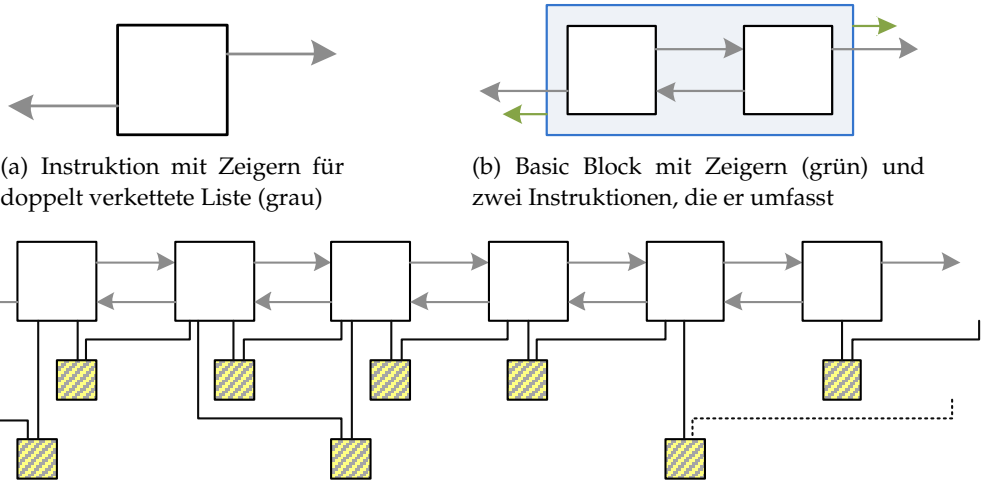
Um den Kontrollfluss abzubilden werden Instanzen der Klasse **cfgEdge<T>** verwendet. Eine Instruktion verfügt für jede mögliche nachfolgende Instruktion (also auch den Standard-Kontrollfluss) über je eine Instanz von **cfgEdge<T>**, die auf genau eine mögliche nachfolgende Instruktion verweist. Jede Instanz repräsentiert eine mögliche Kante im Kontrollflussgraphen (siehe Abbildung 3.4(c)). Die Kanten können wie folgt danach klassifiziert werden, ob es sich um einen Sprung (j) handelt und ob das Ziel spekulativ (s) ist:[8]

- **Standard-Kontrollfluss** Das Ziel der Kante steht fest, es ist die Nachfolgeinstruktion der aktuellen Instruktion. Um es zu erreichen muss kein Sprung ausgeführt werden. ($\neg j \wedge \neg s$)
- **Direkter Sprung** Die Start-Instruktion ist ein Sprung, das Ziel ist bekannt und berechenbar. ($j \wedge \neg s$)
- **Indirekter Sprung** Ausgangspunkt ist eine Sprung-Instruktion, allerdings wird das Ziel erst bei der Programmausführung berechnet. Informationen über mögliche Ziele werden durch teilweise Analyse der Funktionsaufrufe und durch Beobachten des Programmverhaltens bei einer Referenzausführung (Stream-Datei) ermittelt. ($j \wedge s$)

Ähnlich wie bei den Kanten des Kontrollflussgraphen werden die Datenabhängigkeiten zwischen den Befehlen modelliert. Die entsprechende Klasse heißt **dfgEdge<T>**. Datenabhängigkeiten verfügen neben Start und Ziel über eine Ressource, auf die sie sich beziehen (z. B. ein Register) und einen Typ (Gegen-, Ausgabe- oder echte Abhängigkeit).

Eine weitere sehr wichtige Datenstruktur sind die *Basic Blocks*. Ein Basic Block besteht aus einer oder mehreren Instruktionen. Er endet bei einer Instruktion A immer dann, wenn entweder A mehr als einen Nachfolger im Kontrollflussgraphen hat oder

[8] Die Kombination $j \wedge s$ ist nicht denkbar, da eine Instruktion, die kein Sprung ist, immer genau einen Nachfolger hat – es ist die nächste Instruktion.



(a) Instruktion mit Zeigern für doppelt verkettete Liste (grau)

(b) Basic Block mit Zeigern (grün) und zwei Instruktionen, die er umfasst

(c) Programmstück bestehend aus Instruktionen mit nebengeordneten Instanzen von **cfgEdge<T>** (schraffiert) zur Modellierung des Kontrollflusses (schwarz)

Abbildung 3.4.: In GAPtimize verwendete Datenstrukturen zur Repräsentation von Programmcode

die auf A folgende Instruktion $A_{succ} = B$ mehr als eine eingehende Kante im Kontrollflussgraphen besitzt, also von mehr als einer Instruktion aus erreicht werden kann. Da zwischen Basic Blocks genau wie zwischen Instruktionen Kontrollfluss- und Datenabhängigkeiten bestehen, werden die Blöcke analog zu Instruktionen als doppelt verkettete Liste modelliert. Die Klassen **cfgEdge<T>** und **dfgEdge<T>** werden wie oben erklärt verwendet. Ein Basic Block wird durch eine Instanz der Klasse **block** repräsentiert und ihm sind eine oder mehrere Instanzen der Klasse **instruction** zugewiesen (siehe Abbildung 3.4(b)).

Die Funktionstabelle mit Start und Ende jeder Funktion kann über die **program**-Instanz angesprochen werden.

Somit sind alle relevanten Arbeitsbereiche für Analysen und Optimierungen vorhanden.

3.5. Ablauf einer Optimierung

Das Modul GAPtimizeYAML steuert ähnlich wie GAPtimizeCmd den Ablauf einer Programmoptimierung. Dabei werden die in Abbildung 3.5 gezeigten Schritte ausgeführt. Sie lassen sich unterteilen in Ein-/Ausgabe-Aktionen, Analysen und Programmmodifikationen. Die folgenden Unter-Abschnitte erklären sie näher.

3.5.1. Einlesen und Abspeichern eines Programms

Wie bereits erwähnt kann GAPtimize derzeit ein Programm nur einlesen, wenn es bereits in ein Speicherabbild umgewandelt wurde. Es besteht dann aus mehreren Dateien. Davon werden alle Dateien mit Programm-Code (z. B. `p400.hex`, vgl. Tabelle 2.1)

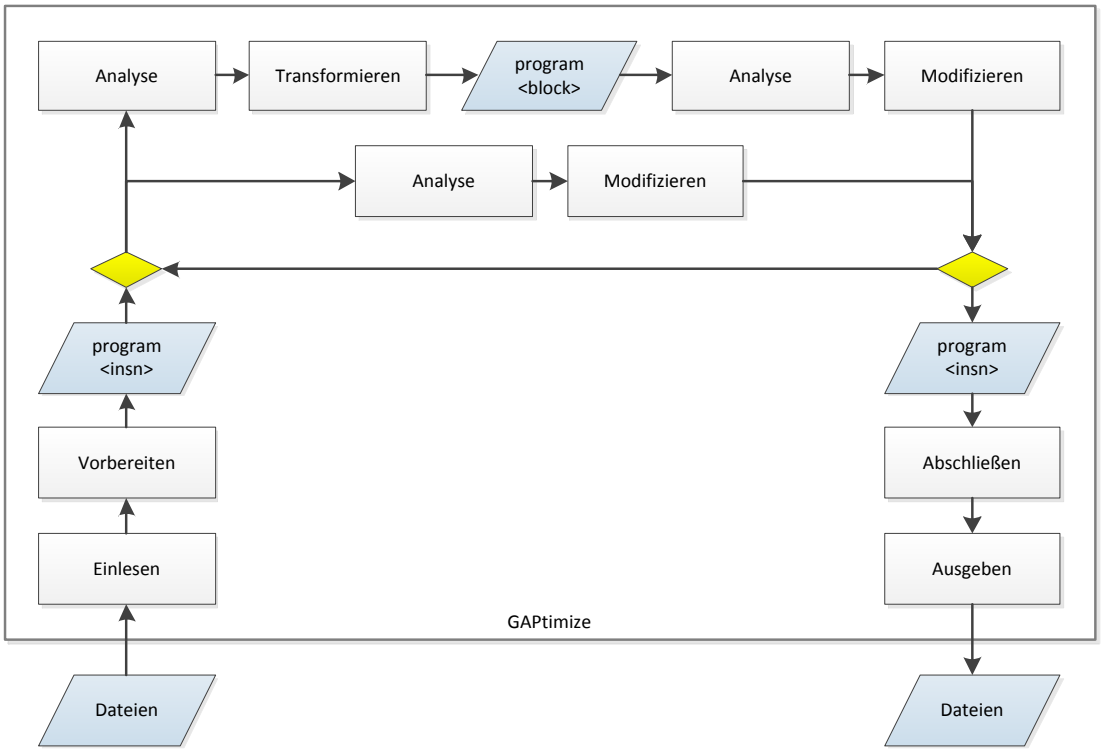


Abbildung 3.5.: Struktur einer Optimierung mit GAPtimize

der Reihe nach geöffnet. Da im PISA-Befehlssatz jeder Befehl gleich lang ist, nämlich 8 Byte, muss keine spezielle Analyse stattfinden um Grenzen von Instruktionen zu erkennen (vgl. [87], [137]), sondern es werden bis zum Dateiende immer acht Bytes gelesen und in je einer Instanz der Klasse **instruction** abgelegt. Auch die Speicher-Adresse, an der die Instruktion stand, wird in der Instruktion vermerkt.

Das Zurückschreiben des Programms in die Dateien des Speicherabbilds funktioniert analog. Es muss während der Modifikationen des Programms nur darauf geachtet werden, dass die Speicheradressen der Befehle (wie gewohnt) nacheinander liegen.

Außerdem verfügt GAPtimize über mehrere Funktionen, mit denen ein Programm oder ein Teil davon in lesbarer Form in eine Text-Datei geschrieben werden kann. Auf Wunsch können so beispielsweise Zwischenergebnisse zugänglich gemacht werden.

3.5.2. Vorbereiten der eingelesenen Instruktionen

Bevor weitere Analysen stattfinden muss eine Eigenheit der verwendeten Standardbibliothek entfernt werden. Standardmäßig wird bei einem schweren Fehler bei der Ausführung eines Systemaufrufs (z. B. Datei öffnen) mit einem unbedingten direkten Sprung (`jump`) zu einer Instruktion gesprungen, die außerhalb der Funktion liegt. Von dort wird eine Funktion aufgerufen, die die Fehlerbehandlung übernimmt. Dadurch wird aber die Regel verletzt, dass jede Funktion genau einen Einsprungpunkt und genau einen Endpunkt hat, der durch die Instruktion `jr $31` (unbedingter Sprung

zur Adresse in Register 31) markiert ist. Um das wiederherzustellen wird der Kontrollfluss etwas abgekürzt, sodass direkt zur Fehlerbehandlungsfunktion gesprungen werden kann ohne den Bereich der Funktion zu verlassen. Jetzt ist das Programm für weitere Analysen verwendbar.

3.5.3. Analyseschritte auf Ebene der Instruktionen

Es werden einige Analyseschritte ausgeführt, die Daten berechnen, die dann für Optimierungen verwendet werden können. Es ist hier leider nicht möglich, genaue Implementierungs-Details der Analyseschritte zu erklären. Meistens sind die Algorithmen bekannt. Einen Überblick über Code-Analysen geben Cooper und Torczon [42] sowie Aho et al. [2].

Rekonstruktion des Kontrollflusses und der Funktionstabelle

Als erste Analyse werden die Kanten des Kontrollflussgraphen berechnet. Da der PISA-Befehlssatz z. B. keine *delay slots* besitzt, treten hier viele Komplikationen, die bei anderen Befehlssätzen vorhanden wären, nicht auf (vgl. [39], [11]).

Es werden für jede Instruktion die berechenbaren ausgehenden Kanten angelegt, das sind die Kanten für den normalen Kontrollfluss und direkte Sprünge, d. h. Sprünge mit definiertem Ziel.

Jetzt kann die Funktionstabelle wiederhergestellt werden. Dabei wird davon ausgegangen, dass die erste Funktion nach ein paar Instruktionen zur Initialisierung an der Adresse 0x4001d8 beginnt. Jede Funktion besitzt genau einen Start- und genau einen Endpunkt. Eine Funktion wird immer mit `jr $31`, dem Funktionsrückspung, beendet. Nach einigen sog. *NOPs*, das sind Instruktionen, die nichts berechnen, wird dann mit der nächsten Instruktion, die wieder Berechnungen durchführt, die nächste Funktion begonnen (vgl. [66], [87]).

Anschließend werden für alle Funktionsrücksprünge spekulativ Kanten angelegt. Die Annahme, dass Funktionen wie gedacht aufgerufen werden und das Ziel des Rücksprungs sich immer genau eine Instruktion nach dem Funktionsaufruf befindet, hat sich bei den analysierten Benchmarks als korrekt erwiesen. Mit diesen Kanten ist jetzt eine Beziehung unter den Funktionen hergestellt und man kann von einem Startpunkt aus alle aufgerufenen Funktionen erreichen.

Leider ist damit aber nicht sichergestellt, dass wirklich alle Kanten des Kontrollflussgraphen erkannt wurden. Es gibt manchmal indirekte Sprünge, bei denen das Sprungziel erst zur Laufzeit ermittelt wird. Teilweise wird es aus dem Speicher geladen (*branch table*), teilweise wird es mit arithmetischen Befehlen berechnet, denkbar ist Vieles. Diese Kanten lassen sich nur mit enormem Aufwand komplett ermitteln (siehe beispielsweise [32], [149]). Um diese Kanten zumindest so gut wie möglich zur Verfügung zu stellen wird eine Datei mit allen ausgeführten Dateien des Programms eingelesen. Dadurch kann zum einen überprüft werden, welche Kanten nicht korrekt angelegt wurden, und zum anderen können für indirekte Sprünge weitere Kanten angelegt werden. Diese Kanten werden als *dynamische Kanten* separat markiert.

Diese Datei (Stream-Datei genannt) kann entweder direkt die Adressen der Befehle in ihrer Ausführungsreihenfolge enthalten oder schon vorverarbeitet sein, sodass sie Paare aus Adressen enthält und dazu die Übergangshäufigkeit sowie statistische Informationen. Diese Übergangshäufigkeiten werden als Kantengewichte für den Kontrollflussgraph verwendet. So stehen für weitere Analysen und Optimierungen Informationen über das Laufzeitverhalten zur Verfügung und es kann bei Optimierungen der Schwerpunkt auf die Teile des Programms gelegt werden, die häufig ausgeführt werden.

Liveness-Analyse auf Register-Ebene

Mit einer Liveness-Analyse kann ermittelt werden, welche Symbole an einem bestimmten Punkt im Programm verwendet werden (*use*) bzw. definiert (*define*) und aktiv ($live_{in}$ bzw. $live_{out}$) sind. Bei der Analyse auf Register-Ebene, wie sie in GAPtimize ausgeführt wird, wird berechnet, welche Register bei einer Instruktion *live* sind. Register sind genau dann *live*, wenn sie in mindestens einer der bei der Ausführung nachfolgenden Instruktionen gelesen werden. Mit diesen Informationen kann beispielsweise ermittelt werden, welche Register noch verfügbar sind für zusätzliche Berechnungen. Für weitere Informationen zur Liveness-Analyse wird auf die bereits erwähnten Bücher [42, 2] verwiesen. Mit der Liveness-Analyse auf Register-Ebene befassen sich z. B. Probst et al. [126] und Muth [115].

Die Liveness-Analyse wird in GAPtimize auf Ebene des Programms ausgeführt, Funktionsgrenzen werden also nicht beachtet. Da im Kontrollflussgraphen auch Kanten enthalten sind, die so evtl. nicht ausgeführt werden, ist die Menge der berechneten Live-Register im Zweifel eher zu groß als zu klein.

Bei der Analyse müssen viele Vereinigungen, Schnitte und Differenzen von Mengen mit Registern berechnet werden. Da die Berechnung mit „klassischen“ Mengen (**std::set**) sehr viel Zeit in Anspruch nahm, wurden die Register-Mengen stattdessen als 64-bit Zahlen modelliert, bei denen den einzelnen Registern Bits zugeordnet wurden. Die Mengenoperationen lassen sich dann als logische Operationen ausdrücken, was die Berechnungszeit drastisch verkürzt.

Erkennen von Schleifen im Kontrollfluss

Es ist wichtig zu wissen, wo im Kontrollflussgraphen Schleifen (Kreise, Zyklen) vorhanden sind. Diese Analyse wird auf Funktionsebene durchgeführt. Es wird dazu ein relativ einfacher rekursiver Algorithmus verwendet, der von einer Tiefensuche abgeleitet ist und bei dessen Ausführung jeder besuchte Knoten markiert wird. Stößt er dann auf einen schon markierten Knoten, so wurde eine Schleife gefunden. Es sind aber weitere Algorithmen denkbar (z. B. [34], prinzipiell zu lösendes Problem: *cycle detection*).

In diesem Algorithmus wird außerdem für jeden Knoten seine Tiefe im Kontrollflussgraph seiner Funktion berechnet. Die Tiefe eines Knoten ist die maximale Anzahl an Kanten, die er von der Wurzel des Graphen – hier ist das der Beginn der Funktion – entfernt ist. Bei der Tiefenberechnung müssen zyklische Kanten ignoriert werden,

sonst wäre die Tiefe der Kanten in den Zyklen unendlich (und der Algorithmus terminiert nicht).

Eine weitere Herausforderung bei der Rekonstruktion von Schleifen ist es, zumindest abschätzen zu können, wie oft die Schleifen ausgeführt werden. Der grundlegende Schritt dazu ist die Rekonstruktion von Registerwerten (siehe Abschnitt 3.5.3). Für die Berechnung der Schleifengrenzen könnte dann beispielsweise *Circular Linear Progression*, wie von Sen et al. [139] vorgestellt, verwendet werden.

Berechnung der Datenabhängigkeiten

Um z. B. Instruktionen neu anordnen zu können ist es essentiell, die Datenabhängigkeiten unter den Instruktionen zu kennen. Ihre Berechnung ist an sich ein gelöstes Problem. Gespeichert werden die Kanten des Datenflussgraphen analog zu den Kontrollflussgraphen (siehe Abschnitt 3.4).

Echte Datenabhängigkeiten werden auf Programm-Ebene berechnet. Grund dafür ist, dass sie Auskunft darüber geben, welche Daten wo verwendet und ob Ergebnisse überhaupt verwendet werden. Deshalb wird hier der größtmögliche Bereich gewählt. Da der Kontrollflussgraph auch spekulativ eingefügte Kanten enthält sind im Zweifel eher zu viele als zu wenige Datenabhängigkeiten vorhanden.

Falsche Datenabhängigkeiten (also Ausgabe- und Gegenabhängigkeiten) werden nur auf Funktionsebene berechnet, da sie ausschließlich für die Anordnung von Instruktionen relevant sind und derartige Änderungen bisher nur auf Funktionsebene angedacht sind.

Mit den echten Datenabhängigkeiten können dann für einige Instruktionen die Werte der Eingaberegister ermittelt werden. Das ist die Vorstufe von *constant propagation*, bei der versucht wird, Berechnungen durch feste Werte zu ersetzen, sodass Instruktionen dann weniger Datenabhängigkeiten aufweisen. Die Korrektheit dieser Analyse kann automatisch sichergestellt werden, indem die berechneten Werte mit Werten aus dem Simulator verglichen werden.

3.5.4. Unterteilen des Programms in „Basic Blocks“

Ein *Basic Block* (siehe 3.4 und [4], im Folgenden als *Block* bezeichnet) besteht aus einer oder mehreren Instruktionen. Ein Programm, repräsentiert durch Instruktionen und ihren Kontrollfluss, kann komplett in Blöcke umgewandelt werden. Ein Block hat folgende Eigenschaften:

- Der Kontrollfluss kann den Block nur an genau einer Stelle, nämlich der ersten Instruktion, betreten. In anderen Worten beginnt die Ausführung eines Blocks immer mit der ersten Instruktion.
- Der Kontrollfluss kann einen Block nur mit der letzten Instruktion verlassen. Es kann also nur nach der letzten Instruktion ein anderer Block ausgeführt werden.

Einen Block kann man sich wie eine besonders komplexe Instruktion vorstellen (siehe Abbildung 3.6), da bei Ausführung der ersten Instruktion eines Blockes sicher alle

anderen Instruktionen des Blockes ausgeführt werden. Durch die Zusammenfassung von Instruktionen zu Blöcken werden einige Analysen einfacher bzw. erst möglich. Grund dafür ist, dass bei Analyseschritten mit beispielsweise exponentieller Komplexität (also $O(n^3)$) die Berechnungszeit stark abnimmt, wenn weniger Elemente analysiert werden müssen.

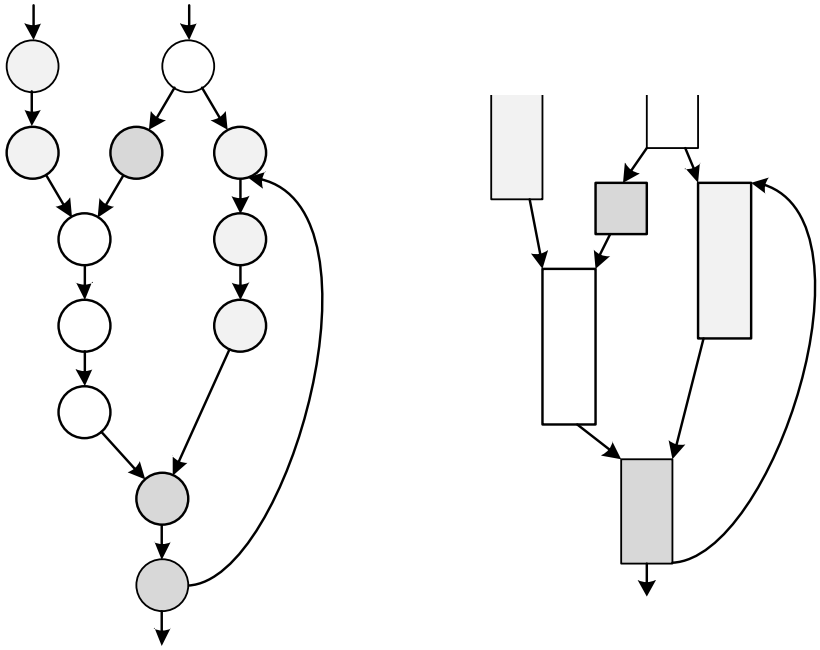


Abbildung 3.6.: CFG eines Programmstückes mit Instruktionen (links) und Blöcken (rechts)

Zur Berechnung der Blöcke wird das Programm Instruktion für Instruktion entsprechend der Anordnung im Speicherabbild durchlaufen. Zu einem Block b_i wird solange die aktuelle Instruktion j hinzugefügt bis eine der folgenden Bedingungen erfüllt ist. Dann wird ein neuer Block b_{i+1} begonnen und j wird zu diesem hinzugefügt:

- Die Instruktion vor j (bezogen auf die Speicheradresse) kann den Kontrollfluss verändern, ist also ein Sprungbefehl.
- j hat im Kontrollflussgraphen mehr als einen Vorgänger.
- Die Instruktion vor j (im Speicherabbild) ist ein NOP-Befehl und j ist kein NOP-Befehl. NOP-Befehle führen keine Berechnungen durch und tauchen z. B. am Anfang des Programms und zwischen Funktionen auf.
- Die Instruktion mit der Adresse nach j ist der Beginn einer Funktion, d. h. sie ist das Ziel von Funktionsaufrufen (OpCode `jal`).
- Die Instruktion mit der Adresse vor j ist Funktionsrücksprung, d. h. sie hat die Form *jr* \$31.

Anschließend wird für das Programm auf Block-Ebene der Kontrollflussgraph analog wie für Instruktionen modelliert; die Basis ist dabei der Kontrollfluss, der auf Instruktions-Ebene berechnet wurde (siehe Abschnitt 3.5.3). Eine Übersicht darüber, wie viele Instruktionen üblicherweise in einem Block enthalten sind, gibt Anhang A.2.

3.5.5. Analyseschritte auf Block-Ebene

Auf Block-Ebene werden – ebenso wie auf Ebene der Instruktionen – der Kontrollfluss, die Funktionstabelle und der Datenflussgraph berechnet; auch eine Liveness-Berechnung auf Registerebene wird durchgeführt. Zusätzlich wird der Dominanz- und Post-Dominanz-Baum berechnet.

Berechnung von Dominanz-Beziehungen

Ein Block A dominiert einen Block B, wenn jeder mögliche Pfad im Kontrollflussgraph, der zu B führt, A enthält. Wird also B ausgeführt, so muss irgendwann vorher A ausgeführt worden sein. Die Dominatoren von B ist die Menge aller Blöcke, die B dominieren (vgl. [4]).

Analog wird Post-Dominanz definiert, nur dass hier ein Block A einen Block B post-dominiert, wenn nach dem Block B auf jeden Fall A ausgeführt werden muss. Das bedeutet, dass jeder Pfad im Kontrollflussgraphen, der bei B startet, A enthält.

Die Dominanzbeziehung $dom(...)$ wird üblicherweise mit einem iterativen Verfahren berechnet (vgl. [42], Kapitel 9.3.2). Für GAPtimize wurde entschieden, die Berechnung auf Funktionsebene durchzuführen. Der Ablauf ist dann wie folgt:

1. Start: Ausgangspunkt ist, dass der erste Block der Funktion n_0 nur sich selbst dominiert: $dom(n_0) := n_0$. Alle anderen Blöcke $N \min\{n_0\}$ werden von allen Blöcken N der Funktion dominiert:

$$\forall n_i \in N, n_i \neq n_0 : dom(n_i) := N$$

2. Setze für alle Blöcke $n_i \in N$ die Menge der dominierenden Blöcke so, dass sie dem Schnitt der Mengen aller dominierenden Blöcke ihrer Vorgänger im CFG $preds(n_i)$ entspricht. In diese Menge wird abschließend n_i eingefügt:

$$\forall n_i \in N, n_i \neq n_0 : dom(n_i) := \{n\} \cup \left(\bigcap_{m \in preds(n_i)} dom(m) \right)$$

3. Der letzte Schritt wird bis zu einem stabilen Zustand wiederholt.

Die Post-Dominatoren werden analog ermittelt, nur dass hier

$$\forall n_i \in N, n_i \neq n_0 : pdom(n_i) := \{n\} \cup \left(\bigcap_{m \in succs(n_i)} pdom(m) \right)$$

verwendet wird, wobei $pdom(...)$ die Post-Dominanz-Beziehung ist und mit $succs(a)$ die Menge der Blöcke ermittelt wird, die im Kontrollflussgraph direkt nach a stehen, für die es also eine Kante zu a gibt.

3.6. Techniken zur Unterstützung der Programm-Restrukturierung

Eine besondere Herausforderung entsteht, wenn Instruktionen zu einem Programm hinzugefügt oder aus ihm gelöscht werden sollen. Bei beiden Fällen verschieben sich die Adressen der nachfolgenden Instruktionen. Dem muss Rechnung getragen werden, andernfalls kann das Programm nicht mehr korrekt ausgeführt werden, da Sprünge auf die falschen Adressen verweisen.

Sprungziele können in der Sprunginstruktion selbst enthalten sein; diese Sprünge werden direkte Sprünge genannt. Wird das Sprungziel aus einem Register gelesen, so ist es ein indirekter Sprung. Die im Register enthaltene Adresse stammt meistens aus dem Arbeitsspeicher. Am wahrscheinlichsten sind die Bereiche *Function Return Stack* (kurz: Stack) und das `.data`-Segment. Im Stack wird nach einem Funktionsaufruf gespeichert, wohin nach Ausführung einer Funktion gesprungen werden soll. Im `.data`-Segment des Arbeitsspeichers sind Konstanten des Programms gespeichert, unter anderem auch die Adressen bestimmter Funktionen und Instruktionen in Funktionen. Außerdem können Adressen mit arithmetischen Befehlen berechnet werden.

Je nach Sprungtyp und Adressquelle muss ein unterschiedliches Verfahren gewählt werden, um dafür zu sorgen, dass das modifizierte Programm letztendlich wieder korrekt ausgeführt werden kann:

- **Direkte Sprünge:** Um die Adressen von direkten Sprüngen, die im Sprungbefehl codiert sind, zu aktualisieren, werden die Kontrollflusskanten verwendet (**cfgEdge<T>**, siehe 3.4). Bei Änderung der Adresse des Sprungzieles ist durch die Kontrollflusskante immer noch eine Verbindung zur richtigen Instruktion vorhanden und die neue Adresse des Sprungzieles kann im Sprungbefehl als Zieladresse eingetragen werden.

- **Indirekte Sprünge – Adresse vom Stack gelesen:** Es ist kein Eingriff nötig, da auf dem Stack (bei korrekter Verwendung) nur Adressen stehen, die der Prozessor zur Laufzeit selbst ermittelt und dort abgelegt hat.

- **Indirekte Sprünge – Adresse aus dem `.data`-Segment:** Adressen aus dem Speicher – es handelt sich um den Bereich ab `0x10000000` (vgl. Abschnitt 2.2.1) – werden v. a. für Sprungtabellen (*branch tables*, siehe z. B. 3.5.3) verwendet. Die betreffenden Dateien des Speicherabbildes werden eingelesen und ohne weitere Interpretation des Dateiinhaltes wird nach Adressen gesucht, die im Programm die Adressen von Blöcken waren (vor der Modifikation des Programms). Sie werden dann durch die neuen Adressen des betreffenden Blockes ersetzt.

Quellcode 3.1: Beispiel für Adressberechnung im Code

```
400598 lui [ OUT $4 {0x400000} IMM {0x40} ]
4005a0 addiu [ OUT $4 {0x400430} IN $4 {0x400000} IMM {0x430} ]
```

Dieses Verfahren ist nicht besonders sorgsam, da es blind Daten ersetzt, die es für Adressen hält. Trotzdem funktioniert es einwandfrei, es konnten für alle Benchmarks keine Fehler in den Ausgabedaten festgestellt werden.

- **Indirekte Sprünge – Adresse durch arithmetische Befehle berechnet:** An manchen Stellen berechnet das Programm die Adresse für einen indirekten Sprung selbst mit arithmetisch-logischen Befehlen. Ein Beispiel zeigt Listing 3.1.

  Diese Befehle müssen angepasst werden. Dazu wird im ganzen Programm nach Paaren von Befehlen mit OpCode `lui` und `addiu` gesucht, bei denen die `addiu`-Instruktion eine gültige Adresse berechnet. Für die Berechnung der Registerausgabewerte werden die Datenabhängigkeiten unter den Befehlen verwendet (vgl. Abschnitt 3.5.3).

  Auch bei diesem Verfahren sind theoretisch fehlerhafte Modifikationen möglich, die sich aber in der Praxis nicht zeigen.

Da das Programm als solches durch die Aktualisierung der Adressen nicht mehr verändert wird, hat die Aktualisierung der Adressen keine Implikationen auf die Performance des Programms. Anders ist das bei alternativen Verfahren, die berechnete Adressen im Programm mit einer Hilfsfunktion korrigieren.

3.7. Plattform für Auswertungen und Statistiken

Neben der Optimierung von Programmen können mit GAPtimize auch Statistiken und Auswertungen erstellt werden. Sie werden im Rahmen dieser Arbeit immer wieder verwendet. Als Beispiel seien die Registerverwendung oder die Verteilung der dynamischen Blocklängen (siehe Anhang A.2) genannt. Zur Verfügung stehen alle Daten, die mit den Analysen aus Abschnitt 3.5 gewonnen werden können. Da der Quellcode von GAPtimize zur Verfügung steht, können weitere Schritte eingebaut werden.

4. Methoden zur Performance-Steigerung

Ziel ist es, die Performance von Programmen bei der Ausführung auf dem GAP zu erhöhen. Die Grundlage dafür findet sich in Kapitel 2.3; dort wird erklärt, wie Performance gemessen werden kann, was die Performance von Programmen beeinflusst und wie die Auswirkungen der Einflüsse quantifiziert werden können.

In Abschnitt 2.2.3 ist ein Klassifikationsrahmen für mögliche Beeinflussungen der Programmausführung vorgestellt worden. Dementsprechend lassen sich die weiteren Abschnitte dieses und des nächsten Kapitels gliedern:

- In Abschnitt 4.1 werden Coding-Guidelines beschrieben.
- In den Abschnitten 4.3, 4.4 und 4.6.1 werden Verfahren vorgestellt, die nur den auszuführenden Programmcode verändern.
- Die Optimierung im Abschnitt 4.5 beeinflusst sowohl die Hardware des Prozessors wie auch das Programm, das ausgeführt wird.
- Verfahren, die ausschließlich die Hardware des Prozessors verändern, werden im Rahmen dieser Arbeit nicht direkt eingeführt, die Suche nach optimalen Hardware-Parametern für den GAP in Abschnitt 5.2 könnte aber so verstanden werden.
- Zusammenfassend erklären die Kapitel 5.3 und 5.4 die Effekte der Kombination verschiedener Verfahren und wie bestmögliche Performance erreicht werden kann.

Außerdem kann zwischen Techniken unterschieden werden, die (a) bereits für andere Prozessorarchitekturen vorgestellt wurden und für den GAP angepasst/abgewandelt werden sowie die (b) speziell für den GAP entwickelt werden. Selbstverständlich sind neu entwickelte Techniken eigentlich interessanter, dennoch werden bereits bekannte Techniken ebenso untersucht, wenn entweder andere oder wesentlich stärkere Auswirkungen auf die Performance erwartet werden.

Eines der Alleinstellungsmerkmale des GAP ist es, dass er Software ausführen kann, die nicht speziell für ihn übersetzt wurde. Somit muss für die Ausführung eines Programms auf dem GAP der Quellcode nicht vorliegen, es genügt eine Binärdatei im richtigen Befehlsformat. Um diese Fähigkeit zu bewahren empfehlen sich als automatische Code-Optimierungen *Post-Link-Optimierungen* (PLOs), die als Eingabe das kompilierte Programm verwenden. Zur Anwendung von PLOs wird GAPtimize verwendet (vgl. Kapitel 3).

Die Code-Optimierung mit GAPtimize ist immer als optional anzusehen. Deshalb wird von allen verwendeten Verfahren gefordert, dass sie so gestaltet sind, dass ein

Programm auf dem GAP selbst dann ausgeführt werden kann, wenn das Programm nicht verändert wurde, die Hardware aber schon. Die daraus resultierenden Performance-Einbußen sollten sich in einem vertretbaren Rahmen befinden; diese Fähigkeit wird als *graceful degradation* bezeichnet.

Für diejenigen Code-Optimierungen, die keine Änderungen der Hardware voraussetzen, können die resultierenden Programme auf dem SimpleScalar-Simulator (siehe [16], vgl. Kapitel 2.2.1) ausgeführt werden. Dies erlaubt einen Rückschluss, ob die Auswirkungen der Optimierungen GAP-spezifisch sind oder nicht.

Bei der Beurteilung der Performance-Auswirkungen einer Optimierung auf dem GAP ist es wichtig, dass die Größe des Arrays und des Befehlszwischenspeichers des GAP bzw. der daraus resultierende Hardwarebedarf mit berücksichtigt werden (vgl. Abschnitt 5.2). Es ist denkbar, dass bei einem sehr groß dimensionierten Prozessor eine Optimierung wesentlich geringere Auswirkungen hat als bei einem Prozessor mit kleinem Array und kleinem Speicher. Deshalb werden für die Bewertung eines Verfahrens unterschiedliche Konfigurationen des GAP verwendet mit stets der selben Anzahl an Konfigurationsebenen und dem selben Befehlszwischenspeicher. Grund dafür ist, dass sowohl Konfigurationsebenen wie auch der Befehlszwischenspeicher dadurch, dass sie den Zugriff auf Instruktionen beschleunigen, geeignet sind, Schwächen des Prozessors, die eigentlich durch Code-Modifikationen umgangen werden sollen, auszugleichen. Somit werden die Auswirkungen von Code-Modifikationen bei vielen Ebenen oder großem Zwischenspeicher sehr stark reduziert oder komplett unterdrückt, was nicht erstrebenswert ist.

Als Benchmarks werden bei der Evaluierung die bereits eingeführten 17 Benchmarks aus der MiBench Benchmark Suite verwendet (vgl. Abschnitt 2.3). Wird für eine Optimierung nach optimalen Einstellungen für Parameter gesucht (Design Space Exploration, DSE), so werden nur 10 der 17 Benchmarks verwendet. Grund dafür ist, dass bei einer DSE sehr viele unterschiedliche Konfigurationen simuliert werden müssen und deshalb nur bestimmte repräsentative Benchmarks dafür geeignet sind. Außerdem kann so ein Eindruck davon gewonnen werden, ob die mit den Test-Benchmarks gefundenen Einstellungen auch für andere Benchmarks passend erscheinen.

4.1. Coding Guidelines

Besonders bei der Programmierung in Sprachen mit niedriger Abstraktion (Assembler, C) tendieren Entwickler dazu, „manuell“ augenscheinlich passende Optimierungs-Konstrukte direkt in den Code einzubauen. Besonders bei Ausführung des erstellten Codes auf einer Architektur, die zum Entwicklungszeitpunkt nicht bekannt oder nur unzureichend verstanden war, kann das zu Problemen führen. Da der Code direkt verändert wurde, können diese Optimierungen nicht oder nur schwer durch einen neuen Compiler oder ein PLO-Tool wie GAPtimize wieder „deaktiviert“ werden und es kann dadurch schlechte Performance verursacht werden.

Ein Beispiel für eine Veränderung mit negativen Einflüssen ist das Abrollen von Schleifen. Es kann in C mit Präprozessor-Anweisungen realisiert werden. Daraus resultieren sehr lange Schleifenkörper, die dann, anders als die ursprüngliche Schleife, bei Ausführung auf dem GAP auf mehrere Konfigurationsebenen verteilt werden müssen. Im schlimmsten Fall werden mehr Ebenen als vorhanden benötigt und es tritt *Thrashing* (vgl. Abschnitt 4.5) auf. Dadurch kann die Performance des GAP einbrechen. Prototypisch dafür sind die Benchmarks `secu-rijndael-*` und `secu-rijndael-*-nounroll`.

Auch andere Konstruktionen können auf dem GAP unter Umständen zu niedriger Performance führen. Beispielsweise sollte Rekursion aufgrund der dann sehr häufig auftretenden Funktionsrücksprünge vermieden werden. Da es sich um einen indirekten Sprung handelt und der GAP dafür keine Vorhersagetechnik besitzt, treten zusätzliche Wartezeiten auf.

Es lassen sich mit Sicherheit weitere Beispiele finden. Um darüber zu informieren, dass sie vermieden werden sollten, eignen sich *Coding Guidelines*. Ein Beispiel dafür ist [98]. Denkbar wäre eine entsprechend diesen Coding Guidelines angepasste Version der Standardbibliothek (vgl. Kapitel 2.2.2) zu erstellen, die dann speziell für den GAP optimiert ist. Erste Versuche, das dynamische Speichermanagement zu optimieren, waren vielversprechend.

4.2. Bedingte Ausführung (Predicated Execution)

Predicated Execution (auch *Branch Predication*, *bedingte Ausführung*) ist ein Verfahren, mit dem ähnlich wie mit Sprungvorhersagetechniken die Verzögerungen, die bei der Ausführung von Sprungbefehlen mit bekanntem Ziel auftreten können, reduziert werden sollen.

Nach Einführung und Motivation (Abschnitt 4.2.1) wird erklärt, wie die bedingte Ausführung im GAP und GAPtimize implementiert ist. Anschließend wird sie (a) als einziges Verfahren zur Sprungbehandlung verwendet (Abschnitt 4.2.3) und (b) zusammen mit einer Sprungvorhersagetechnik kombiniert und ausgewertet (Abschnitt 4.2.4). Verwandte Arbeiten werden in Abschnitt 4.2.5 vorgestellt und eine Zusammenfassung über die bedingte Ausführung gibt Abschnitt 4.2.6.

4.2.1. Einführung und Ziele

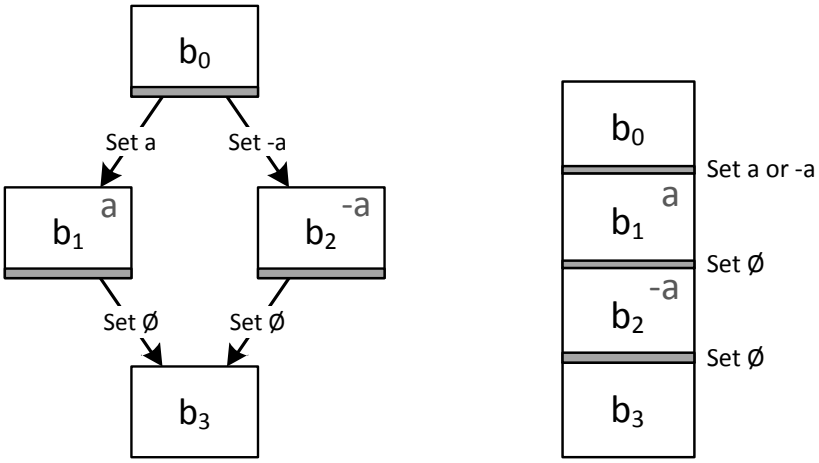
Analog wie bei superskalaren Prozessoren steht das Frontend des GAP vor der Herausforderung, möglichst ohne Verzögerung Befehle für die Ausführung bereitzustellen. Sprungbefehle können dabei hinderlich sein, da vor der Ausführung oft unklar ist, welches der möglichen Sprungziele als nächste Instruktion ausgeführt werden soll.

Die üblichen Sprungvorhersagetechniken schätzen für direkte Sprünge, basierend auf Vergangenheitswerten, den Ausgang eines Sprunges ab, also ob voraussichtlich gesprungen wird oder nicht. Das Frontend lädt dann diesen Zweig des Kontrollflussgraphen und, im Fall des GAP, platziert die Befehle im Array. Wird dann bei der Ausführung entschieden, dass doch der andere Zweig ausgeführt werden soll, so muss die aktuelle Konfiguration des Arrays gelöscht werden (oder auf eine andere Konfigurationsebene umgeschaltet werden) und das Frontend angestoßen werden, damit es den gewünschten Zweig lädt. Das benötigt Zeit. Man spricht von der *branch miss penalty* bzw. Verzögerung durch einen falsch vorhergesagten Sprung.

Anders ist das bei der bedingten Ausführung. Hier wird nicht bereits im Frontend die Entscheidung getroffen, welche Befehle geladen werden sollen, sondern es werden alle Befehle geladen und es wird erst bei der Ausführung entschieden, welche Befehle ausgeführt werden sollen. Es werden also bei einem Sprung beide Zweige des Kontrollflusses geladen.

Jeder Zweig des Kontrollflusses wird mit einer Bedingung versehen, einem *Prädikat*. Der Sprungbefehl, der zur Verzweigung des Kontrollflusses führt, setzt dann das eine oder das andere Prädikat. Es werden nur die Befehle ausgeführt, deren Prädikat gesetzt ist. Die übrigen Befehle werden übersprungen. Befehle sind dann nicht mehr vom *Befehlszähler* (ein Register im Prozessor, das die Adresse des nächsten auszuführenden Befehls enthält) sondern von den Werten der Prädikate abhängig. Somit werden mit der bedingten Ausführung die Kontrollflussabhängigkeiten in Datenflussabhängigkeiten umgewandelt [5].

Grundsätzliches Ziel ist die Erhöhung der Performance des Prozessors (siehe Kapitel 2.3.1). Das soll erreicht werden, indem die Wartezeiten auf das Frontend reduziert werden und durch selteneres Neukonfigurieren des Arrays. Dadurch sollte sich auch



(a) Kontrollflussgraph mit Blöcken (b) Anordnung im Speicher

Abbildung 4.1.: Programmbeispiel mit Verzweigung und Zusammenfluss sowie mögliche Umsetzung mit bedingter Ausführung (Prädikate a und $\neg a$), die Ausführung beginnt bei b_0.

die Anzahl der Schleifen erhöhen, die im Array abgearbeitet werden können, was zu einer weiteren Erhöhung der Performance führen kann.

4.2.2. Unterstützung der bedingten Ausführung in GAP und GAPtimize

Die nachfolgend erörterte Implementierung der bedingten Ausführung unterstützt sowohl bedingte wie auch unbedingte direkte Sprünge, die aber vorwärtsgerichtet[1] sein müssen und Funktionsgrenzen nicht überschreiten dürfen. Anders als bei der Implementierung in einem Compiler findet nach Anwendung der bedingten Ausführung kein erneutes Scheduling der Befehle oder ähnliches statt.

Anforderungen

In den Abbildungen 4.1(a) und 4.1(b) ist ein einfaches Beispiel für eine Verzweigung mit anschließendem Zusammenfluss (`if-else`-Struktur) in einem Kontrollflussgraphen zu sehen. Hier ist erkennbar, welche Operationen der Prozessor für die bedingte Ausführung unterstützen muss:

- Es müssen Strukturen für Prädikate vorhanden sein.
- Prädikate müssen gesetzt werden können.
- Prädikate müssen gelesen bzw. überwacht werden können.

[1] Vorwärts bedeutet, dass die Zieladresse größer ist als die Adresse der Sprunginstruktion. Ob es ein Schleifenrücksprung ist oder nicht, also die Ausrichtung im Kontrollflussgraph, interessiert nicht.

Dabei sind mehrere Implementierungen denkbar. Oft werden als Prädikate einzelne Bits in Spezialregistern verwendet. Meistens werden separate Befehle verwendet um die Prädikate zu setzen. Sie können oft miteinander verknüpft werden (*und* bzw. *oder*). Werden keine eigenen Befehle verwendet, um Prädikate zu definieren, so muss festgelegt werden, ob jeder Befehl oder nur ausgewählte Befehle Prädikate setzen dürfen (*full predication* versus *partial predication*, siehe Abschnitt 4.2.5).

Befehlsausführung im GAP

Beim GAP werden die notwendigen Änderungen am Programm erst als Post-Link-Optimierung durch GAPtimize vorgenommen. Um keine neuen Befehle zum Setzen und Löschen von Prädikaten einfügen zu müssen, was die Vorbereitung eines Programms mit GAPtimize stark erschweren würde, wird jedem Befehl erlaubt, Prädikate zu setzen, zu löschen und zu lesen[2]. Zudem ergeben sich dadurch, dass keine zusätzlichen Befehle eingefügt werden, keine Auswirkungen auf das Verhalten des Befehlszwischenspeichers.

Im GAP wird als Struktur für die Prädikate ein einziges globales Register mit 8 Bits verwendet. Die $2^8 = 256$ unterschiedlichen möglichen Werte des Registers werden im Folgenden Markierungen genannt. Jeder Wert dieses Registers A repräsentiert ein einzelnes oder eine beliebige Kombination aus Bit-Prädikaten, beispielsweise $A_8 = (a \wedge b) \vee c$. Somit kann auf die explizite Verknüpfung von Prädikaten verzichtet werden.

Für die bedingte Ausführung werden drei Bytes im Befehlsformat des PISA-Befehlssatzes (vgl. Kapitel 2.2.1) verwendet, die bisher ungenutzt waren. Es ergibt sich folgende Semantik für die ersten Bytes jedes Befehls:

`OpCode`	`pred_set_taken`	`pred_set_nottaken`	`pred_read`	...

Ein Befehl erwartet die Markierung `pred_read`. Entspricht der Wert des globalen Markierungsregisters dieser Markierung, so wird der Befehl ausgeführt, ansonsten nicht. Nach Ausführung dieses Befehls wird das Markierungsregister auf einen neuen Wert gesetzt. Bei einem Sprungbefehl mit zwei möglichen nächsten Instruktionen im Kontrollflussgraphen wird die Markierung auf `pred_set_taken` gesetzt, wenn der Sprung ausgeführt wird. Ansonsten (das gilt auch für unbedingte Sprünge) nimmt die Markierung den Wert `pred_set_nottaken` an.

Der Wert 0x00 wird als Standard-Markierung verwendet. Diesen Wert besitzen die drei Bytes im Befehlssatz auch, wenn die bedingte Befehlsausführung nicht verwendet wird. Liest ein Befehl die Markierung 0x00 und das Markierungsregister hat diesen Wert, dann wird er ausgeführt, als ob die bedingte Ausführung nicht aktiv wäre. In dieser Situation verändern Sprünge den Befehlszähler.

[2] Dabei bedeutet *lesen*, dass der Befehl nur ausgeführt wird, wenn das Prädikat einen vorgegebenen Wert besitzt.

Vergabe der Markierungen in GAPtimize

GAPtimize muss erstens Situationen finden, bei denen die bedingte Ausführung verwendet werden kann. Das schließt Instruktionen aus, die bei einer Referenzausführung das Ziel von indirekten Sprüngen waren. Außerdem müssen, wie später genauer erklärt wird, die Instruktionen auch eine Heuristik erfüllen, um mögliche Nachteile durch die bedingte Ausführung nach Möglichkeit auszuschließen. Zweitens muss GAPtimize Werte für die Markierungen ermitteln.[3] Drittens müssen diese Markierungen in das Speicherabbild des Programms für die Ausführung auf dem GAP geschrieben werden.

Die Vergabe der Prädikate findet pro Funktion statt. Die Instruktionen der Funktion werden durchlaufen, wobei mit den Instruktionen mit der geringsten Tiefe im Kontrollflussgraphen der Funktion (siehe Kapitel 3.5.3) begonnen wird und die Tiefe dann mit jeder Iteration erhöht wird (vgl. Abbildung 4.2).

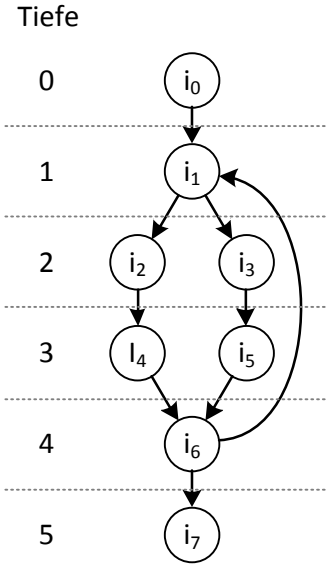


Abbildung 4.2.: Beispiel für Kontrollflussgraph, die Knoten (Instruktionen) i_0 bis i_7 sind nach Tiefe aufsteigend angeordnet.

Für jede Instruktion, die die bedingte Ausführung verwenden soll, wird eine Markierung errechnet, die sie lesen soll. Sie wird in `pred_read` gespeichert. Außerdem müssen alle Instruktionen, die Kontrollflusskanten zu dieser Instruktion haben, so verändert werden, dass sie die berechnete Markierung setzen. Ist die Kontrollflusskante ein Schleifenrücksprung, so muss zudem der OpCode der Instruktion so verändert werden, dass sowohl die Markierung gesetzt als auch der Befehlszähler verändert werden.

[3] Die hier präsentierte Lösung ist leicht an den Algorithmus 2 aus [95] angelehnt.

Berechnung der Markierungen

Um die Anzahl der Bits zu reduzieren, die für die Darstellung der Markierungen benötigt werden, wird für jede Instruktion i zuerst die minimale Speicheradresse einer Instruktion j ermittelt, die eine Kontrollflusskante zu i hat. Dann wird zwischen i und j sowie dem Funktionsende nach einer freien Markierung gesucht. Dazu wird, beginnend bei der Markierung 0x01, überprüft, ob sie von einer Instruktion gelesen wird. Wenn nicht, so kann sie verwendet werden. Ansonsten wird die nächste Markierung, beispielsweise 0x02, überprüft. Wird jede Markierung bereits verwendet, so kann für diese Instruktion die bedingte Ausführung nicht verwendet werden. Andernfalls wird mit diesem Vorgehen eine Markierung so schnell wie möglich erneut vergeben.

Entspricht bei einem Befehl die erwartete Markierung nicht dem Wert des Markierungsregisters, so wird er übersprungen und der im Speicher nachfolgende Befehl überprüft. Obwohl übersprungene Befehle nicht ausgeführt werden, müssen sie dennoch geladen werden, die Konfigurationseinheit durchlaufen und sie belegen Funktionseinheiten im Array. Außerdem ist für die Überprüfung der Vorbedingung – also der Markierung des Befehls – etwas Zeit nötig. Dieser Zeitaufwand kann sich beim Überspringen vieler Befehle derart summieren, dass es schneller wäre, das Array zu löschen und direkt die Zielinstruktion zu laden. Aus diesem Grund wird eine Heuristik verwendet, um die Verwendung der bedingten Ausführung nur auf die Befehle zu beschränken, bei denen die Ausführungszeit aller Voraussicht nach reduziert werden kann.

Für jede Instruktion, die eventuell eine Markierung lesen könnte, wird zuerst überprüft, ob mindestens eine der eingehenden Kanten vorwärts gerichtet ist. Andernfalls kann die bedingte Ausführung keine Vorteile bringen, da sie nur für vorwärts gerichtete Sprünge verwendet werden kann.

Anschließend werden die Länge und das Gewicht der eingehenden Kanten im Kontrollflussgraph untersucht. Dabei wird die Größe L_{max} berechnet, das ist die maximale Länge einer eingehenden Kante, die mindestens einmal ausgeführt wird.

Für diese Größe sind in der Heuristik Grenzwerte enthalten. Für Knoten, die Teil einer Schleife sind, gilt als Grenzwert G_{max}^{loop}. Ansonsten wird G_{max}^{total} verwendet. Ist für eine Instruktion der Wert L_{max} größer als der durch die Heuristik vorgegebene Maximalwert, so wird die bedingte Ausführung für diese Instruktion nicht verwendet.

Mit diesem Verfahren können Markierungen so vergeben werden, dass keiner der verwendeten 17 MiBench-Benchmarks mehr als sieben Markierungen einschließlich 0x00 als Standard-Markierung benötigt (siehe Grafik 4.3). Somit genügen eigentlich drei der vorgesehenen acht Bit für die Markierungen (vgl. 4.2.2).

4.2.3. Verwendung als einziges Verfahren zur Sprungbehandlung

In diesem Abschnitt wird die bedingte Ausführung als einziges Verfahren zur Sprungbehandlung verwendet. Sie konkurriert nur damit, Sprünge nicht zu behandeln und somit viele Array-Leerungen (bzw. Wechsel der Konfigurationsebene, wenn mehrere vorhanden sind) in Kauf zu nehmen.

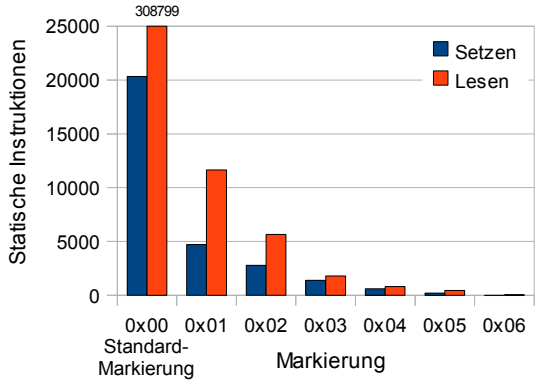


Abbildung 4.3.: Verwendung der Markierungen für 17 MiBench-Benchmarks bei Ausführung mit bedingter Ausführung und ohne Sprungvorhersage sowie $G_{total}^{max} = 99$ und $G_{loop}^{max} = 99$, also häufiger Verwendung der bedingten Ausführung

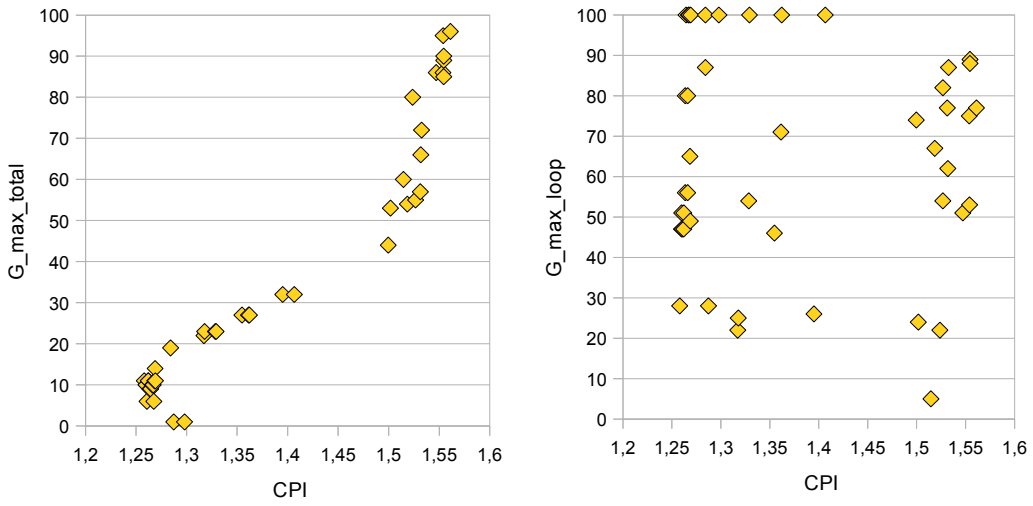
Optimale Parameter für die Heuristik

Da noch unklar ist, wie die zwei Parameter der Heuristik gesetzt werden müssen um optimale Ergebnisse zu erzielen, wird FADSE (siehe [20, 18] und [80]) verwendet, um Werte zu finden, die zu guter Performance führen. Dazu werden für jede Parameterkonfiguration 10 Benchmarks simuliert. Um den Parameterraum, in dem gesucht wird, einzugrenzen wird die Hardware des GAP auf 8 Zeilen, 8 Spalten und 8 Konfigurationsebenen (kurz: 8x8x8) sowie 8 kB Befehlszwischenspeicher festgelegt. Später werden diese Ergebnisse mit den Ergebnissen für Arrays anderer Größe verglichen.

Mit dem genetischen Algorithmus NSGA-II [47] werden 10 Generationen mit je 30 Individuen berechnet. Ziel der Optimierung ist die Reduktion der Maßzahl CPI (Takte pro Instruktion, *Clocks per Instruction*) und somit die Maximierung der Performance. Da die Hardware fixiert ist, also die Hardware-Komplexität immer gleich ist, handelt es sich um eine Suche mit einer Zielfunktion (*single objective design space exploration*). Als Definitionsbereich ist für die Parameter G_{max}^{total} und G_{max}^{loop} die Menge der ganzen Zahlen zwischen 0 und 100 gesetzt $\{0, 1, 2, 3, ...100\}$.

Der Algorithmus findet sehr schnell ein Optimum, die Konfiguration mit $G_{max}^{total} = 11$ und $G_{max}^{loop} = 28$ sowie $CPI = 1,26$. Zum näheren Verständnis des Einflusses der Parameter auf die erzielte Performance zeigen die Diagramme 4.4(a) und (b) die Parameterwerte und die damit erreichte Performance. Es ist klar erkennbar, dass G_{max}^{total} direkt mit dem Wert CPI zusammenhängt. Gute Performance-Werte können nur erreicht werden, wenn G_{max}^{total} einen Wert um 10 besitzt. Kleiner oder größer gewählte Werte haben einen negativen Einfluss auf die Performance des Prozessors. Anders ist das Bild für G_{max}^{loop}, hier ist kein direkter Zusammenhang erkennbar.

Dieser Eindruck wird verstärkt durch Abbildung 4.5. Hier sind für verschiedene Array-Größen die Parameter-Konfigurationen gezeigt, mit denen eine Performance erreicht werden kann, die höchstens zwei Prozent von der für die jeweilige Array-



(a) Zusammenhang zwischen CPI und G_{max}^{total} (b) Zusammenhang zwischen CPI und G_{max}^{loop}

Abbildung 4.4.: Parameter der Heuristik für die Selektion von Befehlen für die bedingte Ausführung und ihre Auswirkung auf die Gesamt-Performance (GAP mit 8x8x8-Array)

Größe bekannten besten Konfiguration abweicht. Diese besten bekannten Konfigurationen sind mit großen Symbolen markiert; sie werden für weitere Auswertungen verwendet. Die meisten der sehr guten Konfigurationen haben Werte für G_{max}^{total} zwischen 8 und 13 und für G_{max}^{loop} zwischen 35 und 100, wobei hier die drei besten bekannten Konfigurationen nicht enthalten sind (18, 28, 28).

Ein Zusammenhang zwischen der Größe des Arrays und den Parametern lässt sich nicht erkennen.

Performance-Auswertung

Mit diesen Einstellungen kann für die 17 MiBench-Benchmarks die Performance, wie in Grafik 4.6 gezeigt, erreicht werden; Abbildung 4.7 stellt den Unterschied zur Ausführung ohne Sprungvorhersage dar. Diese Ergebnisse zeigen, dass die bedingte Ausführung bei den meisten Benchmarks positive Effekte aufweist, allerdings bis auf die beiden ADPCM-Benchmarks nicht mit der Sprungvorhersage gleichziehen kann.

Tabelle 4.1 zeigt eine Übersicht über die verschiedenen Techniken im GAP, mit denen Sprünge behandelt werden können. Die hier notierten Kosten beziehen sich auf zusätzlich benötigte Takte zur Behandlung des Sprungbefehls.

Sprünge, die mit der Schleifenbeschleunigung behandelt werden können, verursachen keine Kosten. Die Schleifenbeschleunigung kann aber nur in einem eingeschränkten Rahmen verwendet werden, nämlich genau dann, wenn das Sprungziel der ersten Instruktion der derzeit aktiven Konfigurationsebene entspricht. Befindet sich das Sprungziel als erste Instruktion auf einer anderen Konfigurationsebene, so kann dorthin mit einem Takt Verzögerung gewechselt werden. Befindet sich das

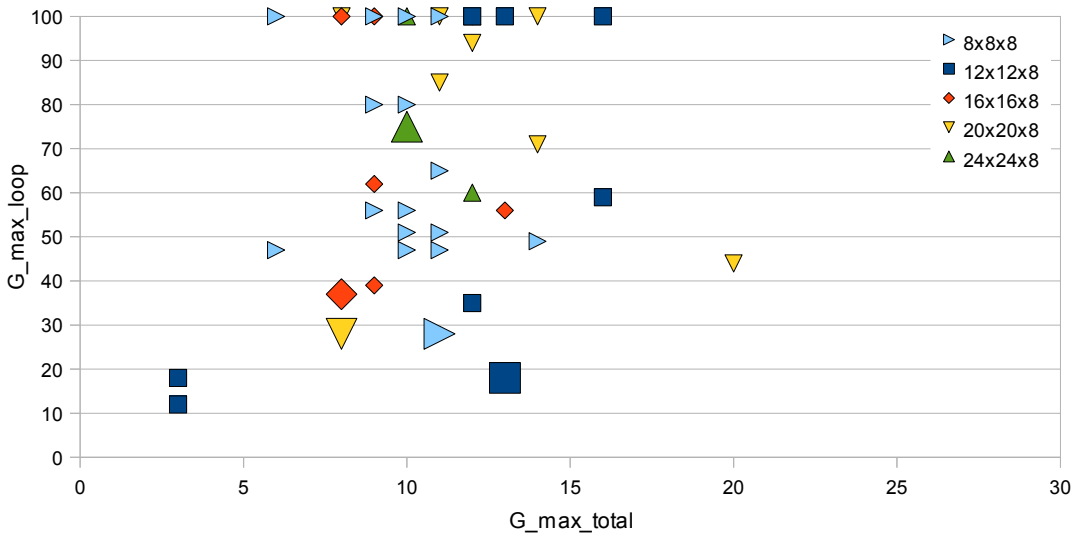


Abbildung 4.5.: Parameter-Werte für Heuristik für verschiedene Array-Größen des GAP, die maximal 2% schlechter sind als das jeweils bekannte Optimum

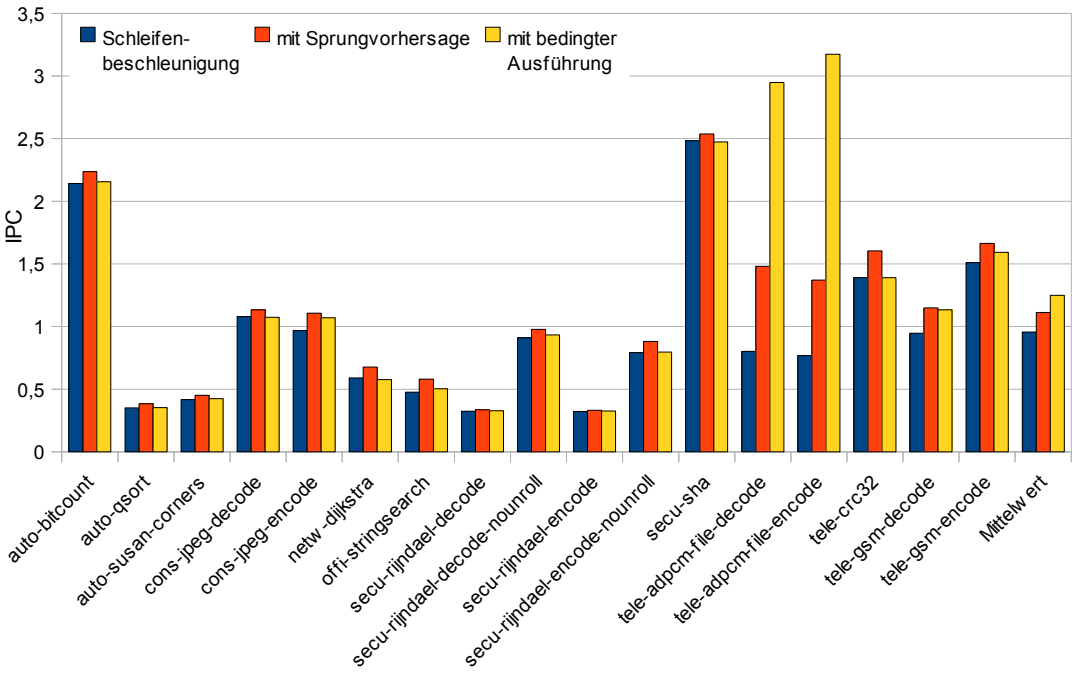


Abbildung 4.6.: Performance-Daten für 17 MiBench-Benchmarks ausgeführt auf dem GAP mit einem Array aus 8x8x8 FUs mit unterschiedlichen Optionen zur Sprungbehandlung (Schleifenbeschleunigung stets aktiviert)

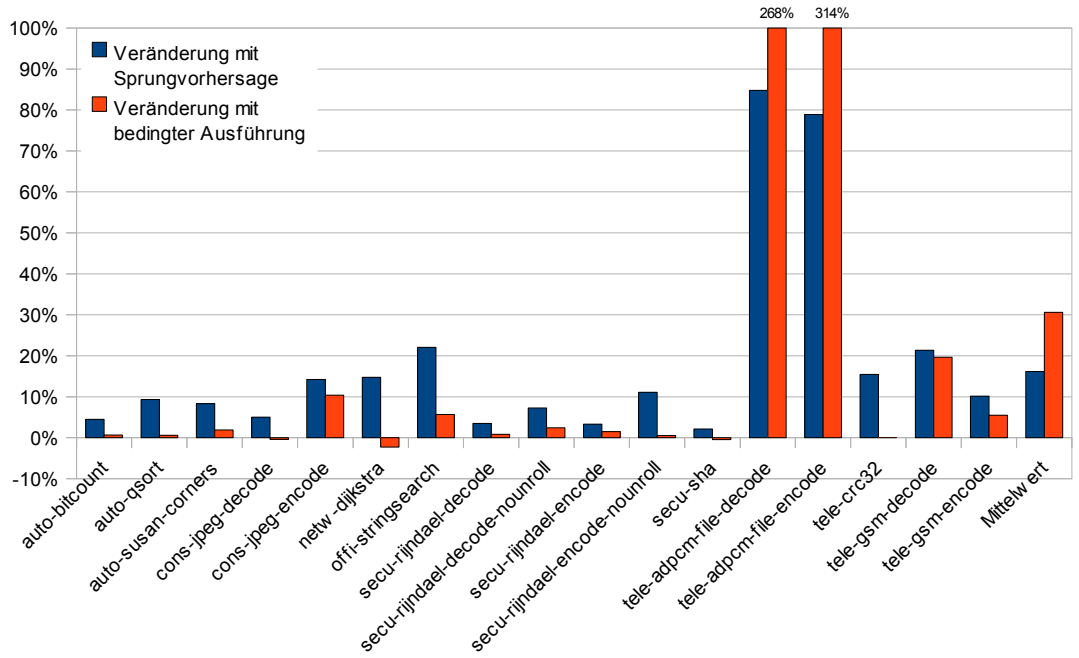


Abbildung 4.7.: Performance-Veränderung für 17 MiBench-Benchmarks ausgeführt auf dem GAP mit einem Array aus 8x8x8 FUs bei unterschiedlichen Optionen zur Sprungbehandlung (Schleifenbeschleunigung stets aktiviert)

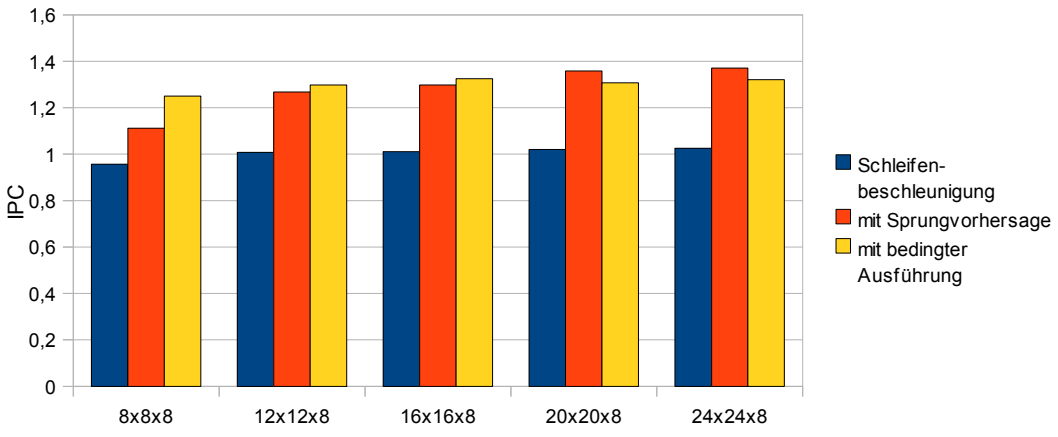


Abbildung 4.8.: Durchschnittliche Performance für 17 MiBench-Benchmarks ausgeführt auf GAP mit unterschiedlichen Array-Größen (X-Achse) und unterschiedlichen Verfahren zur Sprungbehandlung

Verfahren	Sprung-Klassifizierung			Verzögerung	
	Länge	Richtung	target	Sprung	kein Sprung
Nichts tun	–	–	–	5	0
Schleifenbeschleunigung[a]	kurz mittel	rückwärts	–	0	0
Umschalten der Ebene[b]	kurz mittel	rückwärts	–	1	1
Bedingte Ausführung[c]	kurz	vorwärts	bekannt	$1 + n^d$	0
Sprungvorhersage	–	–	bekannt	Korrekt $\Rightarrow$ 0; Ansonsten $\Rightarrow$ 5;	

[a] Nur wenn Sprungziel mit erster Instruktion auf aktiver Ebene übereinstimmt (meist Schleifenrücksprung, also rückwärts gerichteter und kurzer Sprung)
[b] Ziel des Sprunges muss erste Instruktion in einer anderen Ebene sein
[c] Kann nur unter bestimmten Bedingungen verwendet werden, vgl. Kapitel 4.2.2
[d] Kosten abhängig von übersprungenen Instruktionen

Tabelle 4.1.: Verfahren zur Sprungbehandlung im GAP und Kosten in Takten pro Sprung (Abschätzung); Striche bedeuten, dass keine Einschränkung vorliegt

Sprungziel nicht als erste Instruktion in einer anderen Ebene, so muss es geladen werden. Dazu muss das Frontend aktiviert und dann auf die Pipeline gewartet werden, das kostet mindestens fünf Takte, eventuell auch mehr wegen Verzögerungen beim Speicherzugriff.

Bei der bedingten Ausführung entstehen Kosten, wenn ein Sprung ausgeführt werden soll. Dann müssen einige Befehle übersprungen werden. Für die Überprüfung dieser Befehle muss Zeit aufgewendet werden, deswegen kostet jeder angenommene Sprung mindestens einen Takt. Bei Verwendung einer Sprungvorhersagetechnik verursachen korrekt vorhergesagte Sprünge keine Verzögerung, bei falscher Vorhersage kann entweder auf eine andere Konfigurationsebene gewechselt werden (ein Takt) oder es muss eine neue Konfiguration erstellt werden (fünf Takte).

Nehmen wir an, ein Sprung über fünf Instruktionen wird 200-mal ausgeführt, davon wird 100-mal gesprungen und 100-mal nicht. Die Sprungvorhersagetechnik arbeitet mit 85% Genauigkeit, was tatsächlich oft überschritten wird. Zusätzlich wird angenommen, dass sich das Sprungziel mit einer Wahrscheinlichkeit von 50% bereits in einer anderen Ebene befindet. Mit der bedingten Ausführung muss man ca. $1,5 * 100 = 150$ Takte einkalkulieren. Eine Sprungvorhersagetechnik führt zu einem Anteil von $1,00 - 0,85 = 0,15$ falsch vorhergesagten Sprüngen, das sind im Beispiel $200 * 0,15 = 30$ Sprünge. Die Hälfte dieser Sprünge findet ihr Ziel in einer anderen Ebene, die andere Hälfte muss neu geladen werden. Das führt zu einer Gesamtverzögerung von $0,5 * 30 * 1 + 0,5 * 30 * 5 = 15 + 75 = 90$ Takten. Bei diesem Beispiel wäre also die Sprungvorhersage das Mittel der Wahl, da hier die Verzögerung geringer ist.

Bezieht man in diese Betrachtungen noch ein, ob die Zielinstruktion als erste In-

struktion in einer anderen Ebene zu finden ist, weil beispielsweise der Sprung in einer Schleife ist, so kann man schließen, dass auch die Konfigurationsebenen stark dazu beitragen, die Verzögerungen durch Sprünge zu reduzieren. Aus diesem Grund ist zum einen der Performance-Gewinn durch die Sprungvorhersage relativ gering (vgl. Grafik 4.7 - Delta IPC). Ein anderer Effekt ist die in Abbildung 4.4(b) gezeigte geringe Auswirkung des Parameters G_{max}^{loop} auf die Performance.

Eine genaue Analyse der Benchmarks und der mit der bedingten Ausführung erzielten Ergebnisse zeigt, dass die Sprungvorhersage eine sehr hohe Treffergenauigkeit aufweist. Die einzige Ausnahme stellen die ADPCM-Benchmarks dar, bei denen es einige wenige Sprünge gibt, die sehr schwer vorherzusagen sind. Sie können mit der bedingten Ausführung effektiv behandelt werden.

Betrachtet man die durchschnittliche Länge der Konfigurationen, so wächst sie mit Anwendung der bedingten Ausführung. Dieses Ziel kann also erreicht werden. Die Veränderungen sind wieder besonders deutlich bei den ADPCM-Benchmarks.

Zusammenfassung

Als Fazit ist die bedingte Ausführung eine Möglichkeit, die Array-Ausnutzung zu erhöhen und die Verzögerungen durch Sprünge zu reduzieren. Sie ist dabei aber durch die Verwendung von Konfigurationsebenen in der Effektivität begrenzt und kann bis auf wenige Benchmarks nicht mit der klassischen Sprungvorhersage, einer Hardware-Technik, mithalten. Interessant wäre es, die Sprungvorhersage mit der bedingten Ausführung zu kombinieren und so die Vorteile zu erhalten, die mit der bedingten Ausführung bei einigen wenigen Benchmarks erzielt werden können.

4.2.4. Verwendung in Kombination mit Sprungvorhersage

Wie besonders in Grafik 4.7 zu sehen ist, sind die Ergebnisse, die mit der bedingten Ausführung erzielt werden können, nicht besonders überragend. Oft kann mit der normalen Sprungvorhersagetechnik des GAP eine bessere Performance erreicht werden. Dennoch lohnt sich die bedingte Ausführung für einige Benchmarks, vor allem `tele-adpcm-file-encode` und `tele-adpcm-file-decode`, da so signifikant bessere Ergebnisse erreicht werden können.

Es stellt sich die Frage, ob es eine Möglichkeit gibt, die bedingte Ausführung mit der Sprungvorhersagetechnik des GAP so zu kombinieren, dass Ergebnisse erzielt werden können, die besser oder zumindest gleich gut sind wie die besten Ergebnisse der einzelnen Techniken.

Heuristik zur Kombination der bedingten Ausführung und der Sprungvorhersage

Die bedingte Ausführung ist für Sprünge interessant (vgl. Tabelle 4.1), die (a) schwer vorherzusagen sind und (b) deren Ziel am besten nicht auf einer anderen Konfigurationsebene zu finden ist. Da man davon ausgehen kann, dass die Anzahl der Konfigurationsebenen immer beschränkt sein wird und somit immer nur eine sehr begrenzte

Anzahl an Sprüngen ihr Ziel in anderen Konfigurationsebenen finden wird, wird diese Anforderung im Folgenden vernachlässigt.

Es bleiben also schwer vorhersagbare Sprünge. Um sie zu identifizieren könnte beispielsweise beobachtet werden, bei welchen Sprüngen im Prozessor es zu falschen Vorhersagen kommt. Diese Vorgehensweise hat den Vorteil, dass so genau die Sprünge identifiziert werden können, mit denen der im Prozessor eingebaute Vorhersager sich schwer tut. Allerdings müssen dann sehr spezielle Daten aus dem Prozessor extrahiert werden, die weit über das Erstellen einer Stream-Datei mit der Befehlsabfolge (siehe Kapitel 3.5.3) hinausgehen und sehr spezielles Wissen erfordern. Stattdessen wird nachfolgend ein Vorgehen vorgeschlagen, bei dem basierend auf die Befehlsabfolge und statistische Größen versucht wird, die schwer vorhersagbaren Sprünge zu identifizieren.

Um das Verhalten von Sprüngen zu beschreiben genügt es nicht, nur zu notieren, wie der Sprung genommen wird und wie oft nicht; zusätzlich oder stattdessen muss das dynamische Verhalten erfasst werden, quasi die Abfolge der Sprungentscheidungen. In Abbildung 4.9 ist ein Sprung zwischen Blöcken zu sehen. Wird der Sprung genommen, so wird von Block b_0 zu Block b_2 gesprungen, ansonsten zu b_1. NT (*not taken*) gibt an, wie oft von b_0 zu b_1 gesprungen wird, T (*taken*) respektive für die Sprünge von b_0 zu b_2.

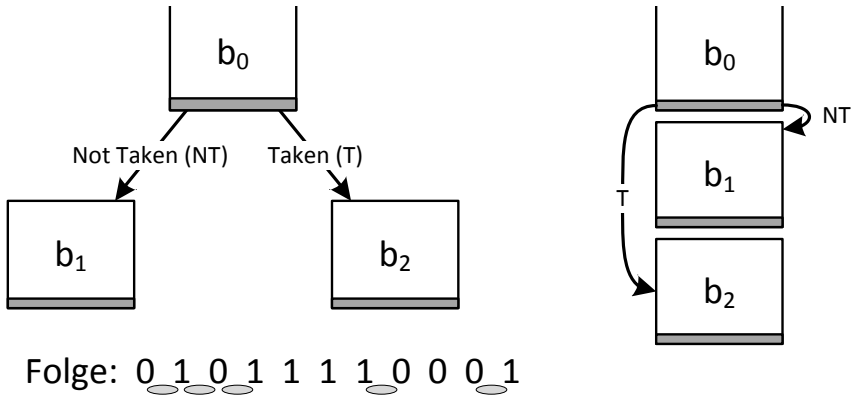


Abbildung 4.9.: Bedingter Sprung zwischen Blöcken mit Sprungabfolge (links Kontrollflussgraph auf Block-Ebene, rechts Anordnung im Speicherabbild)

Das Verhalten des Sprunges lässt sich als Bitvektor beschreiben, bei einer 1 wurde der Sprung angenommen, bei einer 0 nicht. Mit diesem Bitvektor lässt sich berechnen, wie oft der Sprung sein Verhalten geändert hat. Die Folge in Abbildung 4.9 hat fünf Verhaltenswechsel.

Schwer vorhersagbare Sprünge müssen auch erkannt werden, wenn man versucht, Verfahren zur Sprungvorhersage zu optimieren. In diesem Zusammenhang haben Vintan et al. [166] zwei Maßzahlen eingeführt, sie werden als *distribution index* und *polarization index* bezeichnet. Davon abgeleitet sind die Größen *Relevanz* und *Volatilität*. Verändert wurde v. a. der Wertebereich, der jetzt für beide Größen zwischen 0 und

1 liegt:

- Relevanz $relevance := \frac{2*min(T,NT)}{T+NT}; \mathbb{W} = [0...1]$
  wenn nahe 0: T und NT unterschiedlich, es gibt einen selten besuchten Zweig
  wenn nahe 1: T und NT ähnlich, Zweige ähnlich häufig besucht
- Volatilität $volatility := \frac{switches}{2*min(T,NT)}; \mathbb{W} = [0...1]$
  wenn nahe 0: seltene Änderung des Verhaltens
  wenn nahe 1: häufige Änderung des Verhaltens

Die im GAP verwendete bimodale Sprungvorhersage (vgl. Kapitel 2.1.2 und 2.3.3) zeigt eine hohe Vorhersagegenauigkeit vor allem dann, wenn der Sprung einen selten besuchten Zweig hat und/oder der Sprung sein Verhalten selten ändert. Schwierig vorherzusagen ist also das Verhalten eines Sprunges mit hoher Relevanz oder hoher Volatilität.

Parameter für die Heuristik

Für weitere Analysen wurden aus sechs Benchmarks der MiBench Benchmark-Suite 96 bedingte vorwärtsgerichtete Sprünge isoliert, die sehr häufig ausgeführt werden. Zu diesen Sprüngen wurde ermittelt, wie oft sie pro Ausführung Löschungen des Arrays (bei Ausführung mit einer einzigen Konfigurationsebene) verursachen, also der Sprung nicht korrekt vorhergesagt werden konnte. Abbildung 4.10 zeigt diese Daten, die Sprünge sind nach Anzahl der Löschungen pro Ausführung absteigend sortiert.

Darin ist klar zu erkennen, dass die Löschungen pro Ausführung mit der Relevanz des Sprunges zusammenhängen. Außerdem verfügen Sprünge, die zu vielen Array-Löschungen führen, oft auch über einen relativ hohen Wert für die Größe Volatilität. Damit ein Sprung im Weiteren als schwer vorhersagbar klassifiziert wird, muss seine Volatilität über 0,2 liegen und der Wert für die Relevanz über 0,35. Diese Grenzwerte sind durch eine automatische Suche im Parameterraum bestätigt.

Somit können mutmaßlich schwer vorhersagbare Sprünge identifiziert werden. Für diese Sprünge soll dann die bedingte Ausführung angewendet werden. Dazu wird dieselbe Heuristik wie im Abschnitt 4.2.3 mit den Grenzwerten G^{map}_{total} und G^{map}_{loop} verwendet.

In den Abbildungen 4.11(a) und 4.11(b) sind wieder die Zusammenhänge zwischen diesen Parametern und der Performance (CPI, weniger ist besser) für 10 Benchmarks ausgeführt auf GAP mit 8x8x8-Array zu sehen. G^{total}_{max} darf nicht zu klein gewählt werden, denn dann verschlechtert sich die Performance wieder.

Performance-Auswertung

Abbildung 4.12 zeigt für einige Konfigurationen des GAP diejenigen Parameter-Kombinationen, mit denen eine Performance erreicht werden kann, die maximal zwei Prozent von dem für diese Konfiguration bekannten Optimum abweicht. Bereiche mit guten Parameter-Konfigurationen liegen für G^{max}_{total} zwischen 7 und 18 und für G^{max}_{loop}

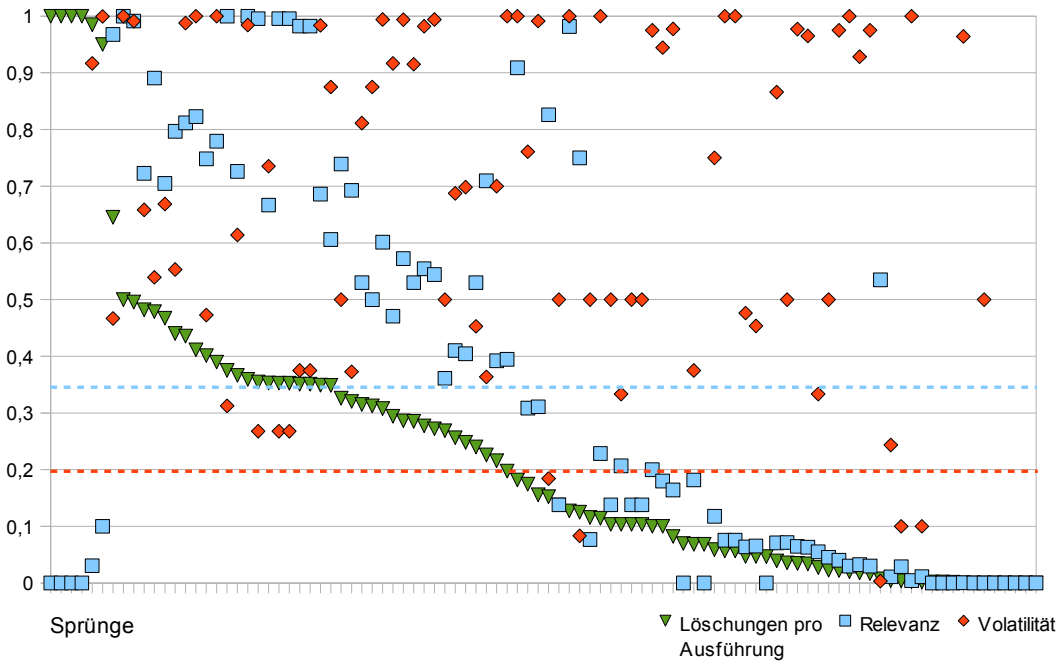
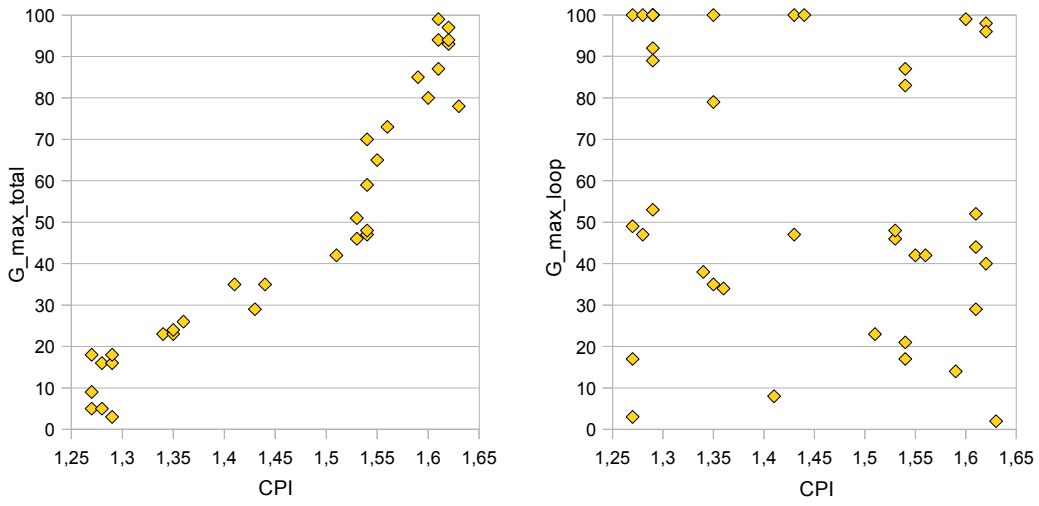


Abbildung 4.10.: Verteilung der Werte für Relevanz und Volatilität für 96 ausgewählte Sprünge und daraus resultierende Anzahl an Neukonfigurationen pro Ausführung



(a) Zusammenhang zwischen CPI und G_{max}^{total} (b) Zusammenhang zwischen CPI und G_{max}^{loop}

Abbildung 4.11.: Parameter der Heuristik für die Selektion von Befehlen für die bedingte Ausführung in Kombination mit der Sprungvorhersage und ihre Auswirkung auf die Gesamt-Performance (GAP mit 8x8x8-Array)

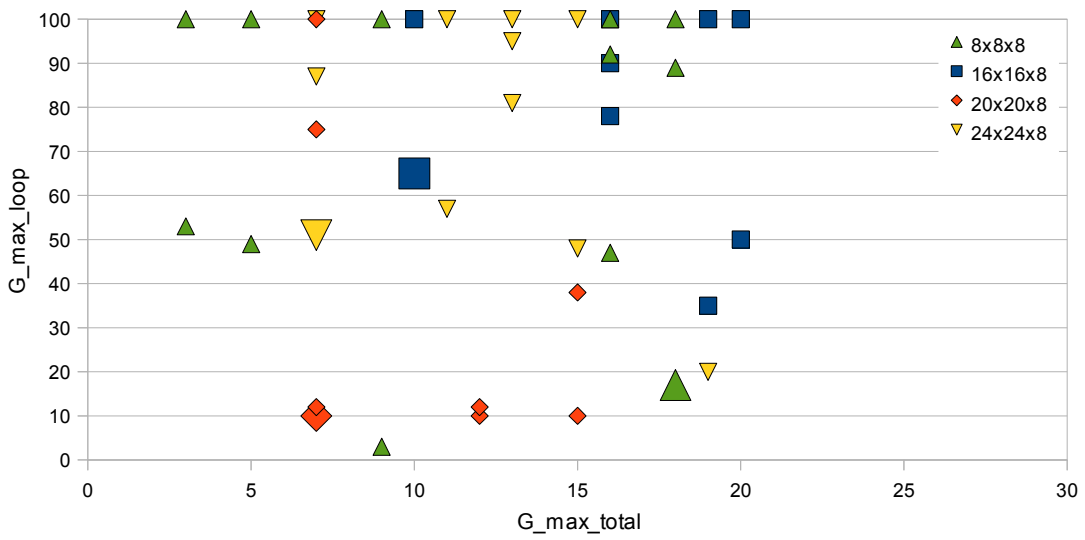


Abbildung 4.12.: Parameter-Werte für Heuristik für verschiedene Array-Größen des GAP, die maximal 2% schlechter sind als das jeweils bekannte Optimum

zwischen 12 und 100. Sprünge, die bei der Kombination mit der Sprungvorhersage mit der bedingten Ausführung behandelt werden, dürfen somit länger sein als Sprünge, die ohne Kombination mit der Sprungvorhersage mit der bedingten Ausführung behandelt werden. Diese Beobachtung deckt sich mit dem Ziel der Kombination.

Die mittlere Performance, die mit der Kombination von bedingter Ausführung und der Sprungvorhersage erreicht werden kann, ist meist sehr ähnlich zu den Werten, die nur mit der Sprungvorhersage erreicht werden können (vgl. Abbildung 4.13). Betrachtet man die einzelnen Benchmarks (siehe Abbildung 4.14), so fällt auf, dass bei den meisten Benchmarks mit der Kombination aus Sprungvorhersage und bedingter Ausführung die Performance bei der Verwendung der Sprungvorhersage allein erreicht wird. Es gibt nur wenige Benchmarks, bei denen der erreichte IPC-Wert unter oder über dem Wert für die Sprungvorhersage liegt. Im Mittel ist die mit der Kombination erzielbare Performance in jedem Fall über der, die mit der bedingten Ausführung erreicht wird. Für die beiden ADPCM-Benchmarks, bei denen mit der bedingten Ausführung überragende Ergebnisse erzielt werden konnten, können jetzt nur für einen der Benchmarks vergleichbare Ergebnisse erzielt werden.

Die Ursachen dafür, dass der Performance-Gewinn, der mit der Kombination erzielt werden kann, so gering ausfällt, sind vor allem, dass der GAP über vielfältige Verfahren verfügt, mit denen Sprünge behandelt werden können und dass, auch wenn schwer vorhersagbare Sprünge vorhanden sind, diese nicht prinzipiell mit der bedingten Ausführung behandelt werden können. Jedoch kann die Kombination aus bedingter Ausführung und Sprungvorhersage eigentlich ihre Ziele erfüllen, da weniger Instruktionen auf das Array abgebildet werden müssen, das Array seltener neu konfiguriert werden muss, die durchschnittliche Konfigurationslänge steigt und die

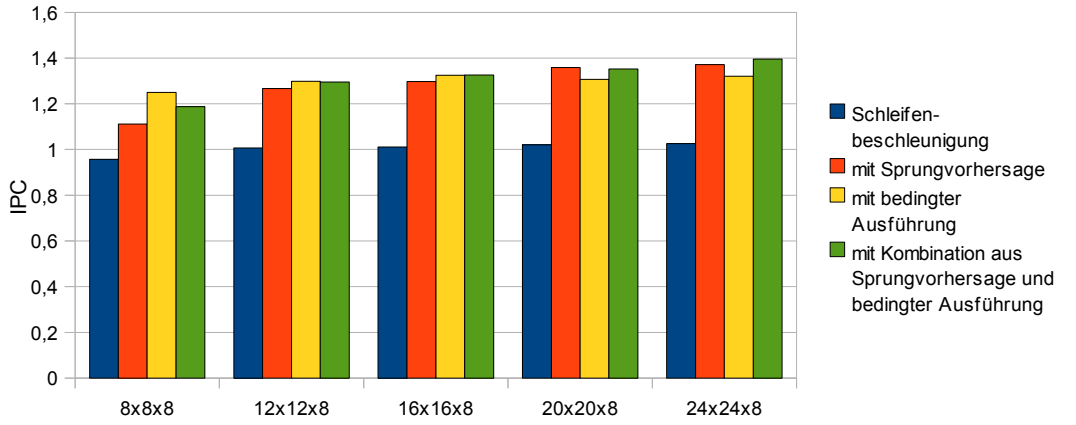


Abbildung 4.13.: Durchschnittliche Performance für 17 MiBench-Benchmarks mit verschiedenen Verfahren zur Sprungbehandlung auf GAP mit unterschiedlichen Array-Größen

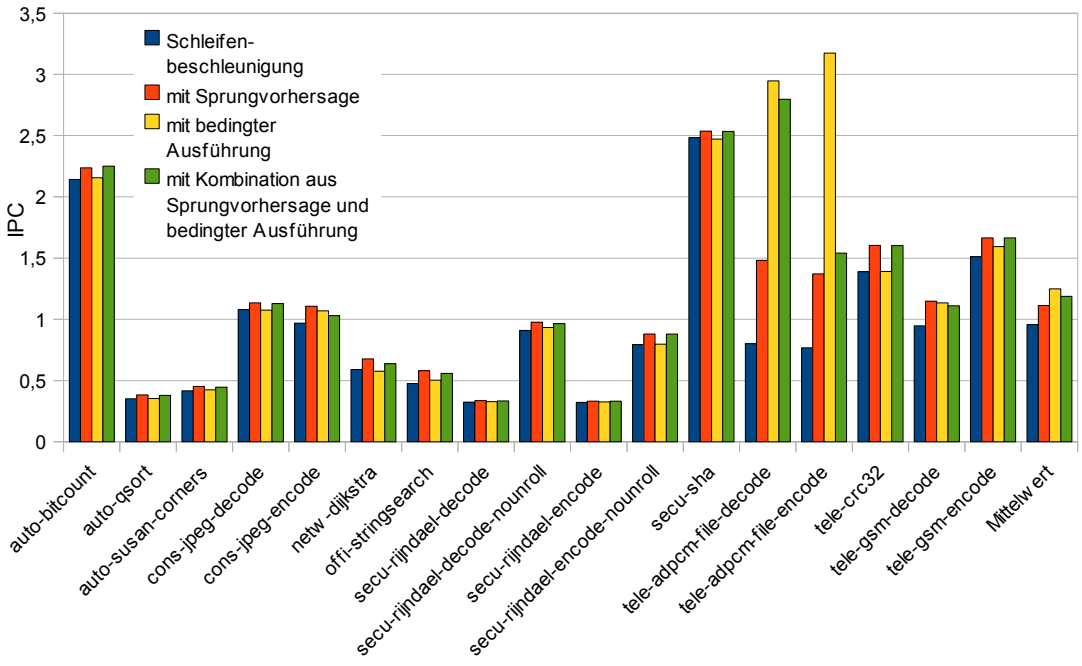


Abbildung 4.14.: Durchschnittliche Performance für 17 MiBench-Benchmarks mit unterschiedlichen Verfahren zur Sprungbehandlung, GAP mit 8x8x8-Array

Hit-Rate der Konfigurationsebenen höher ist (siehe Anhang A.4).

Eine Evaluierung mit dem SimpleScalar-Simulator wäre sehr interessant, ist aber nicht möglich, da für die bedingte Ausführung keine Unterstützung vorhanden ist und nicht einfach nachgerüstet werden kann, da hier auch *value forwarding* etc. berücksichtigt werden müssten.

4.2.5. Verwandte Arbeiten

Die bedingte Ausführung ist nicht neu (vgl. [5, 122, 159]) und wird als effektives Verfahren gesehen um die Limitationen durch den Kontrollfluss zu reduzieren (vgl. [90], [104]). Sie kann so umgesetzt werden, dass alle Instruktionen des Befehlssatzes Prädikate verändern können (*full predication*) oder nur ausgewählte Gruppen (*partial predication*), z. B. Speicherbefehle. Mahlke et al. vergleichen verschiedene Umsetzungen [105]. Es ist keine Implementierung als Post-Link Optimierung bekannt, bei der im Fokus steht, dass weder Instruktionen in den Programmcode eingefügt werden müssen noch die Reihenfolge von Instruktionen verändert wird.

Vernachlässigt man diese Anforderungen, so kann die bedingte Ausführung weitere Optimierungen ermöglichen oder positiv beeinflussen, z. B. Scheduling mit dem Hyperblock [106] und Software Pipelining [167]. Zudem kann bei der Programmausführung mit Prädikaten der Wert von Prädikaten vorhergesagt werden, um die Performance weiter zu erhöhen [31, 127].

Aktuelle Prozessoren, die Unterstützung für die bedingte Ausführung bieten, sind z. B. Intel Itanium [30], ARM Prozessoren [29] und einige Prozessoren von IBM, also POWER bzw. PowerPC.

Auch der TRIPS-Prozessor unterstützt *Predicated Exection* (siehe [111]). Hier kann (beinahe) jeder Befehl Prädikate setzen, es handelt sich deshalb um *full predication*.

Neben dem Gezeigten existieren weitere Ansätze zur Kombination von bedingter Ausführung und Sprungvorhersage, z. B. die *Wish-Branches* von Hyesoon et al. [84]. Dabei wird das Programm zur Ausführung mit der bedingten Ausführung vorbereitet und es wird erst zur Laufzeit entschieden, ob die Sprungvorhersage genutzt oder ob die bedingte Ausführung verwendet wird. Es lassen sich damit gute Ergebnisse erzielen, es sind aber starke Eingriffe in die Hardware nötig.

4.2.6. Zusammenfassung

Einen zusammenfassenden Überblick über die Performance von bedingter Ausführung allein und in Kombination mit Sprungvorhersage gibt Grafik 4.13.

Alles in allem hat die gezeigte Implementierung von bedingter Ausführung die Eigenschaften, die erwartet wurden. Sprünge können mit geringeren Verzögerungen behandelt werden und die durchschnittliche Länge der Konfigurationen nimmt zu, das Array aus Funktionseinheiten wird also effektiver verwendet. Mit ausschließlich bedingter Ausführung kann beinahe vergleichbare Performance wie mit der Sprungvorhersage erreicht werden, sie wird bei einzelnen Benchmarks übertroffen. Mit der Kombination aus bedingter Ausführung und Sprungvorhersage ist die durchschnittliche Performance im Schnitt etwas höher als nur mit Sprungvorhersage.

Der Hauptgrund für die geringen Performance-Gewinne der bedingten Ausführung sind die Konfigurationsebenen, die stark dazu beitragen, auch bei falsch vorhergesagten Sprüngen die Verzögerungen niedrig zu halten. Das gilt ebenso für die Kombination mit der Sprungvorhersage, da hier bei häufig ausgeführten schwer vorhersagbaren Sprüngen, die also auch einen gewissen Einfluss auf die Gesamtperformance haben, die Wahrscheinlichkeit hoch ist, dass beide mögliche Sprungziele bereits in Konfigurationsebenen geladen wurden.

Ein Vorteil der bedingten Ausführung gegenüber der Sprungvorhersage könnte sein, dass in Umgebungen mit harten Echtzeitanforderungen die Dauer der Ausführung für ein Programmstück wahrscheinlich besser abschätzbar ist, da die Ausführungszeit nicht allein vom Zustand des Prozessors bzw. den Interna der Sprungvorhersage abhängig ist.

4.3. Funktionseinbindung

In diesem Abschnitt werden die Funktionseinbindung (weitere Bezeichnungen: *inline expansion*, *inlining of functions*, Inline-Ersetzung, Inline-Expansion) und ihre Effekte untersucht. Nach Einführung und Motivation in Abschnitt 4.3.1 wird die Umsetzung in GAPtimize erklärt (Abschnitt 4.3.2) und evaluiert (Abschnitt 4.3.3). Abschließend werden in Abschnitt 4.3.4 verwandte Arbeiten vorgestellt und abgegrenzt sowie die Ergebnisse zusammengefasst (Abschnitt 4.3.6).

4.3.1. Einführung und Ziele

Die Funktionseinbindung ist eine relativ gebräuchliche und seit langer Zeit bekannte Code-Optimierung, bei der Funktionsaufrufe durch Kopien des Funktionskörpers ersetzt werden. Abbildung 4.15 zeigt ein Beispiel. Ziel ist es, trotz Vergrößerung des Programmcodes und somit einem höheren Risiko für Verzögerungen beim Laden von Instruktionen Effekte zu erzielen, die die Programmausführung beschleunigen:

- Durch weniger Funktionsaufrufe verringert sich die Anzahl langer Sprünge, das kann das Risiko von Cache-Konflikten reduzieren. Erfolgt die Funktionseinbindung durch einen Compiler, so kann oft auf *Spill-Code* verzichtet werden, der die Inhalte aus Registern sichert und wiederherstellt.

- Durch weniger Funktionsrücksprünge sind weniger indirekte Sprünge vorhanden, die im GAP nicht vorhergesagt werden können.

- Einzelne Kopien des Funktionskörpers können durch weitere Optimierungen unterschiedlich und je nach Situation angepasst werden (Spezialisierung, z. B. Ersetzung von Variablen durch feste Werte).

Auch Standard-Compiler wie der GCC unterstützen die Funktionseinbindung. Sie stoßen dabei aber oft an ihre Grenzen, wenn nur einzelne Module nacheinander übersetzt werden. Dann kann die Funktionseinbindung nur auf der Ebene dieser Module durchgeführt werden; Funktionen aus z. B. der Standardbibliothek können nicht eingebunden werden.[4]

Die eigentliche Herausforderung bei der Funktionseinbindung ist es, aus der Menge aller Funktionsaufrufe im Programm eine Teilmenge zu finden, für die eine Ersetzung zielgerichtet ist. Einerseits kann jeder Funktionsaufruf ersetzt werden – ein sehr aggressives Vorgehen. Andererseits ist es sinnvoll zu beachten, welche Aufrufe ersetzt werden, um keine zu großen Programmdateien zu erzeugen. Ein aggressives Vorgehen kann vor allem dann zu Problemen führen, wenn der Aktionsbereich sehr groß gewählt wird.

[4] Mit Version 4.5 wurde GCC so erweitert, dass unter Anwendung spezieller Parameter auch zur Link-Zeit Optimierungen durchgeführt werden können. Das hebt die erwähnte Beschränkung teilweise auf. Für den GAP kann diese Version des GCC nicht verwendet werden.

4.3.2. Umsetzung in GAPtimize

In GAPtimize ist die Funktionseinbindung als *Whole-Program-Optimization* (Optimierung des gesamten Programms, WPO) implementiert. Es wird also nicht wie im GCC auf Modulebene gearbeitet, sondern der Bereich, in dem Änderungen stattfinden können, ist das gesamte Programm. Somit können Funktionsaufrufe ersetzt werden, die für den GCC wegen der Beschränkung des Arbeitsbereichs nicht erreichbar sind.

Das Hauptaugenmerk liegt darauf, die Menge an Funktionsaufrufen, die eingebunden werden soll, gut zu wählen. Die Funktionseinbindung an sich ist relativ problemlos zu erledigen, es müssen aber abschließend alle Referenzen im Programm (Sprünge, Sprungtabellen) angepasst werden (vgl. Kapitel 3.6).

Zur Klassifizierung der einzubindenden Funktionsaufrufe bieten sich statische Informationen an, z. B. die Länge der aufgerufenen Funktion gemessen in Instruktionen, und zudem auch dynamische Informationen, die in GAPtimize nicht aber in normalen Compilern zur Verfügung stehen, z. B. wie oft ein Funktionsaufruf ausgeführt wird.

Die Heuristik, mit der im Folgenden die Teilmenge der Funktionsaufrufe ausgewählt wird, die dann eingebunden werden, verfügt über vier Parameter. Sie sind mit Beschreibung und dem Wertebereich in Tabelle 4.2 aufgeführt. Mit vier Parametern ist sie vergleichsweise einfach aufgebaut (vgl. Abschnitt 4.3.4).

Parameter	Definitionsbereich	Beschreibung
`max_caller_count`	$\{0, 1, 2, 3, ..., 100\}$	Höchstzahl der einzubindenden Funktionsaufrufe
`weight_of_caller`	$\{0, 1, 2, 3, ..., 100\}$	Mindestgewicht, das Funktionsaufruf besitzen muss, um eingebunden zu werden
`length_of_function`	$\{0, 25, 50, 75, ..., 10000\}$	Obergrenze für Länge der einzubindenden Funktion
`insns_per_caller`	$\{0, 1, 2, 3, ..., 200\}$	Anzahl der statischen Instruktionen, die pro Ausführung des Funktionsaufrufs durch Einbindung höchstens hinzugefügt werden dürfen

Tabelle 4.2.: Parameter der Heuristik zur Auswahl von Funktionsaufrufen

4.3.3. Evaluierung

Die Auswertung der Funktionseinbindung findet in mehreren Schritten statt. Erst muss untersucht werden, welche Parameter für die Heuristik verwendet werden sollen und ob sie von der Hardware abhängig sind. Anschließend werden die Ergebnisse, die mit diesen Parametern für mehrere Benchmarks und Hardware-Konfigurationen ermittelt werden, beurteilt und es wird geklärt, ob die erstrebten Ziele der Funktionseinbindung erreicht werden können. Abschließend werden die selben Benchmarks

auch mit dem SimpleScalar-Simulator ausgeführt, um zu klären, ob die erzielten Performance-Änderungen GAP-spezifisch sind oder nicht. Es wird stets die Sprungvorhersage verwendet.

Die Performance des Prozessors sollte nicht mehr mit *Clocks per Instructions (CPI)* oder *Instructions per Cycle (IPC)* gemessen werden, da sich die Anzahl der ausgeführten Instruktionen mit der Funktionseinbindung ändern kann. Stattdessen wird die Zahl *Clocks per reference Instructions (CPRI)* ermittelt, wobei die Anzahl der Referenzinstruktionen den ausgeführten Instruktionen für das nicht-veränderte Programm entspricht. Interessanter ist aber eigentlich, um welchen Prozentsatz sich die Gesamtausführungszeit der Benchmarks verändert, denn das ist der Gewinn durch die Funktionseinbindung.

Parameter für die Heuristik

Bei ersten Versuchen zur Bestimmung optimaler Heuristik-Parameter hat sich gezeigt, dass bei Verwendung mehrerer Benchmarks der Einfluss von `length_of_function` und `max_caller_count` äußerst gering ist. Deshalb wird `length_of_function` fest auf 10000 und `max_caller_count` auf 100 gesetzt. Beide Werte sind die oberen Grenzen des jeweiligen Wertebereichs und führen zu keinen Einschränkungen bei der Auswahl der Funktionsaufrufe.

In den Grafiken 4.16 und 4.17 ist zu sehen, welche Performance mit einigen Werten für `insns_per_caller` und `weight_of_caller` erzielt werden kann. Hierfür wurde das *Framework for Automatic Design Space Exploration* (FADSE, siehe [20], [18] und [80]) verwendet und der genetische Algorithmus NSGA-II (siehe [47]). Zu den Ergebnissen wurden außerdem einige manuell selektierte Punkte hinzugefügt. Es wird der Durchschnittswert für CPRI über zehn Benchmarks errechnet. Als Benchmarks werden die bereits vorgestellten 10 MiBench-Benchmarks verwendet, die ohne Funktionseinbindung aber mit allen sonstigen Optimierungen (`-O3 -fno-inline`) kompiliert wurden. Grund dafür ist, dass für diese ersten Tests möglichst viele Möglichkeiten zur Funktionseinbindung vorhanden sein sollen.

Bei der Auswertung ist zu beachten, dass beide Parameter voneinander abhängig sein können. Somit sind optimale Konfigurationen in den Bereichen der Diagramme zu finden, die für die einzelnen Konfigurationen auf der X-Achse möglichst kleine Bereiche aufweisen.

In Grafik 4.16 ist leicht erkennbar, dass mit hohen Werten für `insns_per_caller` nie gute Performance-Werte erzielt werden können. Das gilt für alle getesteten GAP-Konfigurationen. Mit `insns_per_caller` auf 1 können für alle Konfigurationen sehr gute Werte erreicht werden.

Schwieriger ist die Auswertung für Abbildung 4.17, also für den Parameter `weight_of_caller`. Die vertikalen Linien im Bild zeigen an, dass nicht-optimale Performance mit beinahe allen Werten für diesen Parameter erreicht werden kann. Daraus kann man folgern, dass er sich nicht so stark auf die Performance wie `insns_per_caller` auswirkt. Betrachtet man die Werte der Parameter bei sehr guten Performance-Werten, so bietet es sich an, `weight_of_caller` auf 30 zu setzen, wobei der Unterschied zu 1

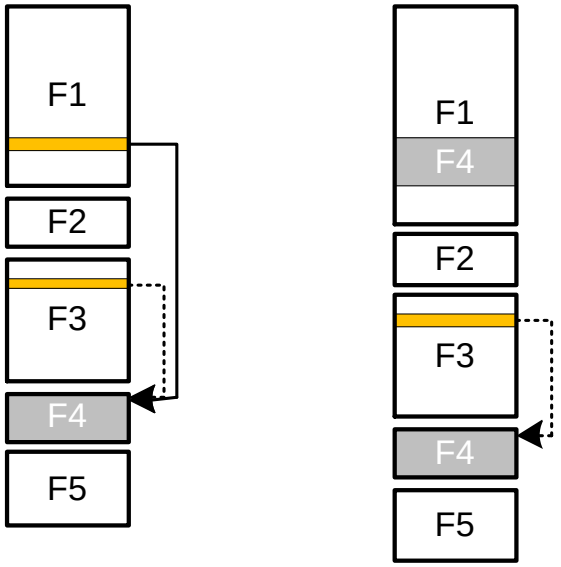


Abbildung 4.15.: Beispiel für Funktionseinbindung: Der Aufruf der Funktion F4 in der Funktion F1 (links) wird durch eine Kopie des Funktionskörpers von F4 ersetzt (rechts).

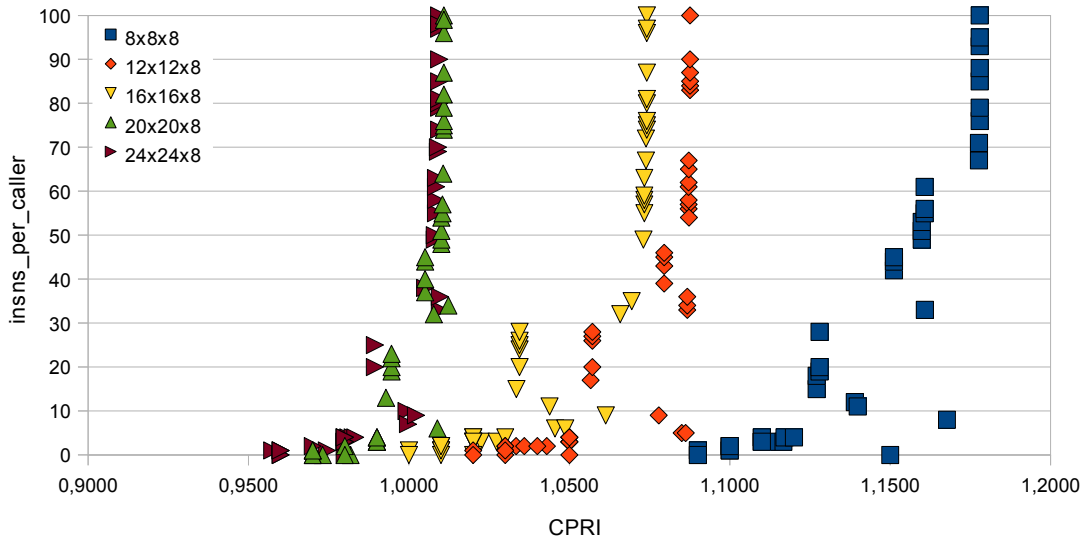


Abbildung 4.16.: Einfluss des Parameters `insns_per_caller` auf die Gesamtperforamnce bei 10 Benchmarks ohne Funktionseinbindung durch GCC für GAP mit unterschiedlichen Array-Größen

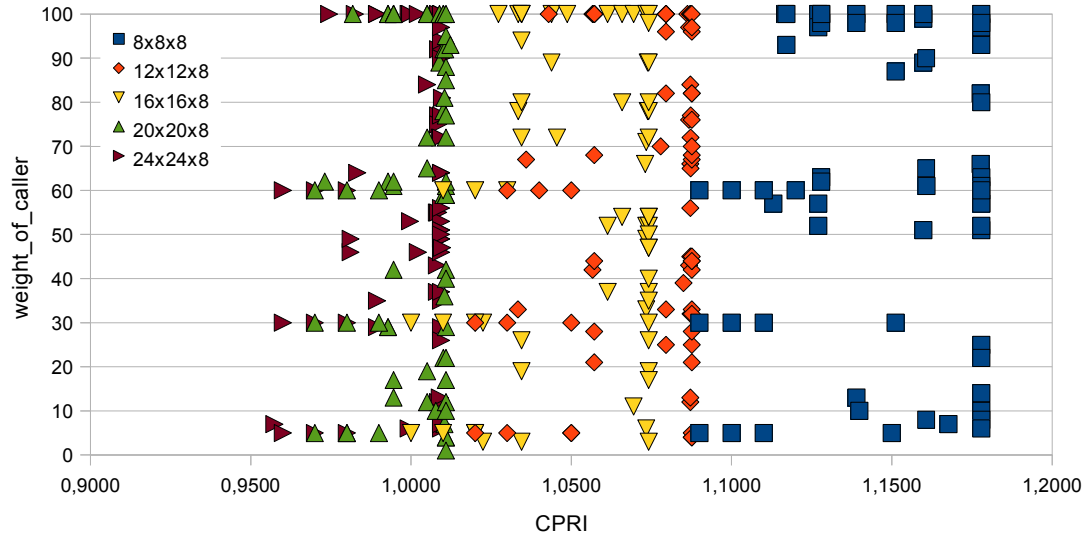


Abbildung 4.17.: Einfluss des Parameters `weight_of_caller` auf die Gesamtperforamnce bei 10 Benchmarks ohne Funktionseinbindung durch GCC für GAP mit unterschiedlichen Array-Größen

oder 60 aber marginal ist.

Beide Diagramme zeigen keine Beeinflussung der optimalen Parameter-Werte durch die Array-Größe des GAP, also sind die Parameter davon unabhängig. Weitere Untersuchungen zeigten, dass auch die Anzahl der Konfigurationsebenen keinen Einfluss hat.

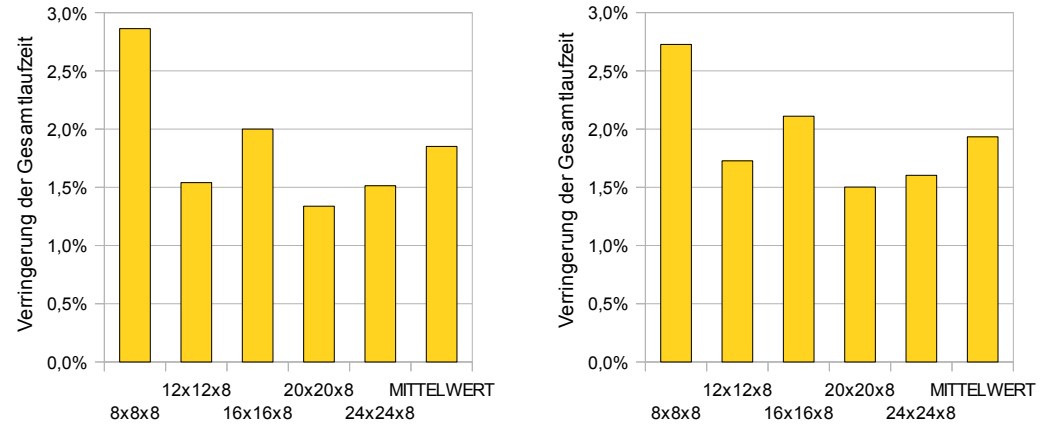
Performance-Auswertung für den GAP

Mit diesen optimalen Parametern werden jetzt 17 statt bisher 10 Benchmarks ausgeführt. Die Programme werden diesmal mit Inlining kompiliert. Die Übersicht über die Performance-Verbesserung – jetzt als Durchschnitt der relativen Verbesserung der einzelnen Benchmark-Laufzeiten gemessen – ist in Abbildung 4.18(a) dargestellt.

Ergänzend zeigt Abbildung 4.18(b) den durchschnittlichen Performance-Gewinn, der mit der Funktionseinbindung durch GAPtimize erreicht werden kann, wenn zuvor GCC keine Funktionseinbindung durchgeführt hat. Erstaunlicherweise sind die Unterschiede zu Abbildung 4.18(a) sehr gering. Anscheinend kann GCC nur sehr wenige Funktionsaufrufe auflösen, da nur bei wenigen Aufrufen das Ziel im zu kompilierenden Programmteil zu finden ist.

Details für die Performance-Verbesserung bei Ausführung auf dem GAP mit einem Array aus 8x8 Funktionseinheiten und 8 Konfigurationsebenen zeigt Abbildung 4.19. Bei einigen Benchmarks tritt keine Änderung der Laufzeit auf. Das liegt oft daran, dass im jeweiligen Benchmark nur sehr wenige Funktionsaufrufe überhaupt enthalten und deshalb keine Ansatzpunkte für die Optimierung vorhanden sind (siehe Abbildung 4.20).

Die Ursachen für die Performance-Gewinne sind v. a. die höhere Parallelität, mit



(a) Durchschnittlicher Performance-Gewinn für 17 Benchmarks, kompiliert mit normalen Parametern

(b) Durchschnittlicher Performance-Gewinn für 17 Benchmarks, die ohne Funktionseinbindung kompiliert wurden

Abbildung 4.18.: Durchschnittlicher Performance-Gewinn durch Funktionseinbindung bei 17 Benchmarks und verschiedenen Array-Konfigurationen

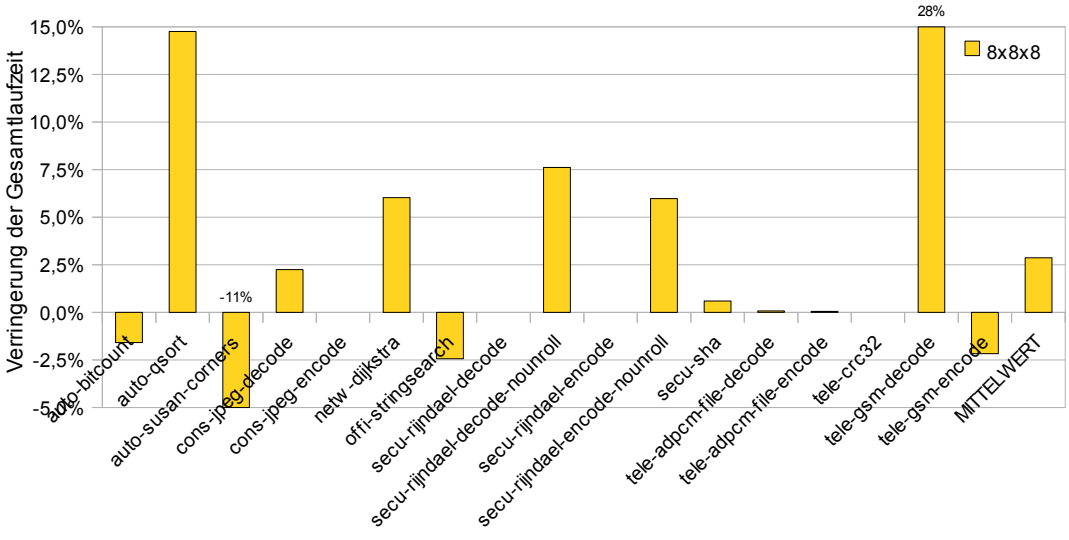


Abbildung 4.19.: Performance-Veränderung durch die Funktionseinbindung für GAP mit Array 8x8x8

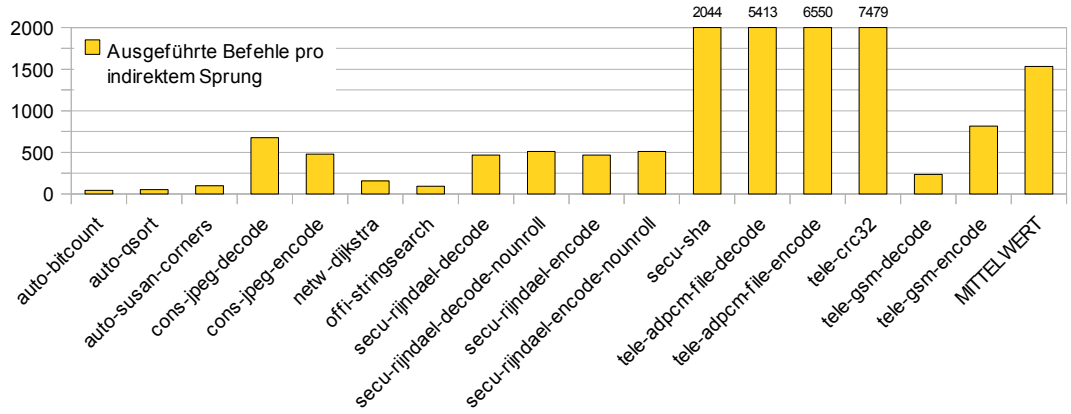


Abbildung 4.20.: Ausgeführte Instruktionen pro indirektem Sprung (wichtigster Teil darin: Funktionsrücksprünge)

der die Instruktionen ausgeführt werden können (sichtbar an IPC) und die niedrigere Anzahl an Sprüngen, die zu Löschungen des Arrays führen. Die durchschnittliche Konfigurationslänge und die Anzahl an Instruktionen, die auf das Array abgebildet werden, ändern sich nur geringfügig (siehe Anhang A.6).

Vergleich Performance-Veränderung bei GAP und SimpleScalar

Um zu vergleichen, ob die hier vorgestellte Funktionseinbindung auf einem „normalen“ superskalaren *out-of-order*-Prozessor ähnliche Effekte hat, sind einige Benchmarks, die mit GAPtimize verändert wurden, auf dem SimpleScalar-Simulator ausgeführt worden. Dazu wurde eine angepasste Version verwendet, die nicht nur ausführbare Dateien laden kann, sondern auch die Speicherabbilder, die GAPtimize erzeugt. Der Out-of-Order-Simulator wurde so konfiguriert, dass die Pipeline- und Cache-Parameter weitgehend mit denen des GAP übereinstimmen; Details sind im Anhang A.5 zu finden.

Abbildung 4.21 zeigt den Performance-Gewinn durch die Funktionseinbindung für den GAP mit einem 8x8x8-Array und SimpleScalar. Es lässt sich kein klarer Trend ableiten. Auffallend ist der Ausreißer für `secu-rijndael-encode-nounroll`, hier ist die veränderte Programmversion deutlich schlechter, wenn sie auf dem SimpleScalar ausgeführt wird. Ansonsten sind die Veränderungen für SimpleScalar meist deutlich größer, egal ob positiv oder negativ. Auffallend ist bei einer genaueren Betrachtung der Performance-Counter des SimpleScalar, dass eine Reduktion der indirekten Sprünge weder mit dem IPC noch mit der Gesamtausführungszeit in Zusammenhang steht, die Gesamtausführungszeit aber durch den Befehlszwischenspeicher stark beeinflusst wird (siehe Anhang A.6).

4.3.4. Verwandte Arbeiten

Die Analyse der Auswirkungen der Funktionseinbindung auf Programmebene auf die Performance des GAP stehen im Mittelpunkt. Dazu wird eine Heuristik vorge-

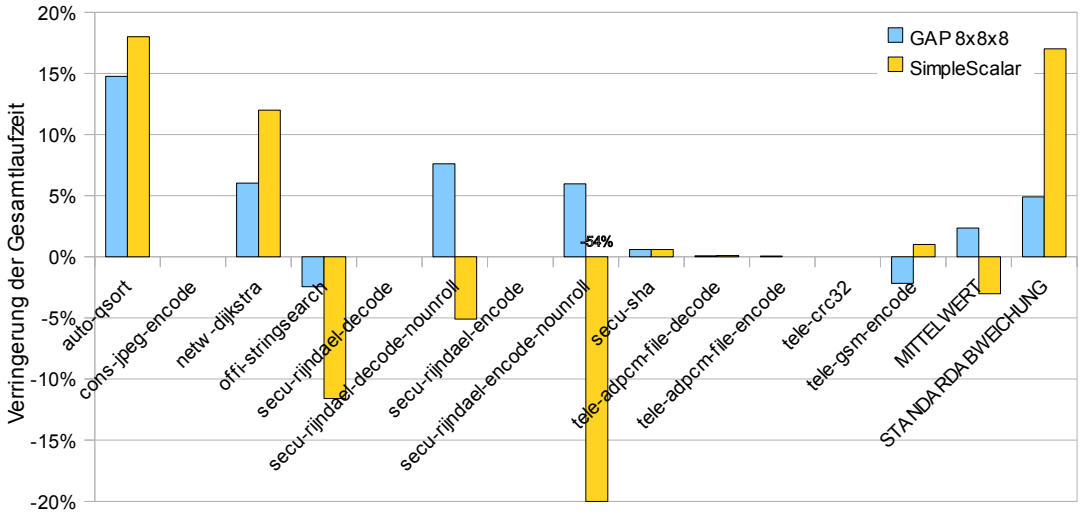


Abbildung 4.21.: Vergleich der Performance-Verbesserung durch die Funktionseinbindung mit GAPtimize für GAP mit 8x8x8-Array und SimpleScalar

schlagen, deren Parameter dann mit Methoden des *Machine Learning* optimiert werden. In diesem Kontext gibt es einige verwandte Arbeiten.

Der ALTO *link-time optimizer* kann Funktionen auf Programmebene einbinden. Es wird eine einfache Strategie verwendet, die die Funktionsgröße, die Anzahl an vorhandenen Funktionsaufrufen und die Ausführungshäufigkeit von Funktionsaufrufen berücksichtigt. Bei geringen Änderungen der Code-Größe von ca. 1% werden geringe Performance-Änderungen von weniger als 2% für SPEC95-Benchmarks erzielt [116].

Auch FDPR kann *Inlining* durchführen [13]. Dabei werden zuerst die Funktionseinbindung und anschließend das Scheduling des Codes durchgeführt. Die erzielten Performance-Gewinne im Rahmen von wenigen Prozent werden nur in Kombination von Einbindung und Scheduling erzielt; wird auf die zweite Phase verzichtet, so sind die Auswirkungen nur marginal. Man könnte meinen, mit der Funktionseinbindung werden v. a. weitere Situationen geschaffen, in denen die Sortierung wirksam arbeiten kann.

Todd Waterman stellt in seiner Doktorarbeit [168] vor, wie Inlining zusammen mit *Adaptive Compilation* verknüpft werden kann. Adaptive Compilation versucht, optimale Parameter für Optimierungen bei der Kompilierung für eine gegebene Situation bestehend aus Programm und Plattform (also Software und Hardware, für Adaptivität siehe auch [71]) zu bestimmen. Dazu wird Feedback aus vorhergehenden Schritten verwendet. Es kommen z. B. genetische Algorithmen zum Einsatz. Waterman und Cooper entwickelten eine eigene Heuristik für die Funktionseinbindung [168, 40]. Sie verfügt über relativ ähnliche Parameter wie die hier verwendete Heuristik (vgl. Abschnitt 4.3.2). Anders als Waterman und Cooper wird aber für GAPtimize versucht, einmalig gute Parameter zu finden, die dann „allgemeingültig“ sind. Das heißt, dass nicht für jedes Programm eine neue Suche begonnen werden muss.

Mit der automatischen Verbesserung einer Inline-Heuristik beschäftigt sich beispielsweise O'Boyle [26] am Beispiel einer Java-VM. Weitere interessante Arbeiten über Inlining sind [28], [173] und [9]. Über die Funktionseinbindung in GCC finden sich Informationen in der Dokumentation[5].

4.3.5. Übertragung auf andere Architekturen

In diesem Kapitel wurde die Funktionseinbindung auf Programmebene statt auf Modulebene durchgeführt. Davon können theoretisch alle Architekturen profitieren, die ebenfalls durch die normale Funktionseinbindung positiv beeinflusst werden. Dazu gehören mit Sicherheit superskalare Prozessoren und VLIW-Architekturen. Wichtig ist die globale Funktionseinbindung auch, da sie Gelegenheiten für weitere Programmoptimierungen schaffen kann.

4.3.6. Zusammenfassung

In diesem Kapitel werden die Effekte der Funktionseinbindung auf Programmebene für die Performance des GAP untersucht. Dazu wird erst eine Heuristik definiert, die entscheidet, welche Funktionsaufrufe in einem Programm ersetzt werden sollen. Sie hat vier Parameter, wovon aber nur `insns_per_caller` und `weight_of_caller` einen wesentlichen Einfluss besitzen. Anschließend werden mit FADSE und einem genetischen Algorithmus optimale Werte für die Heuristik gesucht. Demnach sollte für einzubindende Funktionsaufrufe `insns_per_caller` den Wert 1 besitzen und `weight_of_caller` den Wert 30.

Es kann eine Verbesserung der Gesamtperformance von ca. 2% im Mittel über verschiedene Konfigurationen und Benchmarks beobachtet werden. Für ein 8x8x8-Array ist 28% die maximale Verbesserung, die niedrigste ist -11% bei einem Mittelwert von 2,8%. Ein Vergleich mit Performance-Ergebnissen für den SimpleScalar-Simulator zeigt, dass die Performance-Veränderung wesentlich stärker ist, allerdings sowohl positiv wie auch negativ.

[5] GCC-Dokumentation: `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

4.4. Statische Kontrollfluss-Spekulation

Mit der *statischen spekulativen Ausführung* werden freie Funktionseinheiten verwendet, um damit in nachfolgenden Programmteilen wahrscheinlich benötigte Ergebnisse zu berechnen. Sie stehen dann mit nur sehr geringer Verzögerung bereit. Damit soll die Programmausführung beschleunigt werden.

In den folgenden Teilkapiteln wird die statische spekulative Ausführung erst eingeführt und motiviert (Abschnitt 4.4.1). Ein Beispiel für Auswirkungen und Effekte findet sich in Abschnitt 4.4.2; Details zum Algorithmus und zur Umsetzung in GAPtimize in Abschnitt 4.4.3. In Abschnitt 4.4.4 wird das Verfahren bewertet. Unterschiede zu anderen verwandten Techniken werden in Abschnitt 4.4.5 erörtert und Abschnitt 4.4.7 fasst das Kapitel zusammen.

4.4.1. Einführung und Ziele

Wird ein Programm, das mit dem Standard-Compiler GCC (siehe Kapitel 2.2.2) übersetzt wurde, auf dem GAP ausgeführt, so wird meist nur eine geringe Anzahl an Spalten pro Konfiguration und – hier ist es noch auffälliger – pro Zeile verwendet (siehe Grafik 4.22).

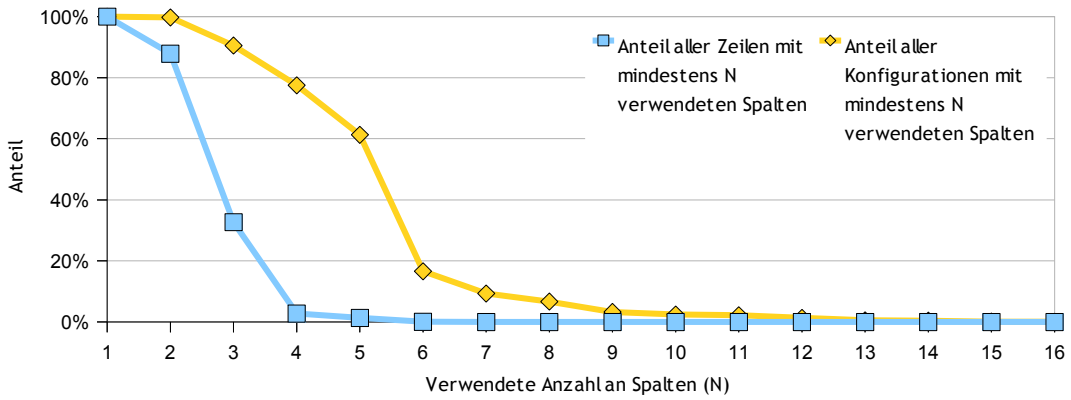


Abbildung 4.22.: Anteil aller Konfigurationen/Zeilen mit mindestens N verwendeten Spalten für Benchmark `jpeg_encode` ausgeführt auf GAP 16x16x16

Dies lässt zwei Schlüsse zu: Zum einen weist nicht besonders optimierter Code geringe Parallelität auf Befehlsebene (*Instruction Level Parallelism*, ILP) auf. Das ist ein bekanntes Problem, das sich auch bei vielen anderen Prozessoren auswirkt [90]. Zum anderen werden von den vielen Funktionseinheiten im GAP zur zeitgleichen Ausführung vorhandener Befehle nur wenige tatsächlich genutzt.

Grundidee der statischen Spekulation ist es, das zu ändern. Dazu werden nicht nur Befehle berechnet, die mit absoluter Sicherheit später ausgeführt werden, sondern auch solche, die sehr wahrscheinlich verwendet werden. Befehle, die in diesen freien Funktionseinheiten ausgeführt werden, würden sehr wahrscheinlich keine Verzögerung der anderen Befehle verursachen.

Eigentlich ist Spekulation hinsichtlich aller Abhängigkeiten denkbar, die ein Befehl haben kann und die somit seine frühzeitigere Ausführung verhindern (Kontrollfluss- und Datenabhängigkeiten, siehe Kapitel 2.1.2). Mit dem vorgestellten Verfahren wird ein Befehl früher ausgeführt, wenn all seine Datenabhängigkeiten erfüllt sind – also die Operanden bereitgestellt sind – und nur noch Kontrollflussabhängigkeiten aufgelöst werden müssen; das heißt eigentlich erst noch ein Sprung berechnet werden muss, um festzustellen, ob der Befehl benötigt wird oder nicht.

Ziel ist es, die Performance des GAP zu erhöhen. Dazu wird versucht, einen größeren Teil der Funktionseinheiten mit Instruktionen zu konfigurieren. Außerdem wird angestrebt, die Anzahl an Zeilen, die Programmteile benötigen, wenn sie im Array aus Funktionseinheiten platziert werden, zu verringern. Dadurch soll die Anzahl an Befehlen pro Konfiguration steigen und die Anzahl an Array-Löschungen für die Ausführung eines Programms sinken.

Der Algorithmus stellt keine Anforderungen an die Dominanz- oder Post-Dominanz-Beziehungen zwischen den Blöcken und seine Anwendbarkeit ist nicht dadurch eingeschränkt, ob es einen oder mehrere Vorgänger oder Nachfolger im Kontrollflussgraphen gibt. Die einzige Ausnahme sind indirekte Sprünge.

Die statische Spekulation ist als Post-Link-Optimierung (PLO) konzipiert; es wird davon ausgegangen, dass schon alle Schritte, z. B. Register-Zuweisung, Befehlsauswahl und Scheduling, bereits stattgefunden haben. Aus diesem Grund wird versucht, sowohl die benötigten Informationen für die Anwendung der Technik wie auch die auszuführenden Änderungen zu reduzieren und dadurch maximale Auswirkungen zu erreichen. Dem Algorithmus stehen Informationen zur Verfügung, die durch Binäranalyse und Profiling gewonnen werden können (siehe Kapitel 3.5.3).

Zusammenfassend soll die statische Kontrollfluss-Spekulation die Performance bei der Programmausführung erhöhen, indem die Funktionseinheiten des GAP effektiver verwendet werden und dadurch das selbe Code-Fragment mehr Spalten und weniger Zeilen verwendet. Somit sollen weniger Neukonfigurationen nötig sein und die Anzahl an parallel ausgeführten Instruktionen steigen.

4.4.2. Beispiel für statische Spekulation

Abbildung 4.23 zeigt die Anordnung der drei Blöcke einer *if-else*-Konstruktion im Array des GAP. Links ist die normale Konfiguration zu sehen, rechts nach Anwendung der statischen Spekulation. In beiden Fällen wird nach Ausführung des oberen Blockes mit einem bedingten Sprung (`BNE`) entschieden, ob die nachfolgenden Instruktionen (hellgrau) ausgeführt werden sollen oder ob sie übersprungen werden.

Klar ersichtlich ist der Einfluss der Datenabhängigkeiten zwischen den Befehlen auf ihre Anordnung im Array; die Abhängigkeiten sind durch Kanten dargestellt. Nach Sprüngen muss im GAP Synchronisation stattfinden, sie wird durch horizontale Linien verdeutlicht. Diese Synchronisation trennt Blöcke voneinander ab.

Mit der statischen Spekulation werden Instruktionen aus dem Block, der eventuell übersprungen wird (Quelle), in den (allgemeiner: jeden) vorstehenden Block verschoben (Ziel). Das Ergebnis ist in Grafik 4.23 rechts zu sehen, alle Instruktionen des

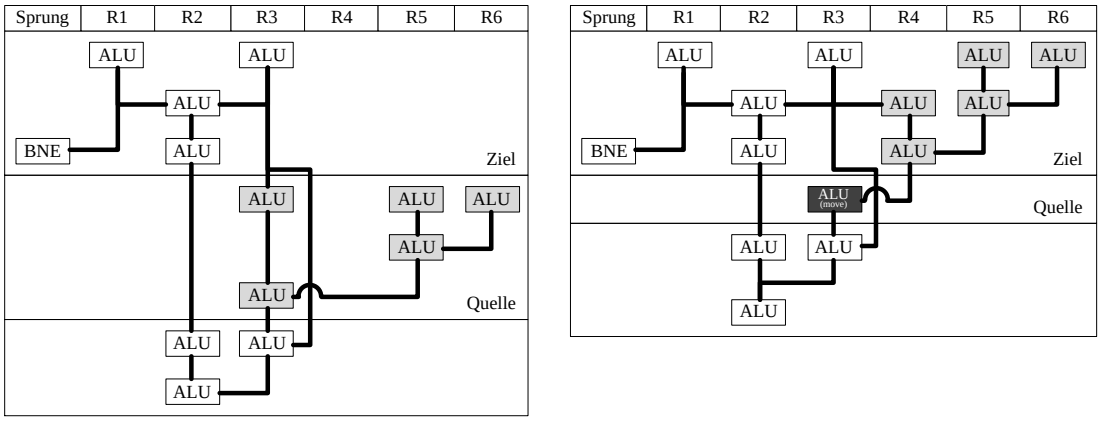


Abbildung 4.23.: Anordnung der Blöcke einer *if-else*-Konstruktion im Array des GAP vor (links) und nach Anwendung der statischen Spekulation (rechts)

bedingt auszuführenden Blocks in hellgrau sind jetzt in den vorstehenden Block verschoben worden und werden dort berechnet.

Soll der mittlere Block übersprungen werden, so dürfen die fälschlicherweise ausgeführten verschobenen Instruktionen keine Auswirkungen auf die weitere Programmausführung haben. Es muss also der Inhalt der (aktiv verwendeten) Register und des Speichers der gleiche sein, außerdem dürfen keine Systemoperationen ausgeführt worden sein. Deshalb werden schreibende Speicherzugriffe und Systemoperationen prinzipiell nicht verschoben. Um Registerwerte zu schützen werden statt des eigentlichen Zielregisters (im Bild `R3`) freie Register für spekulativ berechnete Werte derzeit aktiver Register verwendet (im Bild `R4`). Ist dann die Spekulation aufgelöst, so können die in den temporären Registern berechneten Werte in die eigentlichen Zielregister umkopiert werden (dunkelgrau im Beispiel). Auch wenn es mehrere Kopierbefehle gibt, so können sie parallel ausgeführt werden, da sie zueinander keine Abhängigkeiten aufweisen können. Deshalb ist der dadurch bedingte Overhead sehr gering.

Im Beispiel wird durch Anwendung der statischen Spekulation die Anzahl an benötigten Zeilen im Array reduziert und die Anzahl an verwendeten Spalten vergrößert. Wird der bedingte Block nicht ausgeführt, so ergibt sich keine Verzögerung durch die Code-Modifikation, da alle spekulativ auszuführenden Befehle parallel zu tatsächlich benötigten Befehlen ausgeführt werden können. Wird der Block ausgeführt, so sind die benötigten Ergebnisse bereits berechnet worden und es muss nur eine einzelne Kopieroperation ausgeführt werden, die weniger Zeit benötigt als die Ausführung der fünf Instruktionen.

Dieser Effekt ist stark vom kritischen Pfad der modifizierten Blöcke abhängig. Deshalb wird später eine Funktion zur Abschätzung der Konfigurationslängen verwendet, die die Anwendung der statischen Spekulation maßgeblich steuert.

Besondere Auswirkungen entstehen, wenn Instruktionen über einen Schleifenrücksprung verschoben werden (Beispiel: Abbildung 4.24). Dann wird der Beginn der Schleife spekulativ vor und am Ende der Schleife ausgeführt, es wird also bereits die nächste Iteration vorbereitet. Dadurch erhöht sich sehr wahrscheinlich der Grad an

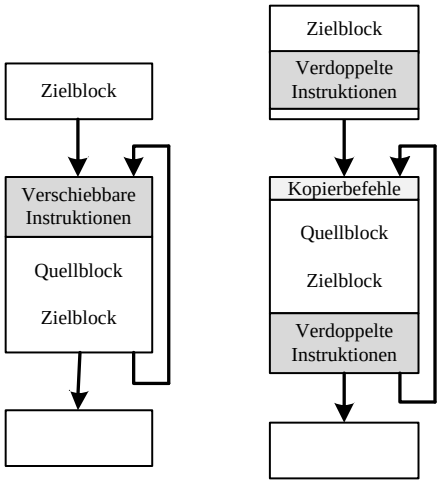


Abbildung 4.24.: Teile eines Blocks, der eine Schleife bildet, werden über den Schleifenrücksprung verschoben (links vor Anwendung der statischen Spekulation, rechts danach)

Parallelität, mit dem die Schleife ausgeführt werden kann. Schleifenabhängigkeiten werden auch berücksichtigt, da spekulativ berechnete Teile erst gültig gemacht werden, wenn klar ist, dass die Schleife ein weiteres Mal ausgeführt werden soll. Das ist sehr ähnlich zu *Software Pipelining* (SWP, siehe Abschnitt 4.6.2).

4.4.3. Algorithmus und Umsetzung in GAPtimize

Mit dem nachfolgenden Schema kann die statische Spekulation auf ein beliebiges Programm angewendet werden. Als Daten werden neben den Blöcken für das Programm der Kontrollfluss auf Instruktionen- und Block-Ebene, die Funktionstabelle und die Ergebnisse der Liveness-Analyse benötigt. Außerdem müssen Informationen über die Kantengewichte im Kontrollflussgraph vorhanden sein, sie können z. B. durch Profiling gewonnen werden. Informationen dazu finden sich in den Kapiteln 3.5.3 und 3.5.4.

Für die statische Spekulation wird zuerst eine Kandidatenliste mit Schritt 1 erstellt. Die Erstellung der Kandidatenliste kann durch verschiedene Parameter beeinflusst werden (siehe Tabelle 4.3). Optimale Werte werden im Rahmen der Evaluation (Abschnitt 4.4.4) ermittelt. Für jeden Block dieser Liste werden dann die Schritte 2 bis 5 ausgeführt.

1. Kandidatenliste

Die Kandidatenliste beinhaltet Blöcke, für die die Anwendung der statischen Spekulation sinnvoll sein könnte, in jedem Fall aber möglich ist. Dazu werden alle Blöcke ermittelt, die mindestens p_w mal ausgeführt werden und die, von einem beliebigen Vorgänger im Kontrollflussgraph aus gesehen, eine durchschnittliche Ausführungswahrscheinlichkeit von mindestens p_p (z. B. 30%) haben.

Parameter	Definitionsbereich	Beschreibung
p_w	$\{0, 10, 20, 30, ..., 1000\}$	Minimale Ausführungshäufigkeit für Quellblock
p_p	$\{0, 1, 2, 3, ..., 100\}$	Minimale durchschnittliche Ausführungswahrscheinlichkeit für Quellblock
p_b	$\{0, 1, 2, 3, ..., 50\}$	Obergrenze für Anzahl behandelter Quellblöcke

Tabelle 4.3.: Parameter der Heuristik zur Auswahl von Blöcken für die Kandidatenliste für die statische Spekulation

Um die Laufzeit des Algorithmus zu reduzieren wird nur in den Funktionen gesucht, die zusammen mindestens 80% der insgesamt ausgeführten Instruktionen abdecken (Pareto-Prinzip). Ausgeschlossen werden müssen Funktionen mit indirekten Sprüngen, da es äußerst schwer ist, alle möglichen Ziele der indirekten Sprünge zu berechnen. Außerdem dürfen die Blöcke, um zur Kandidatenliste hinzugefügt zu werden, nicht erste Blöcke einer Funktion sein, da sonst Instruktionen über Funktionsgrenzen verschoben werden müssten.

2. Bestimmung der verschiebbaren Instruktionen

Dieser und die nachfolgenden Schritte werden für jeden Block der Kandidatenliste ausgeführt. Jeder dieser Blöcke wird dann als *Quellblock* bezeichnet. Alle Blöcke, von denen aus er im Kontrollflussgraph erreicht werden kann, werden als *Zielblöcke T* bezeichnet.

In diesem Schritt wird im Quellblock die maximale Menge an Instruktionen M_{max} bestimmt, die in die Zielblöcke verschoben werden kann. Dazu wird bei der ersten Instruktion begonnen und so lange eine weitere Instruktion zu den verschiebbaren Instruktionen M hinzugefügt, bis eine der folgenden Bedingungen nicht mehr gilt:

- Jede der zu verschiebenden Instruktionen M verändert höchstens ein Register, also nicht den Speicher oder den Systemzustand.
- Es sind genügend temporäre Register verfügbar um zu verhindern, dass die Ausführung des Programms verändert wird, wenn die spekulativ ausgeführten Instruktionen M doch nicht ausgeführt werden sollen.

Während die erste Bedingung leicht zu überprüfen ist, muss für die zweite Bedingung mehr Aufwand betrieben werden. Berechnet werden müssen die Register, die umbenannt werden müssen. Das ist dann der Fall, wenn ein Register von den zu verschiebenden Instruktionen M geschrieben wird, das an mindestens einem Einfügezeitpunkt bereits aktiv (*live*, vgl. Kapitel 3.5.3) ist. Der Einfügungspunkt ist bei einem Zielblock $t_1 \in T$ ohne Sprung am Blockende, bei einem Zielblock $t_2 \in T$ mit Sprung vor eben diesem Sprung.

Die Menge R_{aktiv} der aktiven Register wird angenähert durch die Register, die am Ende mindestens eines Zielblocks $t \in T$ aktiv sind, zusammen mit den Registern, die eine eventuell vorhandene Sprunginstruktion t_{last} am Ende eines Blocks $t \in T$ verwendet:

$$R_{aktiv} = \bigcup_{t}^{T} (live_{out}(t) \cup used(t_{last}))$$

Zusätzlich wird die Menge der Register $R_{geschrieben}$ benötigt, die von den zu verschiebenden Instruktionen M geschrieben werden:

$$R_{geschrieben} = \bigcup_{m}^{M} defined(m)$$

Damit kann dann als Schnittmenge der aktiven Register R_{aktiv} und der Register $R_{geschrieben}$, die geschrieben werden, die Menge der Register $R_{umzubenennen}$ berechnet werden, die durch temporäre Register ersetzt werden müssen:

$$R_{umzubenennen} = R_{aktiv} \cap R_{geschrieben}$$

Als temporäre Register stehen alle Register $R_{verfuegbar}$ zur Verfügung. Sie sind weder aktiv noch werden sie von den zu verschiebenden Instruktionen M definiert:

$$R_{verfuegbar} = \{0, 1, ..., 31\} \setminus \left(R_{aktiv} \cup R_{geschrieben}\right)$$

Ist die Anzahl an Registern in $R_{verfuegbar}$ kleiner als die Anzahl an benötigten Registern $R_{umzubenennen}$, so kann die zuletzt überprüfte Instruktion nicht mehr verschoben werden.

3. Auswahl der zu verschiebenden Instruktionen

Es ist nicht immer sinnvoll, die maximal mögliche Anzahl an Instruktionen zu verschieben. In einer ungünstigen Situation kann (a) der kritische Pfad des Quellblocks oder der Zielblöcke länger werden, was zu Verzögerungen führen kann, oder es können (b) mehr Zeilen aufgrund von Ressourcenbeschränkungen benötigt werden (z. B. viele Speicherzugriffe). Dadurch würden insgesamt längere Konfigurationen erzeugt, was den Zielen der Code-Optimierung widerspricht.

Um das zu verhindern wird die Anzahl an Zeilen, die der Quellblock und jeder Zielblock im Array benötigen, abgeschätzt. Für zusätzliche Kopieroperationen wird eine extra Zeile im Quellblock einkalkuliert. Jede Zeilenanzahl wird mit der Ausführungshäufigkeit des betreffenden Blocks multipliziert und es wird die Summe dieser Werte für alle Quell- und Ziel-Blöcke berechnet. Von dieser Summe wird dann der Wert für die Blöcke ohne Modifikation abgezogen: Es

ergibt sich eine Maßzahl, anhand derer entschieden wird, ob die statische Spekulation ausgeführt werden soll. Dazu muss die Maßzahl positiv sein.

Um die beste Menge an zu verschiebenden Instruktionen M_{opt} zu bestimmen, wird zuerst überprüft, ob M_{max} mindestens eine Instruktion enthält. Beginnend mit dieser Instruktion wird immer eine weitere Instruktion zu M_{opt} hinzugefügt, bis ein optimaler Wert für die Zielfunktion erreicht ist.

Ist M_{opt} leer, es soll also keine Instruktion verschoben werden, so wird mit dem nächsten Block aus der Kandidatenliste und Schritt 2 fortgefahren. Es werden für maximal p_b Quellblöcke tatsächlich Verschiebungen durchgeführt.

4. Verschieben der Instruktionen

Die Instruktionen M_{opt} werden in die Zielblöcke an die Einfügepunkte kopiert und aus dem Quellblock gelöscht. Werden temporäre Register verwendet, so müssen die kopierten Instruktionen entsprechend verändert und Kopierbefehle in den Quellblock eingefügt werden.

5. Aktualisierung der Programmdatei

Wie in Kapitel 3.6 beschrieben müssen die Adressen der Sprungbefehle im Programm aktualisiert werden, da zusätzliche Befehle eingefügt wurden.

Das so veränderte Programm verwendet keine besonderen Instruktionen oder Hardware-Erweiterungen, kann also ohne Änderungen an der Hardware ausgeführt werden. Somit ist es weiterhin auf dem GAP, aber auch auf dem SimpleScalar/PISA-Simulator ausführbar.

4.4.4. Evaluierung

Im Rahmen der Evaluierung wird untersucht, inwieweit die statische Spekulation die Performance von Programmen beeinflusst und was die Ursachen dafür sind.

Performance-Auswertung für den GAP

Benchmarks der MiBench-Suite werden erst mit GCC und aktivierten Optimierungen übersetzt, anschließend wird mit GAPtimize die statische Spekulation angewendet. Die Benchmarks werden auf verschiedenen Hardware-Konfigurationen des GAP ausgeführt, der mit einem taktgenauen Simulator simuliert wird. Es wird immer ein 2-Bit träger Automat (*bimodal branch predictor*) für die Sprungvorhersage verwendet.

Abbildung 4.25 zeigt die maximale Beschleunigung, die für 10 Benchmarks mit dem GAP mit einer, acht oder 32 Konfigurationsebenen erreicht werden kann. Bis auf drei Benchmarks sind die erreichten Werte mit 2% bis 3% Beschleunigung eher mäßig, für die Benchmarks `tele-crc32`, `secu-rijndael-decode-nounroll` und `secu-rijndael-encode-nounroll` werden aber sehr hohe Werte mit einem Maximum von 51,83% für `secu-rijndael-encode-nounroll` bei Ausführung auf dem GAP mit 32 Konfigurationsebenen erreicht. Ursächlich dafür ist, dass bei diesen drei Benchmarks stark dominierende Schleifen vorhanden sind, deren Ausführung beschleunigt werden kann. Dabei arbeiten GAP und die statische Spekulation Hand in Hand:

Durch die statische Spekulation wird die Anzahl der Zeilen, die für den Schleifenkörper benötigt wird, reduziert. Passt die Schleife dann komplett in das Array (bzw. alle Konfigurationsebenen), so kann sie sehr schnell ausgeführt werden (vgl. Kapitel 2.1.2). Für den Benchmark `tele-crc32` können bei einem 12x12x1-Array 428 Schleifeniterationen beschleunigt werden, mit statischer Spekulation sind es 109310 Iterationen. Sind mehr Konfigurationseinheiten vorhanden, so nimmt der Einfluss der statischen Spekulation ab.

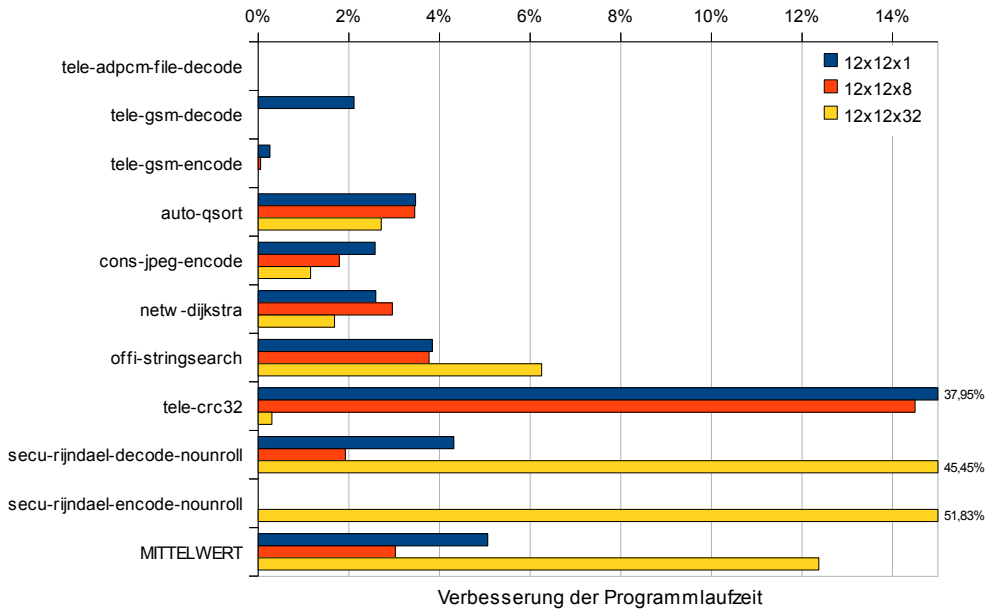
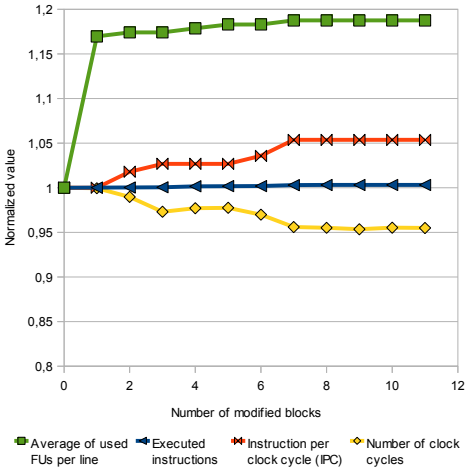


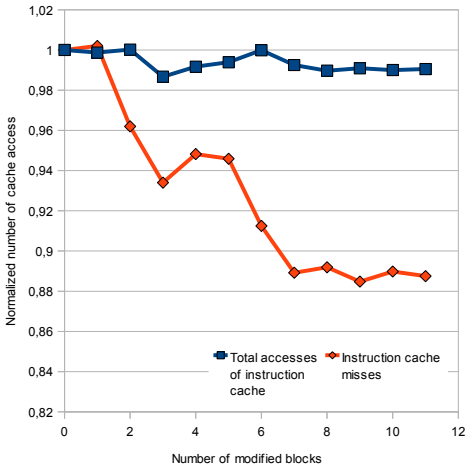
Abbildung 4.25.: Maximale Beschleunigung verschiedener Benchmarks bei Ausführung mit der statischen Spekulation auf dem GAP mit 12x12xN-Array

Eine genaue Analyse der Performance-Daten für `cons-jpeg-encode` ausgeführt auf GAP mit 12x12x1-Array zeigt einige Veränderungen. Durch die statische Spekulation steigt die durchschnittliche Anzahl an verwendeten Funktionseinheiten. Der Grad an Parallelität, mit dem Befehle ausgeführt werden können, wächst geringfügig. Außerdem reduziert sich die Anzahl an Zugriffen auf den Befehlszwischenspeicher, die Anzahl an erfolglosen Zugriffen bleibt aber gleich. Davon unabhängig bleibt die Anzahl ausgeführter Instruktionen beinahe konstant, es wird also (wahrscheinlich) keine Energie durch die statische Spekulation verschwendet. Diese Daten zeigen die Abbildungen 4.26(a), (b), (c) und (d).

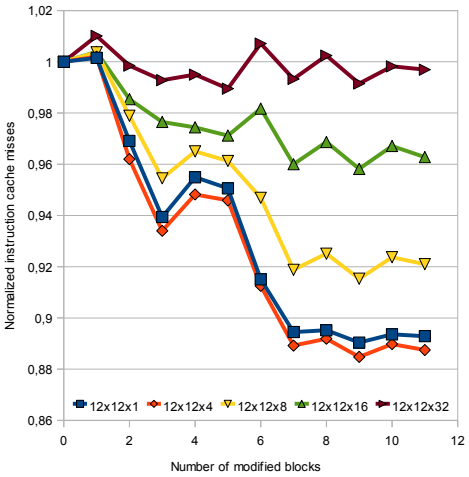
Es lässt sich mit der statischen Spekulation ein hoher Performance-Gewinn erreichen, er ist aber auf bestimmte Benchmarks beschränkt und gilt nicht für alle Konfigurationen des GAP.



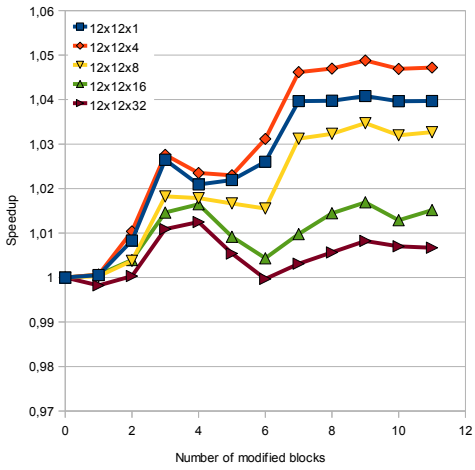
(a) Performance-Indikatoren für `jpeg-encode` bei Ausführung auf dem GAP mit einem 12x12x1-Array



(b) Gesamtanzahl und fehlerhafte Zugriffe auf den Befehlszwischenspeicher bei Ausführung von `jpeg-encode` auf dem GAP mit 12x12x1-Array



(c) Fehlerhafte Zugriffe auf den Befehlszwischenspeicher für `jpeg-encode` für unterschiedliche Konfigurationen des GAP



(d) Speedup als Verhältnis zwischen Gesamttakten vor und nach der Anwendung der statischen Spekulation für `jpeg-encode` und verschiedene GAP-Konfigurationen

Abbildung 4.26.: Performance-Indikatoren für `jpeg-encode`

Parameter für die Heuristik

Anders als bei den vorangehenden Optimierungen ist es für die statische Spekulation ungleich komplizierter, vor Berechnung von Ergebnissen nach optimalen Werten für die Parameter der Heuristik (vgl. Tabelle 4.3) zu suchen. Grund dafür ist, dass die hohe Beschleunigung, die drei der Benchmarks erreichen, dafür sorgt, dass potentielle Verbesserungen für die übrigen sieben Benchmarks, die in der Evaluation verwendet wurden, sehr in den Hintergrund gedrängt werden. Deshalb sind damit ermittelte Werte oft nur für drei Benchmarks optimal, nicht aber für deren Gesamtheit.

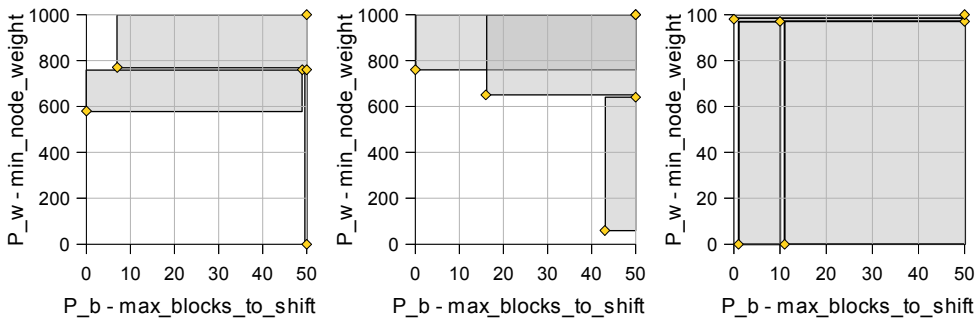
Für Abschnitt 4.4.4 wurden pro Benchmark und pro Hardware-Konfiguration gute Parameter gesucht. Aus diesem Grund sind diese Ergebnisse teilweise etwas besser als in Artikel [79], in dem die statische Spekulation ursprünglich vorgestellt wurde.

Analysiert man die dabei ermittelten Parameter-Konfigurationen, so zeigt sich, dass gute Parameter nicht aus einem einzelnen Bereich stammen, sondern in mehreren Intervallen zu finden sind. Mithilfe einer Technik des *Data Mining*, einem sogenannten *Entscheidungsbaum*, kann man ein Regelwerk erstellen, mit dem sehr gute Parameterkonfigurationen relativ gut beschrieben werden können[6]. Folgende Tabelle zeigt die verwendete Array-Größe, wie viele Konfigurationen über die Regel klassifiziert werden können und wie viele Klassifikationen falsch sind sowie die Regel selbst:

12x12x1	836 / 101	$760 < p_w$	$\wedge$	$p_p \leq 93$	$\wedge$	$7 < p_b$
12x12x1	257 / 66	$p_w \leq 760$	$\wedge$	$p_p \leq 99$	$\wedge$	$49 < p_b$
12x12x1	206 / 76	$570 < p_w \leq 760$	$\wedge$	$p_p \leq 99$	$\wedge$	$p_b \leq 49$
12x12x8	542 / 132	$640 < p_w$	$\wedge$	$p_p \leq 62$	$\wedge$	$15 < p_b$
12x12x8	277 / 104	$50 < p_w \leq 640$	$\wedge$	$p_p \leq 98$	$\wedge$	$42 < p_b$
12x12x8	264 / 84	$750 < p_w$	$\wedge$	$74 < p_p \leq 98$	$\wedge$	
12x12x32	1427 / 354	$p_w \leq 970$	$\wedge$	$6 < p_p \leq 99$	$\wedge$	$10 < p_b$
12x12x32	858 / 123	$970 < p_w$	$\wedge$	$p_p \leq 99$		
12x12x32	238 / 103	$p_w \leq 970$	$\wedge$	$p_p \leq 99$	$\wedge$	$0 < p_b \leq 10$

Diese neun Regeln repräsentieren für die drei Array-Konfigurationen 12x12x1, 12x12x8 und 12x12x32 jeweils einen Großteil der Parameter-Einstellungen, mit denen die statische Spekulation zu guter Performance führt. Die Interpretation der Regeln ergibt, dass p_p beinahe beliebig gewählt werden kann, allerdings nie den Wert 100 besitzen soll, was dazu führen würde, dass nur ohne Spekulation Instruktionen verschoben werden würden. Die Analyse der Parameter p_w und p_b ergibt, dass bei beiden Parametern für den kompletten Wertebereich mindestens eine Regel vorhanden ist. Man kann also einen der beiden Parameter beliebig wählen und wird dann immer eine Regel finden, mit der man den anderen Parameter so definieren kann, dass man sehr gute Ergebnisse erreichen sollte. Das zeigen die Abbildungen 4.27(a), (b) und (c); es sind diejenigen Bereiche im Parameterraum grafisch dargestellt, in denen gute Konfigurationen liegen.

[6] Das F-Maß gibt Auskunft über die Qualität des Klassifikators; es hat für 12x12x1 den Wert 0,729, für 12x12x8 den Wert 0,624 und für 12x12x32 den Wert 0,728 (Minimum 0,0 – Maximum 1,0).



(a) 12x12x1 (b) 12x12x8 (c) 12x12x32

Abbildung 4.27.: Durch Regeln abgedeckte Kombinationen aus P_w und P_b für unterschiedliche Array-Konfigurationen

Überraschend ist, dass es bei 32 Konfigurationsebenen scheinbar egal ist, wie P_w und P_b gewählt werden. Hier ist beinahe der gesamte Raum abgedeckt. Wählt man die Parameter aber so, wie es für 12x12x1 und 12x12x8 vorgeschlagen ist, so ist man quasi auf der sicheren Seite. Eine Hardware-Abhängigkeit der Parameter ist also gegeben, da Konfigurationen, die bei 12x12x32 gute Ergebnisse zeigen, nicht unbedingt auch bei 12x12x1 oder 12x12x8 zu guten Ergebnissen führen.

Führt man die mit GAPtimize veränderten Benchmarks, wobei pro Benchmark optimale Parameter verwendet werden, auf dem SimpleScalar-Simulator aus, so erhält man im Vergleich zu den unveränderten Programmversionen die in Abbildung 4.28 dargestellten Beschleunigungen. Alles in allem scheint die statische Spekulation für den SimpleScalar nur in Ausnahmefällen (`tele-crc32`) eine sinnvolle Code-Optimierung zu sein. Das ändert sich auch nicht, wenn statt *out-of-order*-Zuordnung (wie im Bild zu sehen) nur *in-order*-Zuordnung verwendet wird.

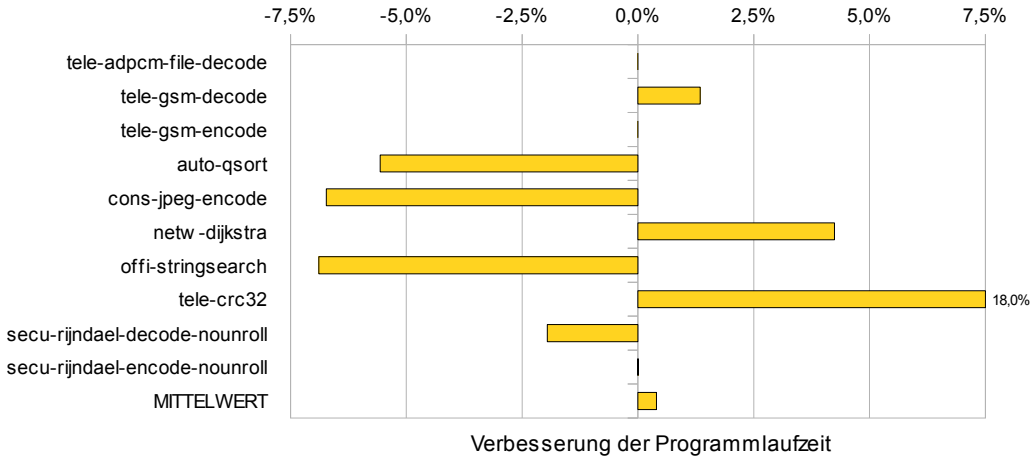


Abbildung 4.28.: Veränderung der Programmausführungszeiten bei Verwendung der statischen Spekulation auf dem SimpleScalar-Simulator

4.4.5. Verwandte Arbeiten

Ähnliche Problemstellungen sind bei der Code-Übersetzung für superskalare und VLIW-Architekturen vorhanden.

Mit *Trace Scheduling* [56] kann für VLIW-Architekturen Parallelität auch dann offengelegt werden, wenn dazu die Grenzen von Blöcken überschritten werden. Die Technik unterteilt ein Programm in Abfolgen von Blöcken, die in dieser Reihenfolge am häufigsten ausgeführt werden, und ordnet die darin befindlichen Instruktionen an, als ob es sich um die Instruktionen eines einzelnen Blockes handeln würde. Zusätzlicher Code ist nötig, um mit unterbrechenden Kontrollflusskanten bei Abfolgen zurechtkommen zu können. In diesem Fall müssen Blöcke kopiert und Kompensationsinstruktionen eingefügt werden.

Trace Scheduling ist z. B. im *Multiflow Trace Scheduling Compiler* [100] umgesetzt. Dieser Compiler versucht, Instruktionen über Verzweigungen im Kontrollfluss zu verschieben, führt das aber nur durch, wenn dazu kein zusätzlicher Kompensationscode nötig ist.

Bernstein et al. [12] schlagen vor, Instruktionen in vorangehende Blöcke zu verschieben. Die Autoren erklären, dass Instruktionen spekulativ verschoben werden könnten, belassen es aber dabei ohne weitere Details zu nennen. Der Fokus liegt klar darauf, einzelne Instruktionen zu verschieben. Tirumalai et al. [155] demonstrieren ein ähnliches Vorgehen. Der Hauptunterschied zur statischen Spekulation liegt darin, dass sie in einem Schritt mehr als eine Instruktion verschieben kann. Zwar könnten die benannten Ansätze öfters ausgeführt werden; dazu müssten aber viele Analysen wesentlich öfters durchlaufen werden, was viel Zeit kosten würde. Zudem werden bei der statischen Spekulation explizit nicht nur Instruktionen verschoben, die mit absoluter Sicherheit ausgeführt werden.

Bei der *Tail Duplication* wird ein nachfolgender Block in alle vorangehenden Blöcke kopiert und anschließend gelöscht. Dadurch wird versucht, die Parallelität unter Befehlen durch Entfernung von Kontrollflussabhängigkeiten zu erhöhen. Tail Duplication wird im Artikel [60] mit ähnlichen Techniken verglichen. Im Vergleich zur statischen Spekulation können mit Tail Duplication nur sicher auszuführende Instruktionen verschoben werden.

Wie bereits erwähnt, zeigt die statische Spekulation Parallelen mit Software Pipelining (siehe z. B. [97]), da Schleifen aufgeteilt und neu verteilt werden können. Allerdings sind die meisten Algorithmen für Software Pipelining wesentlich komplexer, sie können besser optimieren und unterstützen eine höhere Anzahl an Schleifeniterationen, die gleichzeitig ausgeführt werden können. Jedoch kann die statische Spekulation nicht nur Schleifen optimieren.

Mit *Scoreboarding* [153, 152] und *Tomasulo's scheme* [156] in Verbindung mit einer Sprungvorhersagetechnik sind Prozessorbautechniken mit ähnlichem Zweck wie die statische Spekulation vorhanden. Allerdings setzt die damit mögliche *out-of-order*-Ausführung Strukturen voraus, die einen hohen Hardwareaufwand bedingen und neue Einschränkungen mit sich bringen können [44]. Beim Entwurf des GAP wurde zugunsten der asynchronen Ausführung und besserer Vorhersagbarkeit des Ausführungszeitpunktes bewusst gegen diese Ansätze entschieden.

Zusammenfassend sind also die heraustretenden Merkmale der statischen Spekulation, dass (a) mehrere Instruktionen in einem Durchlauf verschoben werden können, dass (b) Instruktionen auch verschoben werden können, wenn sie nicht mit hundertprozentiger Sicherheit ausgeführt werden sollen, in diesem Fall wird Kompensationscode eingefügt, und dass (c) alle Konstellationen von Blöcken unabhängig von Dominanz- oder Postdominanz-Beziehungen behandelt werden können.

4.4.6. Übertragung auf andere Architekturen

Den GAP als Zielarchitektur zeichnet aus, dass er (abhängig von den gewählten Dimensionen des Arrays) viele nicht verwendete Funktionseinheiten besitzt, die ohne damit einhergehende Verzögerungen verwendet werden können. Die statische Spekulation ist auch für andere Prozessoren, die über dieselben oder ähnliche Eigenschaften verfügen, interessant.

Des Weiteren kann die statische Spekulation für Architekturen interessant sein, bei denen es sonst keine Verfahren zur spekulativen Ausführung von Befehlen gibt, also v. a. Architekturen mit in-order-Ausführung und hoher Frontend-Breite und einer nicht zu geringen Anzahl an Registern. In Kombination mit *Transactional Memory* [70] könnten Speicherbefehle ebenso ohne Probleme verschoben werden, sodass sich dann wesentlich mehr Anwendungsmöglichkeiten ergeben würden.

4.4.7. Zusammenfassung

Es wurde ein Algorithmus für den Post-Link-Optimizer GAPtimize vorgestellt, mit dem bei Ausführung auf dem GAP die Performance so modifizierter Programme erhöht werden kann. Dies gelingt durch Verwendung andernfalls unbenutzter Funktionseinheiten. Mit ihnen werden spekulativ Instruktionen ausgeführt, die nur mit hoher Wahrscheinlichkeit tatsächlich benötigt werden. Dadurch entsteht im besten Fall kein zusätzlicher Zeitaufwand, dafür stehen später benötigte Ergebnisse bereits ohne Verzögerung zur Verfügung. Die höchste gemessene Beschleunigung ist 51,83%, der Mittelwert liegt je nach Konfiguration bei ca. 3% bis 12,5%.

Alles in allem prägen die Ergebnisse aber den Eindruck, als ob die statische Spekulation nicht unter allen Umständen ein geeignetes Mittel ist, um die Ausführung von Programmen zu beschleunigen. Die erzielte Beschleunigung ist stark von der Konfiguration des GAP und dem verwendeten Programm abhängig. Deshalb wäre die statische Spekulation ein sehr guter Kandidat für eine adaptive Optimierung, die je nach Situation und Performance des Programms komplett deaktiviert werden könnte.

Als Erweiterung des Algorithmus wäre denkbar, das Verfahren bereits in das *Scheduling* von Befehlen zu integrieren. In diesem Schritt wird, noch bevor Register zugewiesen werden, die Reihenfolge bestimmt, in der Befehle zur Ausführung in den Prozessor geladen werden. Für ein spekulatives Scheduling könnten schon in dieser Phase Befehle so umkopiert und verschoben werden. Da dann Register noch nicht vergeben sind erhöht sich zwar der Bedarf an Registern (*Register pressure*), andererseits können aber viel mehr Befehle verschoben werden, da z. B. schreibende Speicherbefehle keine Barriere mehr darstellen.

4.5. Verbesserte Ersetzungsstrategie für die Konfigurationsebenen

Dieser Abschnitt stellt ein Verfahren vor, mit dem die Konfigurationsebenen des GAP besser verwendet werden können. Dadurch sollen weniger Probleme beim Zugriff auf den Befehlszwischenspeicher entstehen, da schlichtweg seltener auf ihn zugegriffen wird, wodurch die Performance des GAP steigen soll.

4.5.1. Einführung und Ziele

Der GAP verfügt wie in Kapitel 2.1.1 beschrieben über mehrere Konfigurationsebenen. Sie werden verwendet, um Konfigurationen, also Programmteile, direkt in den Funktionseinheiten des Prozessors zu speichern, sodass sie für eine erneute Ausführung mit sehr geringer Verzögerung bereitstehen. Sind alle Ebenen konfiguriert, so muss entschieden werden, welche gelöscht werden kann.

Da auf die Inhalte der Konfigurationsebenen sehr schnell zugegriffen werden kann ist es erstrebenswert, sie möglichst gut zu nutzen. Die verwendete Ersetzungsstrategie sollte also immer diejenigen Konfigurationen bereithalten, die bald wiederverwendet werden. Es bieten sich Ersetzungsstrategien an, die für die Ersetzung von Speicherseiten (*page replacement*) verwendet werden, z. B. *First in first out* (FIFO), *Least recently used* (LRU) oder eine GAP-spezifische Strategie ähnlich zu FIFO. In Experimenten mit dem GAP hat sich gezeigt, dass nicht immer die gewünschte Leistung erreicht wird; sogar eine Strategie, die zufällig Konfigurationen ersetzt (im Folgenden: RANDOM), funktioniert oft besser.

Um das genauer untersuchen zu können werden Methoden dazu vorgestellt; bekannte Ersetzungsstrategien werden damit in Abschnitt 4.5.2 beleuchtet. Die neue Ersetzungsstrategie QdLRU, die Software-unterstützt arbeitet, soll die Schwächen der bekannten Strategien nicht aufweisen (Abschnitt 4.5.3). Evaluiert wird sie in Abschnitt 4.5.4. Abschnitt 4.5.5 stellt verwandte Ansätze vor und die Ergebnisse werden in Abschnitt 4.5.7 zusammengefasst.

4.5.2. Grundlagen und Untersuchung bekannter Ersetzungsstrategien

Maßzahlen zur Bewertung einer Ersetzungsstrategie

Um zu messen, wie erfolgreich eine Ersetzungsstrategie arbeitet, bieten sich zwei Verfahren an:

- Wird nur die **Komponente des Gesamtsystems** betrachtet, in der die Strategie Verwendung findet, so kann die **Trefferquote** berechnet werden. Sie ist für die Konfigurationsebenen das Verhältnis aus erfolgreichen Zugriffen a_{hit} und der Anzahl aller Zugriffe a_{total} auf die Konfigurationsebenen: $h_{total} = \frac{a_{hit}}{a_{total}}$

  Ein Zugriff ist dann erfolgreich, wenn die gewünschte Instruktion als erste Instruktion auf einer Konfigurationsebene gefunden werden kann. Erfolgreiche Zugriffe können zweierlei Ursachen haben:

  - Entweder es wird auf dieselbe Ebene zugegriffen, die momentan ausgeführt wird, dann handelt es sich um eine Schleife. Diese Treffer a_{loop} sind dann eigentlich der Schleifenbeschleunigung (siehe Kapitel 2.1.2) zuzurechnen.
  - Ansonsten handelt es sich tatsächlich um einen Treffer, der der Existenz der Konfigurationsebenen zuzurechnen ist, er wird zu a_{layer} gezählt.

  Es lassen sich entsprechende Trefferquoten für die Schleifenbeschleunigung h_{loop} und die Konfigurationsebenen h_{layer} berechnen, sie stehen wie folgt mit der gesamten Trefferquote h_{total} in Zusammenhang:

$$h_{total} = \frac{a_{hit}}{a_{total}} = \frac{a_{loop}}{a_{total}} + \frac{a_{layer}}{a_{total}} = h_{loop} + h_{layer}$$

  Jede Ersetzungsstrategie kann nur die Trefferquote für die Konfigurationsebenen h_{layer}beeinflussen. Bei gleichen Array-Abmessungen und demselben Benchmark ist h_{loop} unabhängig von der Anzahl der Konfigurationseben immer gleich. Die Trefferquote h_{total} kann entweder durch Auswerten eines Traces ermittelt oder direkt im GAP gemessen werden. Die Berechnung mit einem Trace ist hinreichend genau, es werden sehr ähnliche Ergebnisse wie bei der Ausführung im GAP erreicht.

- Betrachtet man das **Gesamtsystem**, also hier den GAP an sich, so kann seine Performance ganz normal mit dem **IPC** (siehe Kapitel 2.3.1) gemessen werden. Darin fließt das Verhalten der Ersetzungsstrategie mit ein.

Bekannte Strategien und ihr Verhalten

Übertragbar auf die Ersetzung im GAP sind Verfahren, die ebenso für die Ersetzung von Speicherseiten (*page replacement*) verwendet werden. Die bestmögliche Trefferquote kann mit dem Verfahren von Belady [10], es wird nachfolgend OPT genannt, erzielt werden. Allerdings handelt es sich dabei um eine *Offline-Strategie*, d. h. sie kann erst angewendet werden, wenn die komplette Zugriffsfolge verfügbar ist. Somit ist OPT nur theoretisch einsetzbar. Ersetzt wird immer das Element, dessen nächste Verwendung am weitesten in der Zukunft liegt.

In Abbildung 4.29 ist für 15 Benchmarks die durchschnittliche gesamte Trefferquote h_{total} für die Strategien FIFO, LRU, RANDOM und OPT zu sehen. Basis ist der Trace eines GAP mit einem Array mit 12 Spalten und Zeilen und einer veränderbaren Anzahl an Konfigurationseinheiten, also 12x12xN.

Für eine einzelne Konfigurationsebene entspricht die gesamte Trefferquote der Trefferquote für die Schleifenbeschleunigung, also $h_{total} = h_{loop}$, d. h. $h_{layer} = 0$ und somit ist sie gleich für alle Strategien. Mit steigender Anzahl an Konfigurationsebenen steigt auch die Trefferquote der Ersetzungsstrategien. Auffällig ist, dass RANDOM in allen Fällen bessere Trefferquoten als LRU und FIFO erreicht, obwohl sie ausschließlich kontextunabhängig und zufällig entscheidet, im Vergleich zu LRU also relativ stupide handelt.

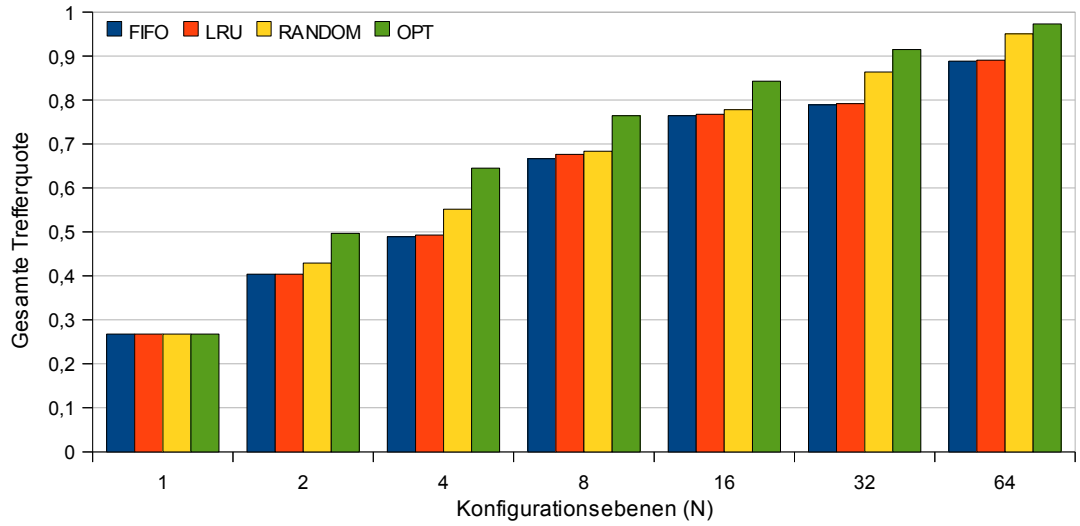


Abbildung 4.29.: Durchschnittliche gesamte Trefferquote für 15 Benchmarks für den GAP mit einem 12x12-Array und unterschiedlicher Anzahl an Konfigurationsebenen

Details zu den gesamten Trefferquoten für den GAP mit einem 12x12x16-Array für 15 Benchmarks zeigt Abbildung 4.30. Für einige Benchmarks zeigt LRU fast keine Fehlzugriffe, also $h_{total} \approx 1$. Für andere Benchmarks, z. B. `auto-qsort`, `netw-dijkstra`, `offi-stringsearch`, `secu-rijndael-encode`, gibt es eine signifikante Lücke zwischen LRU und OPT; sie werden deshalb im Folgenden näher betrachtet. Diese Lücke Δ zwischen LRU und OPT, siehe Tabelle 4.5.2, stellt das Verbesserungspotential für eine neue Ersetzungsstrategie dar.

Konfigurationsebenen	$\Delta :=$ OPT $-$ LRU	Δ/LRU
1	0	0%
2	0.09	23%
4	0.15	31%
8	0.09	13%
16	0.08	10%
32	0.12	16%
64	0.08	9%
Mittelwert	0.09	15%

Tabelle 4.4.: Unterschied zwischen der Trefferquote für LRU und OPT

Thrashing als Hauptkritikpunkt an LRU

Das Verhalten der Ersetzungsstrategien kann am besten anhand einer grafischen Darstellung beobachtet werden, sie wird im folgenden *Zugriffs-Plot* genannt. Dazu wer-

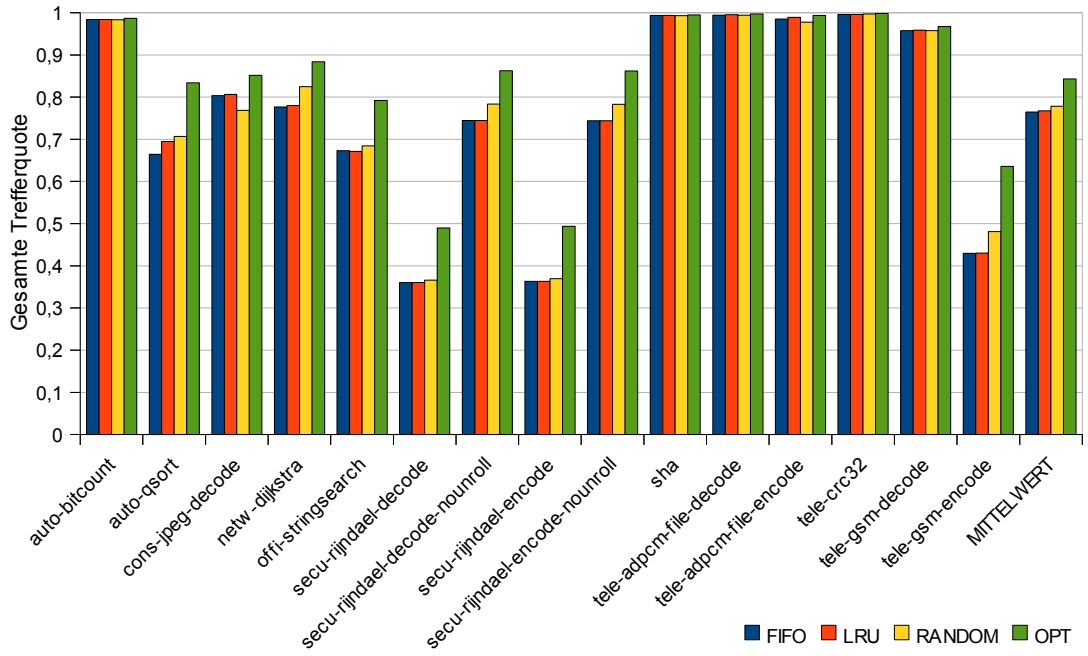


Abbildung 4.30.: Gesamte Trefferquote für den GAP mit 12x12x16 Array

den bei der Ausführung des Programms im GAP die Zugriffe auf die Konfigurationen aufgezeichnet und dann in einer Bitmap-Grafik dargestellt. Jeder Zugriff wird durch einen Punkt repräsentiert. Die Koordinaten (X, Y) des Punktes werden wie folgt bestimmt:

X-Koordinate: Alle Zugriffe werden fortlaufend nummeriert, diese Zahlen werden als X-Koordinate verwendet. Somit ist pro „Spalte“ der Grafik genau ein Zugriff zu sehen.

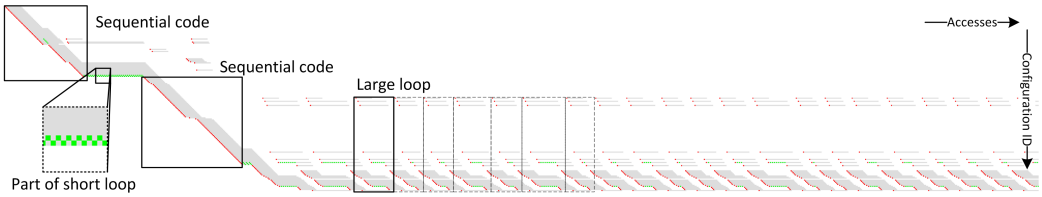
Erfolgen nacheinander mehrere Zugriffe auf dieselbe Konfiguration (Schleife), so wird keine neue Nummer vergeben sondern die letzte beibehalten und der Punkt speziell eingefärbt.

Y-Koordinate: Die Konfigurationen werden so, wie auf sie zugegriffen wird, fortlaufend indiziert. Wird auf eine Konfiguration erneut zugegriffen, so wird der bereits vergebene Index verwendet.

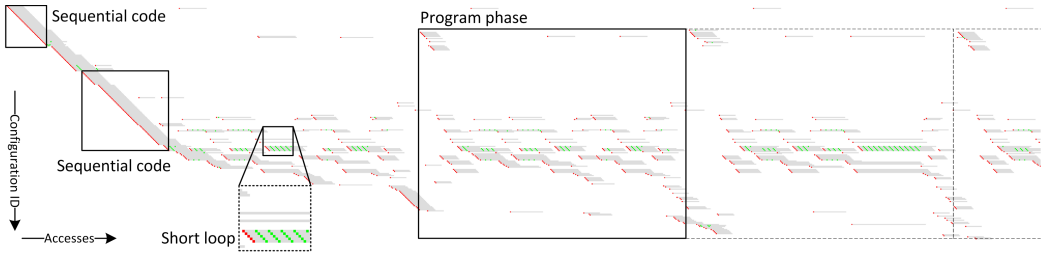
Bei einem erfolgreichen Zugriff wird der Punkt grün eingefärbt; bei einem Fehlzugriff, d. h. die gewünschte Konfiguration ist nicht in einer Ebene vorhanden, wird der Punkt rot gezeichnet. Ist die Schleifenbeschleunigung aktiv – in diesem Fall hat die Ersetzungsstrategie keinen Einfluss – so wird der Punkt blau eingefärbt. Zusätzlich werden bei jedem Zugriff alle anderen Konfigurationen, die in den Ebenen verfügbar sind, durch hellgraue Punkte angedeutet. Wird eine Konfiguration verdrängt, so endet ihre graue horizontale Linie. Sie hat mit einem roten Punkt begonnen und auf ihr lie-

gen evtl. weitere grüne Punkte, die symbolisieren, dass die Konfiguration erfolgreich wiederverwendet werden konnte.

Die Abbildungen 4.31(a) und (b) zeigen jeweils die ersten ca. 1000 Zugriffe auf Konfigurationsebenen für die Benchmarks `offi-stringsearch` und `auto-qsort`. Neben der Tatsache, dass die Anzahl der Konfigurationseinheiten, auf die zugegriffen wird, relativ klein ist im Vergleich zur Anzahl an Zugriffen, lassen sich einige Zugriffsmuster erkennen, die sich stark durch ihre Größe, Periodizität und Lokalität [49] unterscheiden; Aktikel [78] enthält weitere Details.



(a) Zugriffs-Plot für den Benchmark `offi-stringsearch`



(b) Zugriffs-Plot für den Benchmark `auto-qsort`

Abbildung 4.31.: Ca. 1000 Zugriffe auf die Konfigurationen in einem GAP mit 12x12x16-Array dargestellt als Zugriffs-Plot; LRU wird als Ersetzungsstrategie verwendet; einzelne Pattern sind gekennzeichnet (Details siehe [78])

Die wichtigste Erkenntnis aus der Analyse der Zugriffsmuster ist, dass sich die Strategien LRU, RANDOM und OPT für die meisten Pattern sehr ähnlich verhalten, wobei RANDOM fast immer Schwächen zeigt. Einzig bei großen Schleifen mit mehr Elementen als Konfigurationsebenen im GAP, siehe Abbildung 4.32, gibt es starke Unterschiede:

LRU zeigt katastrophales Verhalten; es werden immer die Konfigurationen gelöscht, die bald wieder verwendet werden würden. Somit sinkt die Trefferquote auf 0,0 – man spricht von *Thrashing* [48].

RANDOM zeigt eine höhere Trefferquote, da zumindest ein paar Konfigurationen zufällig lange genug zur erneuten Ausführung in den Ebenen verbleiben.

OPT erzielt die bestmögliche Trefferquote, indem einige Konfigurationen in jeder Iteration der Schleife in den Ebenen verbleiben und somit sicher wiederverwendet werden können.

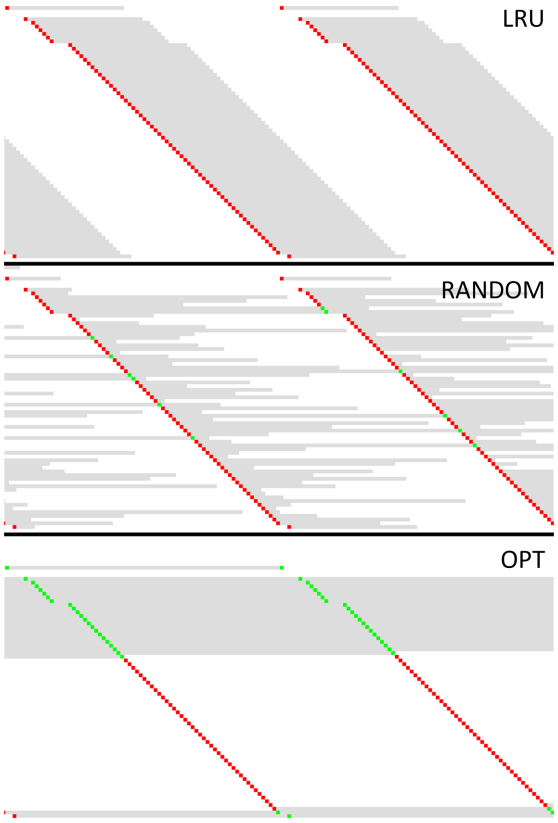


Abbildung 4.32.: Zugriffs-Plot für eine große Schleife mit unterschiedlichen Ersetzungsstrategien (`secu-rijndael` auf GAP mit 12x12x32 Array)

Zusammenfassend verhält sich LRU bei großen Schleifen aufgrund von Thrashing katastrophal. Um dieses Problem zu umgehen kann die Anzahl der Konfigurationsebenen vergrößert werden. Dadurch wird die eigentliche Ursache des Problems aber nicht gelöst, bei komplexeren Programmen tritt es erneut auf. Deshalb wird eine optimierte Ersetzungsstrategie gesucht, die Thrashing verhindert oder zumindest weniger anfällig ist.

4.5.3. Ersetzungsstrategie QdLRU

Mit einer neuen Strategie soll Thrashing deutlich reduziert werden, indem das Verhalten von OPT für große Schleifen imitiert wird. Dazu werden vor Programmausführung diejenigen Situationen bestimmt, bei denen Thrashing auftreten könnte. Als Basis wird LRU verwendet. Aus den beteiligten Konfigurationen werden einige ausgewählt und so markiert, dass sie bei der Ersetzung möglichst schnell wieder aus den Ebenen verdrängt werden. Diese Konfigurationen werden dazu in der internen Zugriffsfolge von LRU an der Position eingefügt, an der das Element stehen würde, dessen Verwendung am längsten zurückliegt. Die neue Strategie wird *Quick drop LRU (QdLRU)* genannt.

Basis ist die Annahme, dass sich *Working Sets* identifizieren lassen. Ein Working Set umfasst diejenigen Konfigurationen, die nacheinander immer wieder ausgeführt werden, bei denen also die Anzahl an Zugriffen auf das Working Set wesentlich höher ist als die Anzahl an Konfigurationen im Working Set. Sei $W := [c_0, ..., c_{47}]$ ein Working Set mit den Konfigurationen c_0 bis c_{47}. Wird es auf einem GAP mit 32 Konfigurationsebenen ausgeführt, so führt das bei LRU zu Thrashing, da die Größe des Working Set größer ist als die Anzahl an Konfigurationsebenen, $|W| = 48 > 32$.

Wird W längere Zeit ausgeführt, so fällt die Trefferquote h_{layer} bei Verwendung von LRU auf 0. Mit RANDOM kann als Trefferquote $h_{layer} \approx 0,22$ erzielt werden. Mit der Strategie OPT werden 31 der 48 Konfigurationen gespeichert und in der übrigen Konfigurationsebene nacheinander alle sonstigen Konfigurationen geladen, somit $h_{layer} = 31/48 = 0,64583$. Mit QdLRU soll dieselbe und optimale Trefferquote erreicht werden können.

Markieren von Instruktionen

Um ein Programm für QdLRU vorzubereiten muss es bereits einmal auf dem Zielsystem ausgeführt worden sein, da Informationen über die Start-Instruktionen der Konfiguration und die Reihenfolge der Zugriffe benötigt werden. Als nächster Schritt werden die sogenannten *Konfigurationslinien* gesucht, sie nähern die Working Sets an. Eine Konfigurationslinie $C = \{c_0, ..., c_i\}$ entspricht im Wesentlichen einer diagonalen Linie im Zugriffs-Plot, z. B. in Abbildung 4.32. Um alle Linien $\mathbb{C} = \{C_0, ..., C_j\}$ zu berechnen wird der Algorithmus aus Quellcode 4.1 auf die Zugriffsfolge der Konfigurationen angewendet.

Eine Konfigurationslinie stellt damit ein Working Set mit dem „kleinsten Grad an Wiederverwendung“ dar. Sie wird mit der Annahme erstellt, dass, sobald auf eine Konfiguration erneut zugegriffen wird, es sich um einen Sprung zum Beginn der Linie oder um den Beginn einer neuen Linie handeln muss. Ergebnis ist eine Menge von Konfigurationslinien, jede bestehend aus mehreren Konfigurationen und versehen mit den Ausführungshäufigkeiten der Linie.

Als nächster Schritt werden aus den Konfigurationen in den Konfigurationslinien einige ausgewählt, die schnell aus den Konfigurationsebenen verdrängt werden sollen. Dazu werden die Konfigurationslinien in zwei Mengen unterteilt: $\mathbb{C}_{short}$ enthält Konfigurationslinien mit weniger Elementen als der Prozessor Konfigurationsebenen hat, $\mathbb{C}_{long}$ für alle anderen, die folglich nicht ohne Verdrängung von Konfigurationen wiederholt ausgeführt werden können. Damit wird folgendes Schema abgearbeitet:

1. Wähle eine Konfigurationslinie `item` aus $\mathbb{C}_{long}$.

2. Wähle aus `item` diejenige Konfiguration, die am seltensten in den kurzen Konfigurationslinien $\mathbb{C}_{short}$ verwendet wird; somit wird sie am seltensten unabsichtlich schnell verdrängt und der negative Einfluss auf die Trefferquote wird reduziert. Markiere die erste Instruktion der Konfiguration (und somit die Konfiguration).

Quellcode 4.1: Algorithmus zur Berechnung von Konfigurationslinien

```
input: list<configuration> trace

#define line list<configuration>
set<line> all_lines
map<line, int> line_counters

line current_line = {}
configuration last_configuration

foreach(configuration item in trace)
    if(item == last_configuration)
        // Do nothing
    else if(item ∉ current_line)
        current_line += item
        last_configuration = item
    else
        all_lines += current_line
        line_counters[current_line]++
        current_line = {}
        last_configuration = item
```

3. Verschiebe alle Konfigurationslinien aus $\mathbb{C}_{long}$ nach $\mathbb{C}_{short}$, deren Anzahl an Konfigurationen abzüglich der Anzahl markierter Konfigurationen kleiner oder gleich groß ist wie die Anzahl an Konfigurationsebenen im Prozessor.

4. Wenn $\mathbb{C}_{long}$ nicht leer ist, fahre bei Schritt 1 fort.

Um Markierungen zu setzen wird GAPtimize (siehe Kapitel 3) verwendet. Es wird bei den markierten Befehlen ein einzelnes Bit gesetzt, das den GAP dann veranlasst, eine Konfiguration, die mit einem markierten Befehl beginnt, schnell zu verdrängen.

Ausführung des modifizierten Programms

Die für QdLRU notwendigen Änderungen an der Hardware des GAP sind sehr einfach, es muss nur die Logik für die Ersetzung von Konfigurationsebenen verändert werden. Anstatt eine neue Konfiguration immer an der *most recently used*-Position (also am Anfang) der internen Zugriffsfolge von LRU einzufügen muss erst überprüft werden, ob die erste Instruktion der Konfiguration markiert ist oder nicht. Ist sie markiert, wird die Konfiguration als *least recently used* eingefügt, ansonsten als *most recently used*. Mit einer Markierung wird die Konfiguration beim nächsten Verdrängungsvorgang sofort entfernt und durch eine andere Konfiguration ersetzt.

Wird ein Programm auf dem GAP ausgeführt, das nicht für QdLRU vorbereitet wurde, so wird ganz normal LRU als Ersetzungsstrategie verwendet, es treten also keinerlei Nachteile auf.

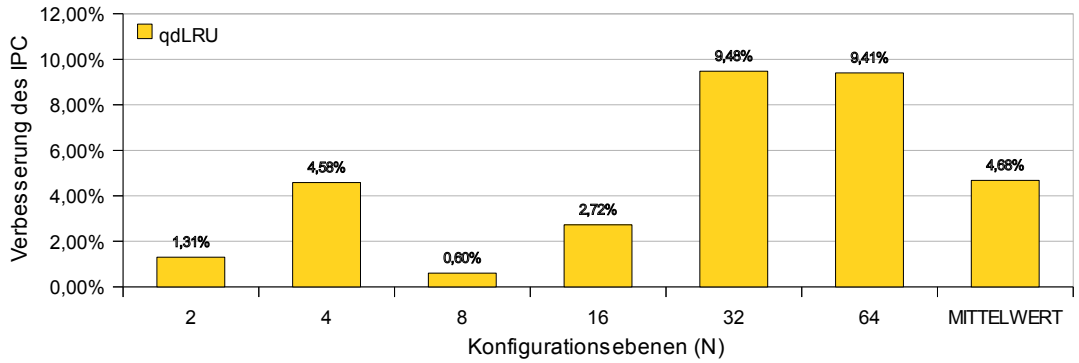


Abbildung 4.33.: Durchschnittliche Verbesserung der Performance des GAP (IPC) für 15 Benchmarks bei Verwendung von QdLRU statt LRU und einem 12x12xN-Array

4.5.4. Bewertung

Für die Bewertung von QdLRU wird der taktgenaue Simulator für den GAP entsprechend erweitert, sodass er LRU und die neue Ersetzungsstrategie QdLRU unterstützt. Mit GAPtimize werden die Benchmarks entsprechend vorbereitet. Als Größe für das Array wird 12x12 gewählt mit unterschiedlich vielen Konfigurationsebenen. Der Performance-Unterschied zwischen der sonst verwendeten Ersetzungsstrategie des GAP und LRU ist mit durchschnittlich 0,27% bei der Gesamtperformance bzw. 1,04% für die gesamte Trefferquote h_{total} für 15 Benchmarks bei einem 12x12x16-Array vernachlässigbar.

Es wird mit QdLRU als Ersatz für LRU für die 15 auch in Abbildung 4.30 gezeigten Benchmarks eine durchschnittliche Verbesserung der System-Performance von 4,68% erzielt. Wie in Abbildung 4.33 zu sehen ist, wird die maximale Verbesserung für 32 bzw. 64 Konfigurationsebenen erzielt (9,48% bzw. 9,41%). Bei Interpretation dieser Werte muss mit berücksichtigt werden, dass für einige der Benchmarks keinerlei Verbesserung erwartet wurde und erreicht wird, da kein Verbesserungspotenzial vorhanden ist.

Die Veränderungen der Trefferquote und des IPC als Maß für die Performance des Gesamtsystems werden für einige Benchmarks, die Thrashing-anfällig sind, in Abbildung 4.34 dargestellt. Eine Performance-Verbesserung von über 300% kann für `secu-rijndael-encode` und `secu-rijndael-decode` erreicht werden, für beide Benchmarks verbessert sich die Trefferquote um 0,5 im Vergleich zu LRU. Beide Benchmarks sind äußerst anfällig für Thrashing, wenn LRU verwendet wird. Eine Verknüpfung zwischen Verbesserung der Trefferquote und der Performance des GAP ist leicht erkennbar.

Abbildung 4.35 zeigt die simulierte durchschnittliche gesamte Trefferquote h_{total} für 15 Benchmarks für die Ersetzungsstrategien LRU, RANDOM, QdLRU und OPT. Mit QdLRU kann für alle Konfigurationen eine höhere Trefferquote erzielt werden als mit LRU und RANDOM. Betrachtet man die Lücke zwischen LRU und OPT, so kann sie

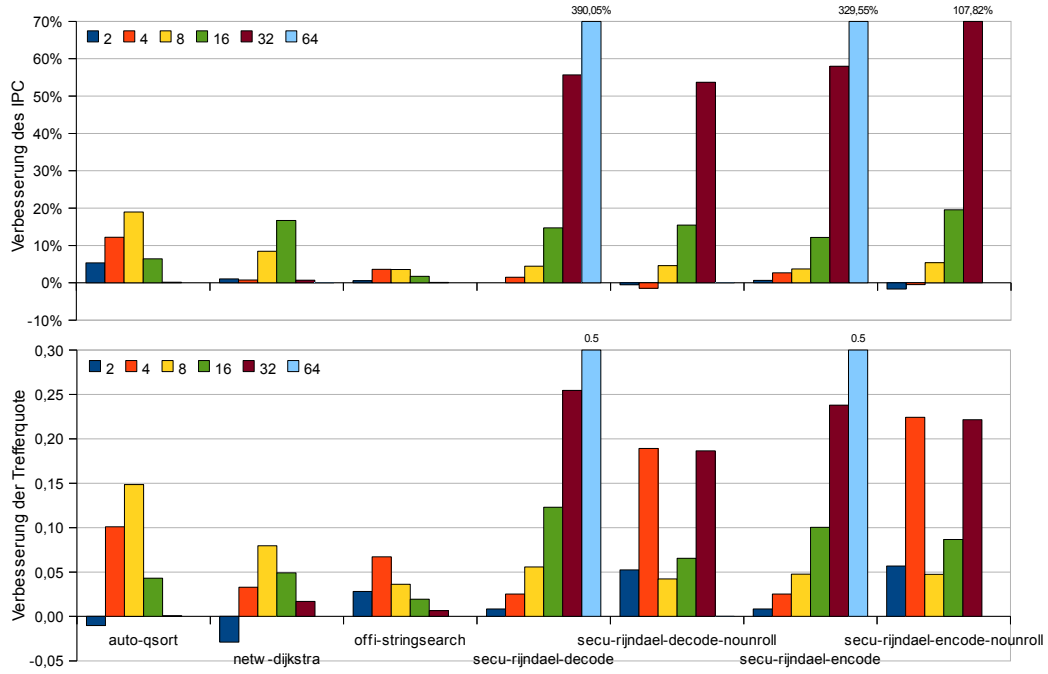


Abbildung 4.34.: Relative Verbesserung durch QdLRU statt LRU des IPC (oben) und Verbesserung der Trefferquote (unten) für ausgewählte Benchmarks mit Thrashing-Risiko bei Ausführung auf dem GAP mit einem 12x12xN-Array

durchschnittlich um 65,97% verkleinert werden. Der kleinste Wert sind 48,38% für 2 Konfigurationsebenen, der größte Wert sind 78,05% für 32 Konfigurationsebenen (siehe Abbildung 4.36).

Bei höherer Speicherlatenz ist davon auszugehen, dass sich die positiven Effekte von QdLRU weiter verstärken, da mit einer höheren Trefferquote für die Konfigurationsebenen eine starke Reduktion der Zugriffe auf den Befehlszwischenspeicher einhergeht. Somit treten wahrscheinlich weniger Fehlzugriffe auf. Würde hingegen die Größe des Befehlszwischenspeichers erhöht, so reduzieren sich die Auswirkungen von QdLRU, da dann weniger Wartezeiten beim Zugriff auf den Befehlszwischenspeicher auftreten würden.

4.5.5. Verwandte Ansätze

Da in den Konfigurationsebenen Programmcode gespeichert wird, arbeiten sie eigentlich wie ein Befehlszwischenspeicher. Darin werden Ersetzungsstrategien verwendet um Platz zu schaffen für neue Programmteile, die geladen werden sollen, da der Befehlszwischenspeicher im Allgemeinen kleiner ist als das Programm, das ausgeführt werden soll. Eine besondere Form davon sind *Trace Caches*, die Programmteile in der Abfolge, in der sie ausgeführt werden, speichern – somit arbeiten sie mit höherer Granularität, ähnlich wie die Konfigurationsebenen. Verwandte Arbeiten lassen sich in

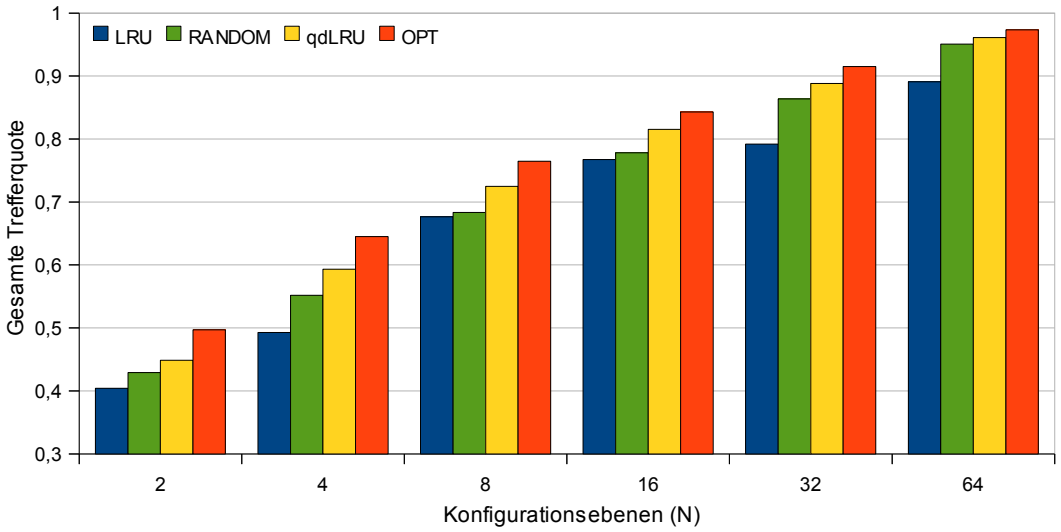


Abbildung 4.35.: Simulierte gesamte Trefferquote für LRU, RANDOM, QdLRU und OPT (Durchschnitt für 15 Benchmarks, GAP mit 12x12xN-Array)

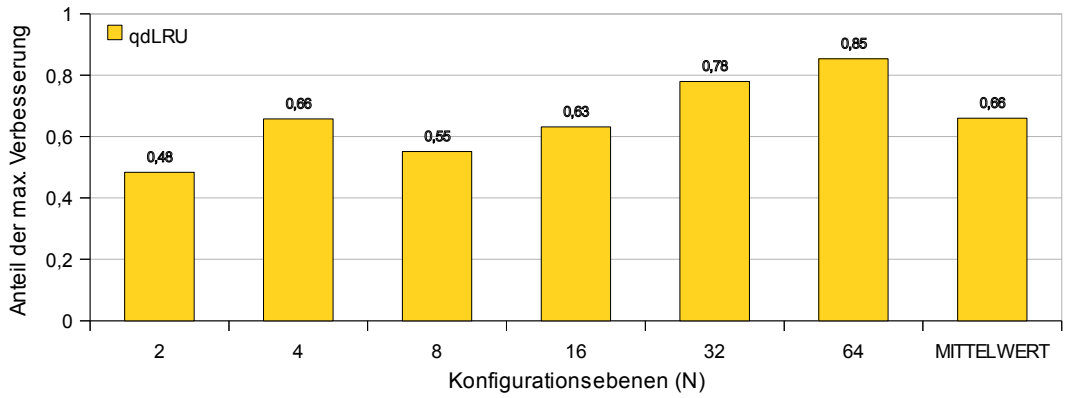


Abbildung 4.36.: Anteil der Lücke zwischen LRU und OPT, der durch QdLRU geschlossen werden kann (Durchschnitt für 15 Benchmarks, GAP mit 12x12xN-Array)

beiden Bereichen finden.

Der Hauptunterschied zwischen Konfigurationsebenen und normalen Befehlszwischenspeichern ist, dass es bei den Ebenen nur eine sehr kleine Anzahl an Positionen gibt, an denen gespeichert werden kann (Beispiel: 64 vs. 4096). Deshalb sind Situationen, bei denen Thrashing auftreten könnte, wesentlich häufiger bei den Konfigurationsebenen als bei normalen Zwischenspeichern.

Anders als für die meisten anderen Prozessoren gibt es für den GAP mit GAPtimize ein mächtiges Werkzeug, mit dem Programme basierend auf Ergebnissen vorhergehender Ausführungen modifiziert werden können. Somit ist ein Ansatz wie QdLRU für den GAP schnell umzusetzen, wohingegen ähnliches für andere Architekturen einen immensen Aufwand bedeuten kann.

Alle Ersetzungsstrategien, die nur in Hardware implementiert sind, lassen sich in zwei unterschiedliche Klassen unterscheiden.

Zur ersten Klasse zählen alle wohlbekannten Algorithmen, die mit geringem und in jedem Fall vertretbarem Aufwand implementiert werden können. Dazu zählen FIFO, RANDOM, WsClock [24] und LRU. Es wird vermutet, dass LRU von diesen Strategien die überlegene ist. Die Performance von LRU wurde zusammen mit FIFO und RANDOM bereits mit der maximal erreichbaren Performance verglichen (Abschnitt 4.5.2) und es wurden gewichtige Schwächen aufgedeckt.

Zur zweiten Klasse zählen wesentlich komplexere Hardware-basierte Algorithmen wie die *Dynamic Insertion Policy (DIP)* von Qureshi et al. [129] und der *Shepherd Cache*, der von Rajan et al. [130] vorgestellt wurde. Beide verlangen wesentlich komplexere Hardware. Bei Experimenten stellte sich außerdem heraus, dass in der spezifischen Situation, die die Konfigurationsebenen des GAP darstellen, mit dem Shepherd Cache nur ähnliche Trefferquoten wie mit LRU erzielt werden können. Bei der DIP sind nur dann gute Ergebnisse zu erreichen, wenn sehr schnell zwischen den beiden Extremen, also LRU und BIP, gewechselt werden kann. BIP wird verwendet, um Thrashing zu verhindern, ist aber relativ nutzlos für alle anderen Situationen. Leider ist es aufgrund der kleinen Anzahl an Konfigurationsebenen nicht möglich, wie vorgeschlagen die Ebenen in zwei Gruppen aufzuteilen. Die geringe Anzahl an vorhandenen Konfigurationsebenen steht der Anwendung von Strategien wie ARC [112] entgegen, die zwei Gruppen erstellen und beide Gruppen unterschiedlich handhaben.

Es gibt eine Vielzahl weiterer Techniken (siehe z. B. [83]), jedoch sind die meisten davon aufgrund der geringen Anzahl an Konfigurationsebenen im GAP nicht verwendbar oder effektiv.

Trace Caches wurden von Rotenberg et al. [135] vorgestellt. Sie erfüllen für superskalare Prozessoren sehr ähnliche Aufgaben wie die Konfigurationsebenen für den GAP. Zum aktuellen Zeitpunkt ist aber kein Artikel bekannt, der sich mit Ersetzungsstrategien oder Thrashing in diesem Kontext beschäftigt.

4.5.6. Übertragung auf andere Architekturen

Die Ersetzungsstrategie ist eigentlich vorrangig für den GAP entworfen worden. Für andere Architekturen ist sie nur von geringem Nutzen, da es meist keine Strukturen

wie die Konfigurationsebenen gibt. Trace Caches bilden eine Ausnahme, sind aber nicht häufig zu finden.

Im Prinzip lässt sich LRU als Ersetzungsstrategie immer dann mit QdLRU ersetzen, wenn es (a) bekannte und wiederkehrende Zugriffsfolgen gibt, die (b) im Voraus oder mit Feedback ermittelt und die (c) mit wenigen Speicherplätzen gepuffert werden können.

4.5.7. Zusammenfassung

Im Vergleich verschiedener Ersetzungsstrategien für die Konfigurationsebenen des GAP, in denen Programmabschnitte ähnlich zu einem Trace Cache gespeichert werden können, hat sich gezeigt, dass die Ersetzungsstrategie LRU wesentlich schlechtere Ergebnisse zeigt als RANDOM. Im Normalfall geht man aber davon aus, dass das LRU die überlegene Strategie ist. Um die Ursachen für die Diskrepanz zu ergründen wurde eine grafische Darstellung für Zugriffsfolgen, das Zugriffs-Plot, vorgestellt. An diesen Grafiken lässt sich erkennen, dass bei größeren Schleifen LRU immer die Elemente aus den Konfigurationsebenen verdrängt, die bald wieder verwendet werden würden. Dieses Verhalten wird Thrashing genannt. RANDOM hat normalerweise niedrigere Trefferquoten, zeigt aber kein Thrashing.

Um Thrashing zu verhindern wird LRU erweitert. Dazu werden einige Konfigurationen mit Markierungen versehen, sie werden dann sehr schnell wieder aus den Konfigurationsebenen verdrängt. Die erweiterte Version von LRU wird QdLRU bezeichnet. Um die Markierungen zu vergeben werden Informationen über die Programmausführung benötigt. Das Programm wird durch GAPtimize dann entsprechend vorbereitet. Die nötigen Hardware-Änderungen sind, wenn LRU bereits unterstützt wird, sehr gering.

Mit QdLRU kann im Vergleich zu LRU bei 15 Benchmarks eine um 4,68% höhere Performance gemessen werden, der Spitzenwert 390% resultiert aus einer um 0,50 gestiegenen Trefferquote.

4.6. Weitere Optimierungen

Weitere Optimierungen und ihr Einfluss auf die Performance des GAP werden in diesem Abschnitt vorgestellt. Diese Optimierungen sind nicht zur automatischen Anwendung in GAPtimize implementiert; sie können teilweise manuell im Programmcode umgesetzt werden. Ausschlaggebend dafür, diese Optimierungen nicht in GAPtimize zu integrieren, war der geringe Erkenntnisgewinn bzw. die geringe Beschleunigung, die damit hätte erzielt werden können.

4.6.1. Abrollen von Schleifen

Programme enthalten fast immer Schleifenstrukturen, die sehr häufig ausgeführt werden. Ist bekannt, wie oft eine Schleife ausgeführt werden soll, so kann sie theoretisch durch die entsprechende Anzahl an Kopien des Schleifenkörpers ersetzt werden. Man spricht von *Loop Unrolling* (oder *Loop Unwinding*, Abrollen von Schleifen, vgl. [42] Kapitel 10.3 und [69] Kapitel 2.2). Im vorgenannten Fall wird die Schleife vollständig abgerollt, was zu einem sehr langen Programm führen kann.

Alternativ kann die Schleife auch nur ein paar Mal abgerollt werden (*Partial Loop Unrolling*), z. B. acht Mal. Dieser Wert wird Abrollfaktor f genannt. Soll die Schleife n mal ausgeführt werden, so müssen, falls n nicht durch den Abrollfaktor f ohne Rest teilbar ist, also $n \bmod f \neq 0$, entweder vor oder nach $n \operatorname{div} f$-maliger Ausführung des abgerollten Codes noch die übrigen $n \bmod f$ Iterationen ausgeführt werden. Quellcode 4.2 zeigt ein Beispiel dafür, es handelt sich um den Kern eines FIR-Filters. Derartige digitale Filter werden häufig in eingebetteten Systemen eingesetzt.

Quellcode 4.2: Fünffach abgerollter Kern eines FIR-Filters

```
#define LOOPBODY acc += (int32_t)(*coeffp++) * (int32_t)(*inputp--)

int k;
int repeat = filterLength / 5;
int left = filterLength % 5;

for (k = 0 ; k < repeat; k += 1) {
    LOOPBODY; LOOPBODY; LOOPBODY; LOOPBODY; LOOPBODY;
}

for (k = 0 ; k < left; k++) {
  LOOPBODY;
}
```

Interessant ist eigentlich nur der Teil, in dem die Schleifeniterationen abgerollt werden. Das wesentliche Ziel ist es, die Schleifeniterationen mit geringer Überlappung und somit höherer Parallelität auszuführen. Das soll ermöglicht werden, indem durch die blockweise Ausführung Kontrollflussabhängigkeiten und -befehle eliminiert werden, die beim GAP Synchronisationspausen erforderlich machen würden (siehe Ab-

bildung 4.37). Da das Schleifenabbruchkritierium seltener überprüft wird, werden zusätzlich weniger Befehle ausgeführt, was die Ausführung der Schleifen zusätzlich beschleunigt.

Allerdings können auch negative Effekte auftreten (siehe z. B. Artikel [170]). Da mehr Programmcode erzeugt wird, können mehr Fehlzugriffe beim Befehlszwischenspeicher auftreten. Wird der Abrollfaktor ungünstig gewählt, so kann bei Ausführung auf dem GAP die abgerollte Schleife zu lange sein oder zu viele limitierte Ressourcen benötigen, sodass sie nicht komplett in das Array eingebaut werden kann. Zudem hat die Anzahl der einzeln auszuführenden Iterationen einen großen Einfluss, da der dafür nötige Programmcode separat und zusätzlich geladen werden muss und nur relativ selten ausgeführt wird.

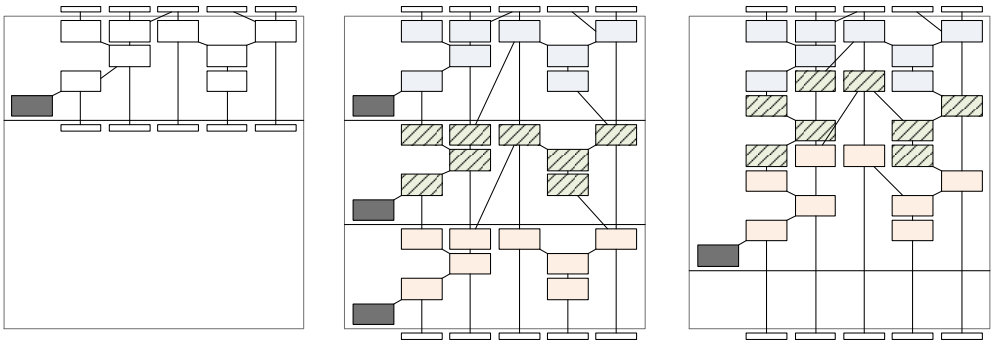


Abbildung 4.37.: Schema der Abbildung einer Schleife ohne Abrollen (links), dreifach abgerollt mit Prüfen des Abbruchkritierums nach jeder Phase (mitte) und ohne Überprüfung (rechts)

Um den Einfluss des Abrollfaktors auf die Programmausführung bei SimpleScalar und GAP zu zeigen wird der bereits erwähnte FIR-Filters verwendet. Er realisiert einen Tiefpass-Filter und besitzt 27 Filter-Koeffizienten. Für jeden Wert, der aus einer Audio-Datei gelesen wird, muss mit jedem Koeffizienten eine Multiply-Accumulate-Operation (MAC) durchgeführt werden. Diese Schleife wird mit Abrollfaktoren zwischen 1 und 12 abgerollt. Abbildung 4.38 zeigt die Ergebnisse; für den GAP wurde die Beschränkung auf eine Multiplikationseinheit gelockert, es können maximal acht Multiplikationen pro Konfiguration stattfinden.

Ein Vergleich der Werte für die Gesamtprogrammlaufzeit mit der Anzahl der Zeilen pro Konfiguration und der Anzahl an einzeln auszuführenden Iterationen (siehe Abbildungen 4.39(a) und (b)) lässt für das Beispiel folgende Schlüsse zu:

- Bereits ein geringer Abrollfaktor genügt, um relativ gute Ergebnisse zu erreichen.
- Die Anzahl einzeln auszuführender Iterationen hat einen negativen Einfluss auf die Performance des Systems. Besonders im gewählten Beispiel, bei dem nur 27 Iterationen ausgeführt werden, ist das deutlich zu beobachten. Mit Abrollfaktor 9 kann die Schleife komplett ausgeführt werden, mit diesem Wert werden sehr gute Ergebnisse erzielt.

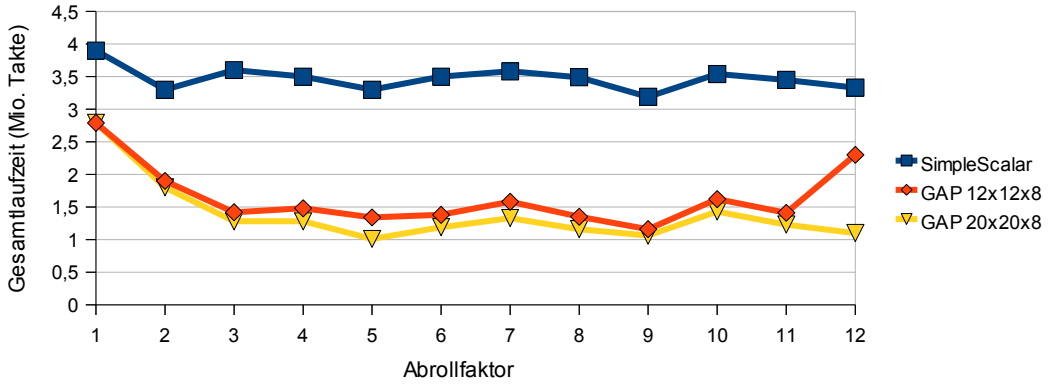


Abbildung 4.38.: Vergleich der Gesamtlaufzeiten für GAP und SimpleScalar für verschiedene Abrollfaktoren

- Es ist wichtig, dass der Abrollfaktor nicht zu hoch gewählt wird. Der optimale Wert für GAP 20x20x8 ist fünf, hier hat der abgerollte Teil der Schleife gerade noch in einer Konfiguration Platz (19 Zeilen werden benötigt).

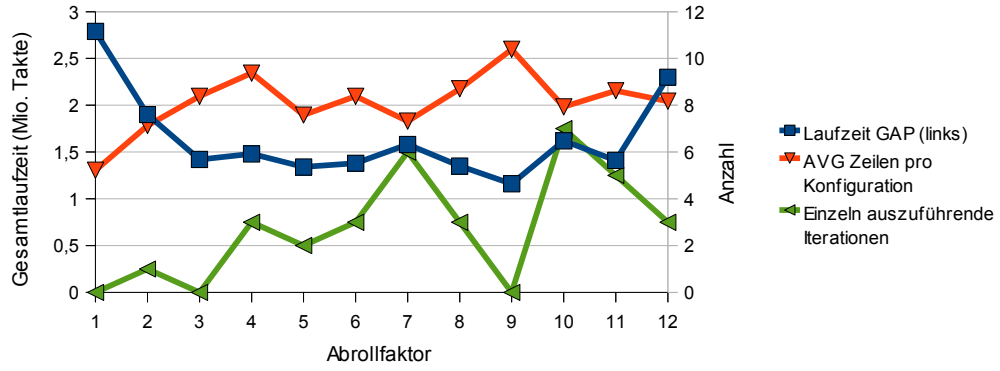
Ein besonderes Code-Konstrukt zum Abrollen von Schleifen ist das *Duff's Device*[7], Quellcode 4.3 zeigt ein Beispiel. Hier wird durch ein Switch-Statement die Schleife so umgebaut, dass kein separater Code benötigt wird, um einzelne Iterationen abzuarbeiten. Das Duff's Device wird häufig eingesetzt aber auch häufig kritisiert; Grund dafür ist die schlechte Lesbarkeit und dass eventuell effektivere Compiler-Optimierungen dadurch verhindert werden könnten. Bei Evaluierung mit dem GAP und dem SimpleScalar-Simulator wurden sehr ähnliche Werte wie für normales Abrollen festgestellt, es bringt also für das gewählte Beispiel aus Performance-Sicht weder Vor- noch Nachteile.

Quellcode 4.3: Duff's Device für vierfach abgerollten FIR-Kernel

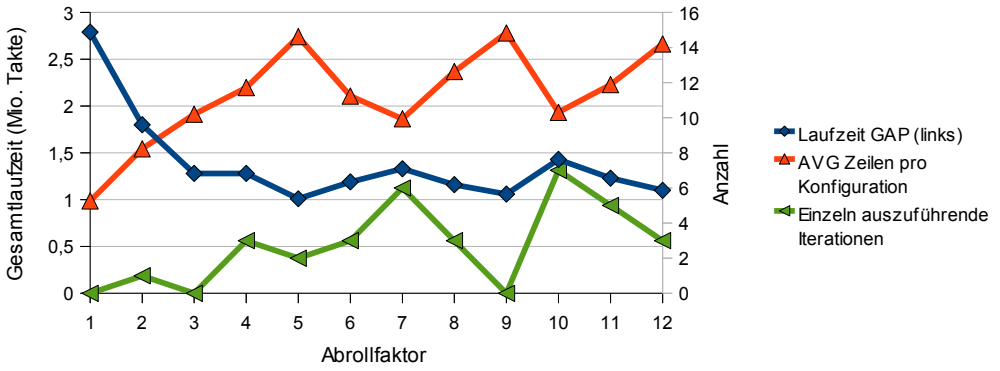
```
#define LOOPBODY acc += (int32_t)(*coeffp++) * (int32_t)(*inputp--)

int left = (filterLength + 3) / 4;

switch(filterLength % 4) {
case 0:  do { LOOPBODY;
case 3:       LOOPBODY;
case 2:       LOOPBODY;
case 1:       LOOPBODY;
         } while(--left > 0);
}
```

Sollen Schleifen automatisch abgerollt werden, so ist also wichtig, den Abrollfaktor

[7] siehe `http://www.lysator.liu.se/c/duffs-device.html`



(a) Werte für GAP mit 12x12x8-Array



(b) Werte für GAP mit 20x20x8-Array

Abbildung 4.39.: Vergleich von Gesamtlaufzeit (linke Achse) mit der durchschnittlich verwendeten Anzahl an Zeilen pro Konfiguration (rechte Achse) und den einzeln auszuführenden Iterationen (rechte Achse)

mit Bedacht zu wählen. Es muss pro Schleife ein optimaler Faktor bestimmt werden, sodass die abgerollte Schleife eine vorher festzulegende Anzahl an Konfigurationsebenen möglichst effektiv verwendet, also eine oder zwei Ebenen möglichst vollständig füllt. Dabei sind im Besonderen Ressourcenbeschränkungen zu beachten, z. B. die Anzahl an Multiplikatoren.

Bei Implementierung in einem Post-Link-Optimizer wie GAPtimize ist die Berechnung der Schleifengrenzen ein nicht einfach zu lösendes Problem. Schleifen abzurollen ist nur dann sinnvoll, wenn einige Schleifeniterationen ohne zwischenzeitliches Überprüfen der Abbruchbedingung ausgeführt werden können. Das ist besonders problematisch, wenn vor Ausführung der Schleife nicht klar ist, nach wie vielen Iterationen sie abzubrechen ist. Ansonsten entsteht im GAP eine Synchronisationsbarriere; in diesem Fall könnte dann statt in der selben Ebene zu bleiben wieder an den Beginn der selben Ebene gesprungen werden, was keine zusätzliche Zeit benötigen würde.

Da die Bestimmung eines optimalen Abrollfaktors durch die Array-Größe beeinflusst wird, muss das Programm, um optimale Performance zu erlangen, für jede

Array-Größe separat modifiziert werden. Man spricht von einer adaptiven Optimierung (siehe später Kapitel 5.2 und Kapitel 5.4).

Normalerweise wird das Abrollen von Schleifen bei der Codeübersetzung durch den Compiler erledigt. Auch der GCC, mit dem Code für den GAP übersetzt werden kann, unterstützt es, verwendet aber eine sehr einfache Heuristik dazu. Aus diesem Grund ist die Option für den GAP nicht verwendbar.

Dennoch ist es nur bedingt sinnvoll, das Abrollen von Schleifen in GAPtimize einzubauen, da neuere Versionen des GCC optimierte Heuristiken für das Abrollen verwenden, die noch dazu sehr einfach beeinflusst werden können. Beispielsweise bietet GCC in der aktuellen Version 4.6 Parameter, mit denen festgelegt werden kann, wie viele Instruktionen eine abgerollte Schleife haben darf (maximal und im Durchschnitt) und wie oft eine Schleife maximal abgerollt werden darf[8].

Ein mögliches Forschungsziel wäre es also, optimale Parameter für eine Heuristik zum Abrollen von Schleifen zu finden, so dass für eine gegebene GAP-Konfiguration und ein gegebenes Programm maximale Performance erreicht werden kann. In diesem Kontext ist als verwandte Arbeit beispielsweise der Artikel von Stephensson et al. [147] zu erwähnen, in dem mit zwei verschiedenen Modellen anhand von Schleifeneigenschaften optimale Abrollfaktoren für Schleifen vorhergesagt werden. Auch Monsifrot et al. [114] beschäftigen sich mit genau diesem Thema.

Ergebnisse wären bei Unterstützung des GAP durch eine aktuelle Compiler-Version zudem wesentlich einfacher zu erreichen.

4.6.2. Software Pipelining

Der GAP besitzt viele Funktionseinheiten, die parallel Befehle ausführen können. In Abschnitt 4.4 wurde gezeigt, dass im Normalfall nur eine geringe Anzahl dieser Einheiten tatsächlich verwendet wird. Die wesentliche Ursache dafür sind Abhängigkeiten unter den Befehlen, die dafür sorgen, dass Befehle in nacheinander folgenden Zeilen im Array eingebaut werden müssen. Um die Parallelität zwischen Befehlen zu erhöhen, wird das Prinzip einer mehrstufigen Pipeline in einem Prozessor (vgl. Kapitel 2.1.1) auf Schleifen übertragen. Statt immer nur einen Befehl (Iteration) gleichzeitig auszuführen, wird der Befehl bzw. die Iteration in mehrere Schritte unterteilt, die dann zeitgleich ausgeführt werden können. Die so entstehende Struktur wird Software Pipeline (vgl. Lam [89]) genannt.

Abbildung 4.40 zeigt ein Beispiel. Die Schleife besteht ursprünglich aus den Schritten **ABCD**; jeder der Schritte besteht aus einem oder mehreren Befehlen. Diese Schritte werden immer wieder nacheinander ausgeführt, bis ein Abbruchkriterium gilt und die Schleife mit **D** verlassen wird. **B** kann erst ausgeführt werden, wenn **A** abgeschlossen ist, da davon auszugehen ist, dass **B** Ergebnisse verwendet, die in **A** berechnet wurden (echte Datenabhängigkeiten). Wird diese Schleife zu einer Pipeline umgebaut, so ist das Ziel, nach einer „Warmlaufphase“, dem *Prolog*, vier Iterationen der Schleife gleichzeitig zu berechnen. Dazu wird der *Kernel* **DCBA** nacheinander ausgeführt. In

[8] Siehe GCC-Dokumentation: `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

diesem Zustand kann mit jeder Ausführung des Kernels eine Iteration abgeschlossen werden. Die Ausführung des Kernels sollte wesentlich weniger Zeit benötigen als die Ausführung der ursprünglichen Schleife **ABCD**, da es unter den Schritten des Kernels keine echten Datenabhängigkeiten mehr geben kann. Diese Abhängigkeiten werden in Schleifenabhängigkeiten (*loop carried dependencies*) umgewandelt, d. h. jede Ausführung des Kernels ist von den Ergebnissen der letzten Ausführung des Kernels abhängig. Um die Schleife dann zu beenden und die noch offenen Iterationen abzuschließen wird ein *Epilog* ausgeführt.

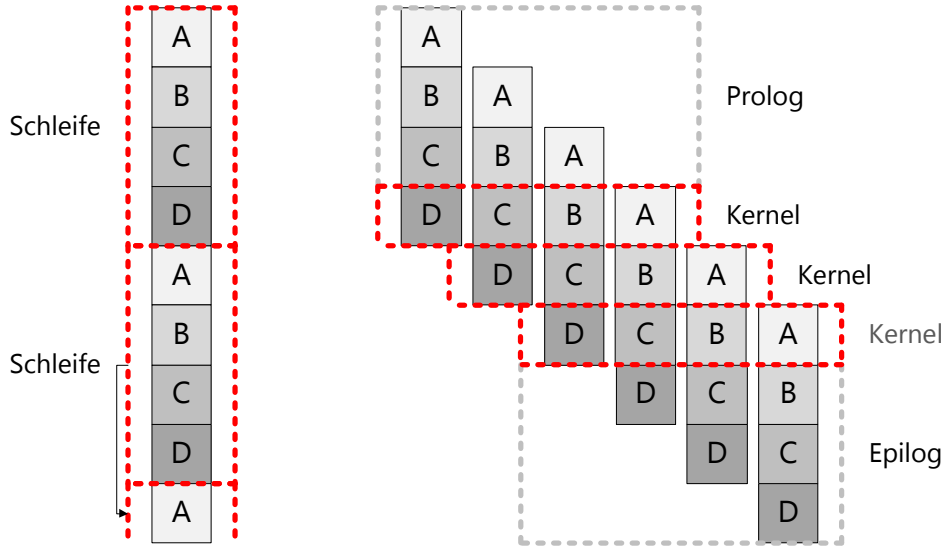


Abbildung 4.40.: Beispiel für Software Pipelining: Schleife in normaler Form (links, ca zwei Iterationen) und als Software Pipeline (rechts, sechs Iterationen)

Um eine Schleife so umzubauen, dass sie eine Software Pipeline darstellt, können verschiedene Verfahren angewendet werden. Neben manueller Implementierung im Quellcode des Programms gibt es verschiedene Scheduling-Algorithmen, die Schleifen parallelisieren können, z. B. *Rotation Scheduling* von Chao et al. [27], *Iterative Modulo Scheduling* von B. Ramakrishna Rau [131] und *Swing Modulo Scheduling* von Llosa et al. [97, 96], um nur wenige zu nennen. Iterative Modulo Scheduling und Swing Modulo Scheduling sind wahrscheinlich die bekanntesten Verfahren, da sie in den Intel Compiler ICC bzw. in GCC integriert wurden. Einen detaillierten Überblick über Software Pipelining (SWP) und die damit verbundenen Probleme gibt [3]. Generell ist SWP ein stark „beforschtes“ Gebiet; die Anzahl an Veröffentlichungen dazu ist extrem hoch.

Für rekonfigurierbare Architekturen wird SWP gerne verwendet, um Schleifen auf der Hardware abzubilden. Beispiele sind Modulo Scheduling für den GAPR-Prozessor [22] und mit dem Compiler DRESC [113] bzw. Artikel [121] allgemein für *Coarse Grained Reconfigurable Architectures* (CGRA). Allerdings gelten bei all diesen Architekturen nicht die selben Beschränkungen wie beim GAP, beispielsweise können Instruktionen beim GAP nicht Compiler-gesteuert beliebig auf den Funktionseinhei-

ten platziert werden. Dann arbeiten die Funktionseinheiten mit wesentlich höherem Grad an Parallelität, beispielsweise ist es meist möglich, dass alle Funktionseinheiten gleichzeitig arbeiten und somit mehrere Schleifeniterationen tatsächlich parallel berechnen. Das ist beim GAP nicht möglich, da immer nur eine „Front“ an Funktionseinheiten gleichzeitig aktiv ist. Somit sind diese Arbeiten nicht direkt auf den GAP übertragbar (vgl. Kapitel 2.2.3).

Man erhofft sich von SWP einen höheren Grad an Parallelität auf Befehlsebene (ILP). Beim GAP würde das bedeuten, dass mehr Befehle pro Zeile im Array platziert werden können, somit sollte sich die Höhe der Konfiguration reduzieren. Das führt zu (a) schnellerer Ausführung des Codes und (b) eventuell zu weniger Neu-Konfigurationen oder Wechseln der Konfigurationsebene.

Für eine Implementierung als Post-Link-Optimierung in GAPtimize gibt es einige Probleme, die schwierig und/oder aufwändig zu lösen sind. Am schwerwiegendsten ist, dass die Register bereits zugewiesen wurden, dabei werden einige wesentliche Entscheidungen getroffen, die nur sehr schwer rückgängig gemacht werden können. Das führt zu vielen falschen Datenabhängigkeiten. Außerdem ist es sehr schwer, die Abhängigkeiten zwischen Zugriffen auf den Speicher zu rekonstruieren. Eine einfache Annahme wäre, dass nur die Reihenfolge von Lesezugriffen veränderbar ist und die Lesezugriffe relativ zu Schreibzugriffen wie im Eingabeprogramm positioniert sein müssen (vergleichbar mit *Memory disambiguation*). Dadurch gehen weitere Möglichkeiten zur Optimierung verloren. Das dritte Problem ist die Wiederherstellung der Schleifenzähler und der Abbruchbedingungen (siehe 3.5.3).

Eine Implementierung auf niedriger Ebene wurde von Markus Pister [124] und Tanya M. Lattner [91] vorgestellt. Die Ergebnisse von Lattner sind eigentlich ernüchternd, es kann nur geringe Beschleunigung mit der Implementierung von Swing Modulo Scheduling erreicht werden. Grund dafür ist, dass die Zielarchitektur sehr wenige Recheneinheiten besitzt, somit dadurch noch gravierendere Restriktionen auftreten. Pister erwähnt weder in seinem Artikel noch in der Diplomarbeit, auf der der Artikel basiert, Werte für die Beschleunigung der Ausführung. Weitere Artikel stellen oft den erzielten Speed-Up nur für die veränderten Schleifen vor, nicht aber für das gesamte Programm.

Beim Versuch, Swing Modulo Scheduling in GAPtimize zu implementieren, wurden einige Hindernisse erkannt. Erstens waren die Gelegenheiten, SWP anzuwenden, sehr selten, da die Schleifen dazu bestimmte Bedingungen erfüllen müssen (keine Sprünge, wenig Speicherzugriffe, die Anzahl der Iterationen muss vorab bekannt sein etc.). Zweitens stellte es sich als großes Problem dar, die Schleifen rechtzeitig abzubrechen, sodass nicht zu viele Iterationen ausgeführt werden. Drittens war die Anzahl freier Register normalerweise sehr gering, sodass zusätzlicher *Spill code* eingefügt werden müsste, der wiederum negative Auswirkungen auf die Performance hat. Viertens führen die zu pessimistisch berechneten Datenabhängigkeiten oft dazu, dass eine Schleife oft nicht unterteilt werden kann, sodass kein Prolog oder Epilog gebildet werden kann. Diese Probleme sind in dieser Intensität nicht vorhanden, wenn die Registerzuweisung erst nach dem Scheduling erfolgt.

Starke Parallelen gibt es zwischen der statischen Spekulation (siehe Abschnitt 4.4)

und SWP. Werden mit der statischen Spekulation Instruktionen über eine Schleifenrücksprungkante verschoben (vgl. Abbildung 4.24, Seite 86), so können sie dann (theoretisch) parallel zum Ende des restlichen Schleifenkörpers ausgeführt werden. Nichts anderes wird bei SWP angestrebt, wobei hier bei entsprechend aufgebauten Schleifen der Grad an Parallelität wesentlich höher sein kann, falls mehr als zwei Iterationen der Schleife im Kernel ausgeführt werden. Die statische Spekulation ist also eine einfache Art von SWP.

Mit Version 4.4 verfügt der Compiler GCC über eine Implementierung von SWP durch Swing Modulo Scheduling [65]. Allerdings sind auch hier keine Performance-Daten verfügbar. Zudem scheint diese Implementierung noch in einem sehr frühen Stadium und nicht immer anwendbar zu sein. Es bleibt zu hoffen, dass daran weiter gearbeitet wird. Sollte eine aktuelle Version des GCC für den GAP portiert werden, so könnte er ebenso davon profitieren.

Interessant wäre es, für den GAP die bedingte Befehlsausführung mit SWP zu kombinieren. Dadurch könnten Schleifen mit mehr als einem Block optimiert werden (vgl. [167]). Genauso wäre es denkbar, den GAP um eine Hardware-Unterstützung für SWP ähnlich zu den rotierenden Registerbänken der i64-Architektur zu erweitern.

4.6.3. Verkürzung des kritischen Pfades

Auch andere Prozessorarchitekturen, z. B. VLIW-Prozessoren und superskalare Prozessoren, werden oft von Datenabhängigkeiten behindert. Wesentlich ist dabei der kritische Pfad unter den Befehlen.

Enthält der kritische Pfad Gegenabhängigkeiten und Ausgabeabhängigkeiten, so besteht manchmal die Möglichkeit, durch beispielsweise optimiertes Scheduling, die Verwendung anderer Register oder die erneute Berechnung von Ergebnissen diese Abhängigkeiten zu eliminieren. In den Jahren 1995 und 1996 stellten drei Forschergruppen dazu Artikel vor [136, 55, 23].

Könnte mit diesen Methoden der kritische Pfad tatsächlich verkürzt werden, so wären für den GAP theoretisch niedrigere Konfigurationshöhen zu erwarten. In der Praxis ist aber zu beachten, dass durch die vor jeder Sprung-Instruktion nötige Synchronisation der Einfluss von Ausgabe- und Gegenabhängigkeiten stark reduziert wird: Es sind nur die Datenabhängigkeiten in einzelnen Blöcken relevant, die aufgrund der normalerweise niedrigen Block-Längen sehr kurz sind.

Deshalb und da die Vermeidung dieser falschen Datenabhängigkeiten eigentlich Aufgabe jedes Compilers ist, wurden derartige Techniken für den GAP nicht näher untersucht.

5. Gleichzeitige Optimierung von Hardware- und Software-Parametern

Der Grid-ALU-Prozessor (GAP) würde sich für eingebettete Systemen mit hohen Performance-Anforderungen eignen. Eingebettete Systeme zeichnen sich oft dadurch aus, dass der auf ihnen ausgeführte Programmcode nur selten verändert wird – meist werden sie einmalig programmiert und arbeiten dann über Jahre. Somit wäre es bei entsprechenden Performance-Gewinnen durchaus vertretbar, wenn die Kompilierung von Programmcode längere Zeit in Anspruch nimmt.

Das Programm kann dann so angepasst werden, dass es auf gegebener Hardware optimale Performance erreicht. Hierfür könnte eine von mehreren Versionen des GAP mit unterschiedlich großen Arrays so gewählt werden, dass die gewünschte hohe Performance erreicht wird.

In der Theorie lässt sich beides zum einen statisch, also einmalig vor Bereitstellung des Systems, wie auch dynamisch, also bei der Verwendung des Systems, durchführen. Als dynamische Methoden könnten dynamische Programm-Optimierungen, Just-in-Time-Kompilierung und rekonfigurierbare Systeme zum Einsatz kommen. Allerdings verursachen diese Methoden einen hohen zusätzlichen Aufwand und sind für den GAP derzeit nicht verfügbar. In diesem Kapitel werden ausschließlich statische Verfahren untersucht.

Ziel ist es, für ein einzelnes oder mehrere Programme zum einen eine optimale Hardware-Konfiguration zu finden, mit der maximale Performance in den gegebenen Beschränkungen erreicht werden kann, und zum anderen eine Teilmenge an Programmoptimierungen, die im gegebenen Fall die Performance noch weiter verbessern können. Das lässt sich in ein Problem übersetzen, das mit Methoden zur *Automatischen Suche in einem Parameterraum* (*Automatic Design Space Exploration*, ADSE) gelöst werden kann.

In Abschnitt 5.1 werden Grundlagen erklärt, das *Framework for Automatic Design Space Exploration* (FADSE) als Tool und verwandte Arbeiten dazu vorgestellt. Anschließend werden mit FADSE optimale Hardware-Parameter für eine Menge an Benchmarks gesucht. Dazu wird erst eine Abschätzung der Hardware-Komplexität des GAP eingeführt und darauf aufbauend werden die Ergebnisse der ADSE genau untersucht (Abschnitt 5.2). Der Parameterraum wird erweitert, indem in Abschnitt 5.3 zusätzlich zu den Hardware-Parametern die in den vorhergehenden Kapiteln vorgestellten Code-Optimierungen kombiniert und evaluiert werden. Als Spezialisierung wird in Abschnitt 5.4 nur je ein Benchmark gewählt und dieser zusammen mit der Hardware so optimiert, dass er optimale Performance erreicht. Man spricht dann von adaptiver Optimierung der Hardware und des gewählten Benchmarks. Abschnitt 5.5 fasst die Ergebnisse zusammen.

5.1. Automatische Suche im Parameterraum

5.1.1. Grundlagen

Ziel ist es, zum Zweck der Performance-Optimierung sowohl die Hardware-Parameter des GAP wie auch die in GAPtimize umgesetzten Optimierungen passend zu wählen und zu konfigurieren. All diese Einstellungen lassen sich als Parameter ausdrücken. Für jeden Parameter gibt es viele mögliche Belegungen. Man kann diese n Parameter als Dimensionen eines n-dimensionalen *Parameterraums* (*Parameter space*) $P \subset \mathbb{Z}^n$ sehen. Jeder Punkt in diesem Raum entspricht einer Belegung der Parameter mit Werten, also einer *Konfiguration* $p = (p_0, ..., p_{n-1})$.

Um die Qualität einer Konfiguration messen zu können, muss sie ausgewertet werden. Bei Prozessorarchitekturen und Code-Optimierungen findet das (meist) durch Anwendung der Optimierungen und Simulation[1] statt. Aus den generierten Ergebnissen werden Werte für ein oder mehrere *Optimierungsziele* (engl. *Objectives*) ermittelt. Der von den Zielen aufgespannte Raum heißt *Zielraum* (engl. *Objective space*) $O \subset \mathbb{R}^m$ mit Punkten $o = (o_0, ..., o_{m-1})$.

Durch die Auswertung einer Konfiguration wird ihr somit ein Punkt im Zielraum zugeordnet, die Auswertung ist also eine Projektion $f(p) := P \rightarrow O$ auf den Zielraum:

$$(p_0, ..., p_n) \rightarrow (o_0, ..., o_m)$$

Bei Zielen kann entweder eine Minimierung oder eine Maximierung angestrebt werden; zur Vereinfachung werden im Folgenden alle Ziele minimiert. Soll der Zahlenwert eines Zieles maximiert werden, so kann stattdessen der Kehrbruch minimiert werden.

Gibt es mehr als ein Ziel, so ist nicht ohne Weiteres klar, welche von zwei Konfigurationen de facto besser ist, da keine (natürliche) Ordnungsrelation definiert ist. Dazu wird die *Dominanzbeziehung* verwendet:

Definition 2. *Eine Konfiguration p_1 dominiert eine Konfiguration p_2, wenn p_1 für mindestens ein Ziel bessere Werte als p_2 besitzt und für alle anderen Ziele mindestens gleich gut ist wie p_2.*

Alle Konfigurationen, die von keiner anderen Konfiguration dominiert werden, bilden die *Pareto-Front* im Zielraum und sind *Pareto-optimal*. Abbildung 5.1 zeigt diese Zusammenhänge.

Zur Bestimmung der Pareto-Front – das ist das eigentliche Ziel einer ADSE – gibt es mehrere Verfahren mit unterschiedlichem Aufwand und unterschiedlich hohem Risiko, dass eine hinreichend gute Annäherung der Pareto-Front nicht gefunden wird. Die drei am häufigsten anzutreffenden Ansätze sind[2]:

[1] Ergebnisse können ebenso mit Schätzverfahren (*Performance prediction*, z. B. [52]) oder heuristischen Verfahren wie beispielsweise SimPoints [123] generiert werden.

[2] Künzle et al. [86] nennt insgesamt fünf Verfahren.

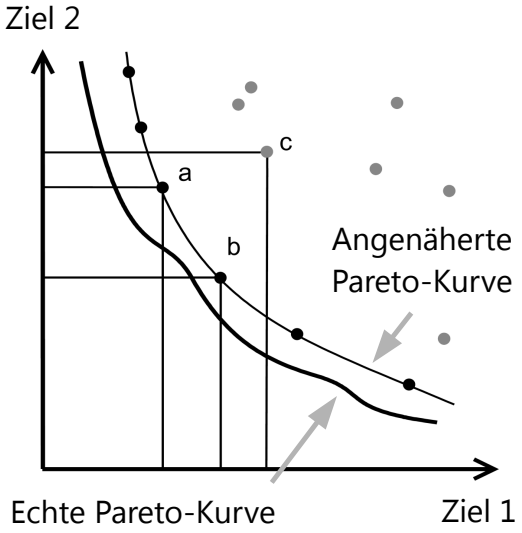


Abbildung 5.1.: Diagramm des Zielraumes eines Problems, bei dem für zwei Ziele minimale Werte gefunden werden müssen. Die Punkte *a* und *b* dominieren Punkt *c*; die Punkte *a* und *b* werden nicht dominiert, somit sind sie Pareto-optimal.

- **Vollständige Exploration:** Alle Punkte des Parameterraums werden untersucht, somit werden alle Punkte des Zielraums bestimmt. Dadurch wird die tatsächliche Pareto-Front sicher gefunden. Allerdings muss eine sehr große Anzahl an Konfigurationen ausgewertet werden, das dauert oft viel zu lang.

- **Manuelle Exploration:** Es werden manuell die auszuwertenden Punkte festgelegt. Je nach Erfahrung oder Glück bei der Selektion von Punkten wird die Pareto-Front unterschiedlich gut angenähert. Die Suche kann nach kurzer Zeit beendet werden, führt aber nicht sicher zu einer brauchbaren Annäherung der Pareto-Front, da unter Umständen lokale Optima gefunden werden (vgl. Probleme bei *Greedy*-Algorithmen).

- **Heuristische Exploration:** Es wird nur ein geringer Teil der Punkte im Parameterraum ausgewertet. Diese Punkte werden aber mit bestimmten Algorithmen sowohl logisch nachvollziehbar als auch zufällig gewählt, sodass lokale Optima vermieden werden. Damit wird bei geringer Laufzeit eine meist sehr gute Approximation der Pareto-Front erreicht.

Bei großem Parameterräumen sind meist nur heuristische Verfahren anwendbar. Es bieten sich dann vor allem von der Natur inspirierte evolutionäre Algorithmen an; meist handelt es sich um genetische (z. B. NSGA-II [47]) oder Partikel-Schwarm-Algorithmen (z. B. SMPSO [119]). Bei evolutionären Algorithmen werden Gruppen von Konfigurationen untersucht, die Populationen genannt werden (siehe Abbildung 5.2 für NSGA-II). Die Konfigurationen werden dementsprechend als Individuen

bezeichnet. Populationen werden von den zuletzt betrachteten Populationen abgeleitet; somit werden immer neue Generationen von Individuen erzeugt. Dadurch nimmt die Qualität der Annäherung der Pareto-Front immer mehr zu. Jedoch ist nie sicher, dass die tatsächliche Pareto-Front gefunden wurde; sie bleibt weiter unbekannt.

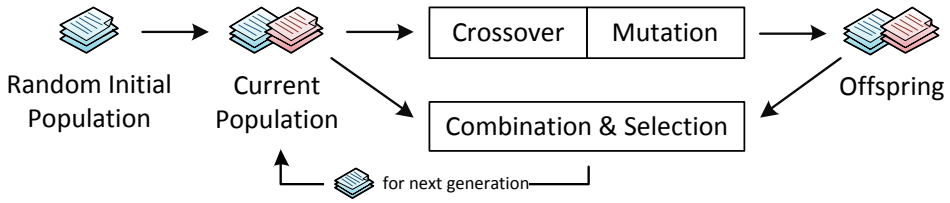


Abbildung 5.2.: Struktur von NSGA-II: Eine zufällig erstellte initiale Population wird ausgewertet und stellt die erste Generation dar. Mit *Crossover* und *Mutation* wird eine neue Population erzeugt und ausgewertet, sie heißt *Offspring*. Aus der aktuellen Generation und dem Offspring wird mit *Selection* eine neue Population erzeugt, die dann als nächste Generation verwendet wird.

Das Ergebnis der ADSE ist eine Menge an Konfigurationen, die durch keine anderen untersuchten Konfigurationen dominiert werden und so die Pareto-Front approximieren. Die Qualität dieser Annäherung lässt sich nicht beurteilen, da dazu die unbekannte echte Pareto-Front benötigt werden würde.

Wird ein Algorithmus zur ADSE verwendet, der mit Generationen arbeitet, so können diese aber miteinander verglichen werden und es kann ermittelt werden, ob die Suche noch kontinuierlich bessere Ergebnisse liefert. Dazu wird das *Hypervolume* berechnet. Es stellt (bei einem Minimierungsproblem) bezüglich eines festzulegenden Referenzpunktes das Verhältnis der Fläche zwischen ihm und der Pareto-Approximation sowie dem Nullpunkt und dem Referenzpunkt dar (siehe Abbildung 5.3). Abhängig von der Form der echten Pareto-Front und der Wahl des Referenzpunktes kann das Hypervolume sehr unterschiedliche Zahlenwerte zwischen 0 und 1 annehmen. Deshalb lässt sich, wenn die echte Pareto-Front unbekannt ist, anhand der Hypervolume-Werte keine Aussage über die Qualität der Approximation ableiten.[3]

Die angenäherte Pareto-Front drückt aus, welche besten Ergebnisse gefunden wurden, beschreibt also ein Gebiet im Zielraum. Interessanter ist aber, wie Konfigurationen aufgebaut sein sollten, um sehr gute Werte zu zeigen. Darüber gibt die Pareto-Approximation aber keine Auskunft, da sie meist aus wenigen Konfigurationen aufgebaut ist, die im Parameterraum stark gestreut sein können.

Um Bereiche im Parameterraum zu beschreiben, in denen sehr gute Konfigurationen stark vertreten sind, kann entweder manuell oder automatisch vorgegangen werden.

Für ein weitgehend automatisiertes Verfahren werden mit einem Entscheidungsbaum (beispielsweise mit dem Algorithmus C4.5 von Ross [134]) alle im Rahmen

[3]Weitere Metriken, die aber im Folgenden nicht verwendet werden, sind *Coverage* und *Seven point average distance* [35].

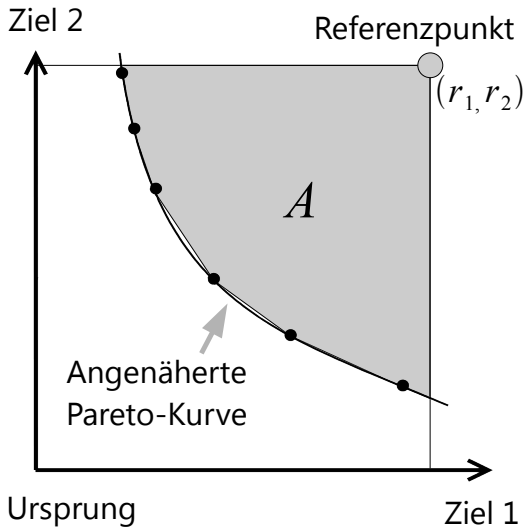


Abbildung 5.3.: Angenäherte Pareto-Front für zwei zu minimierende Ziele; das Verhältnis der grauen Fläche zwischen Approximation und Referenzpunkt und der Fläche zwischen dem Ursprung und dem Referenzpunkt ist das Hypervolume: $h = \frac{A}{r_1 * r_2}$

der ADSE ausgewerteten Konfigurationen klassifiziert. Als Kriterium wird verwendet, ob die Konfiguration in einer ε-Umgebung um die Pareto-Approximation liegt oder nicht. Ein Entscheidungsbaum ist ein binärer Baum, an dessen Verzweigungen überprüft wird, ob eine Variable zwischen zwei Werten liegt oder nicht. Jedem Blatt des Baumes ist eine der Klassifikationsklassen zugeordnet. Die Klassifizierung erfolgt dann, indem von der Wurzel aus so lange Bedingungen überprüft und somit Verzweigungen genommen werden, bis ein Blatt letztendlich erreicht wird. Von diesem Entscheidungsbaum können dann sehr leicht Bedingungen abgeleitet werden, die für sehr gute Konfigurationen zutreffen müssen. Dazu wird für eine Teilmenge der Parameter jeweils mit einer unteren und oberen Schranke ein Bereich festgelegt. Sind für eine Konfiguration alle Parameter in den vorgegebenen Bereichen, sodass eine Bedingung komplett erfüllt wird, so ist davon auszugehen, dass die Konfiguration sehr nah an der Pareto-Approximation liegen könnte (vgl. Abbildung 5.4). Weitere Details und wie der Entscheidungsbaum zur Beschleunigung einer ADSE verwendet werden kann finden sich im Artikel [76].

Alternativ können Bedingungen grafisch anhand von Diagrammen abgeleitet und somit Zusammenhänge zwischen Parametern herausgearbeitet werden. Da damit wesentlich mächtigere Schlussfolgerungen als mit den oben genannten Bedingungen möglich und diese viel intuitiver dargestellt werden können, wird diese manuelle Vorgehensweise im Folgenden verwendet.

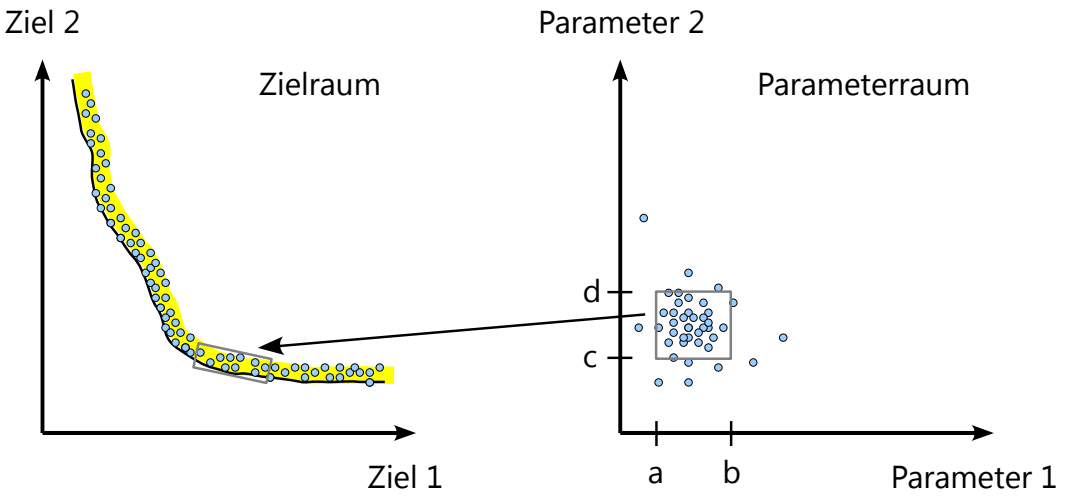


Abbildung 5.4.: Optimierungsproblem mit zwei Parametern und zwei zu minimierenden Zielen; alle im Parameterraum gezeigten Punkte werden auf das Rechteck im Zielraum abgebildet, somit ist die Bedingung $(a < p_1 < b) \wedge (c < p_2 < d)$ dazu geeignet, um diese Konfigurationen hinreichend genau zu beschreiben.

5.1.2. Framework for Automatic Design Space Explorations (FADSE)

Das *Framework for Automatic Design Space Explorations* (FADSE, siehe [20, 21, 80]), ein OpenSource-Projekt, wurde mit dem Ziel entwickelt, den großen Parameterraum, den die Parameter von Mehrkern-Prozessorarchitekturen aufspannen, zu untersuchen.

FADSE ist komplett in Java geschrieben. Dadurch kann es leicht angepasst werden und neue Simulatoren wie der GAP können daran angebunden werden. Die Spezifikation des Parameterraumes und der Zielfunktionen erfolgt in einer XML-Datei, ist also leicht anpassbar. In dieser XML-Datei können Bedingungen für Zusammenhänge unter den Parametern eingetragen werden, z. B. dass Parameter ungültig sind, wenn ein Parameter einen vorgegebenen Wert hat. Die Bedingungen klammern unsinnige Bereiche aus dem Parameterraum aus und beschleunigen somit die DSE (siehe Abschnitt 5.3).

Für die automatische Suche in einem Parameterraum werden aktuelle evolutionäre Algorithmen für mehrere Zielfunktionen aus der jMetal-Bibliothek[4] [53] verwendet. Ein Vergleich der drei Algorithmen NSGA-II, SPEA2 [174] und SMPSO bei Verwendung von FADSE und dem GAP findet sich im Artikel [19].

Im Artikel [80] werden einige Erweiterungen für FADSE vorgestellt, mit denen ADSEs beschleunigt werden können und FADSE robuster auf Fehler reagieren kann. Um die Geschwindigkeit zu erhöhen wird NSGA-2 so angepasst, dass die Auswer-

[4]Internetseite: `http://jmetal.sourceforge.net/`

tung von Konfigurationen nicht mehr sequentiell stattfinden muss, sondern dass in einem Client-Server-Verbund mehrere Client-PCs die Auswertungen parallel ausführen können. Außerdem werden die Ergebnisse zu ausgewerteten Individuen in einer Datenbank gepuffert, sodass sie bei später erneut nötiger Auswertung sehr schnell verfügbar sind. Beide Verfahren reduzieren die benötigte Zeit pro Generation dramatisch. Um auf Fehlern bei der Auswertung von Konfigurationen (z. B. Absturz des Prozessor-Simulators) geeignet reagieren zu können wird versucht, die Konfiguration noch ein weiteres Mal auszuführen. Gibt es auch dabei Probleme, so wird sie als unlösbar markiert. Sollten Probleme allgemeinerer Natur auftreten, also dass z. B. der FADSE-Server ausfällt, so kann eine Textdatei, die nach jeder abgeschlossenen Generation ausgegeben wird, als Checkpoint verwendet werden, um die Suche nicht erneut bei Null beginnen zu müssen.

Speziell für den GAP wurde FADSE so erweitert, dass Simulationen automatisch nach Überschreiten einer pro Benchmark spezifizierbaren Anzahl an Taktzyklen abgebrochen werden; dann kann davon ausgegangen werden, dass die Ausführung des Benchmarks fehlerhaft ist. Außerdem werden nach Abschluss einer Simulation die erzeugten Ausgaben und Dateien mit Referenzdaten verglichen, sodass sichergestellt werden kann, dass die Simulation korrekt ausgeführt wurde.

5.1.3. Alternative Frameworks für die automatische Suche im Parameterraum

Die Suche nach optimalen Parameterkonfigurationen ist ein „Standardproblem“, es gibt viele Programme, die bei seiner Lösung helfen können. Alternativ zu FADSE könnte entweder ein anderes Framework zum Einsatz kommen oder eine andere Software, die speziell für die Probleme bei der Optimierung von Prozessorarchitekturen optimiert wurde.

Im Rahmen des Multicube-FP7-Projekts wurden *Multicube Explorer* zur automatischen Suche im Parameterraum von Mehrprozessor-SoC-Architeturen entwickelt [145]. Es weist viele Ähnlichkeiten zu FADSE auf, beispielsweise unterstützt es mehrere Algorithmen und verfügt mit *Response surface modelling* über ein Verfahren zur Beschleunigung. Dabei wird eigentlich basierend auf den schon ausgewerteten Individuen vorhergesagt, welche Performance neue Konfigurationen vermutlich zeigen werden. Es wird über XML-Dateien konfiguriert. Anders als FADSE unterstützt es aber keine verteilte Evaluation von Individuen. Ähnlich vollständig wie Multicube ist das Framework *PISA* von Künzle et al.[86].

Der Fokus von *Magellan* [82] ist auf Mehrkern-Architekturen, wobei eigentlich nur ein Simulator unterstützt wird (SMTSIM) und die Suche nur eingeschränkt mehrere Zielfunktionen berücksichtigen kann.

Archexplorer.org [50] ist eine Webseite zur kollaborativen Optimierung von Prozessorarchitekturen. Die Idee dahinter ist, dass Anwender Modelle für optimierte Prozessorbestandteile bereitstellen können, aus denen dann versucht wird, einen möglichst guten Prozessor zu konstruieren. Details zu den verwendeten Algorithmen sind nicht genau erläutert, als Simulator wird UNISIM verwendet.

NASA [81] unterstützt nur einen eigens entwickelten Algorithmus mit einer Zielfunktion.

Das Programm *SESAME* [120] bietet wesentlich mehr als nur DSE und stellt den kompletten *System level design flow* mit „design space exploration (DSE), system-level synthesis, application mapping, [und] system prototyping“ bereit.

5.2. Optimale Hardware-Parameter

Anders als viele bekannte Prozessorarchitekturen verfügt der GAP mit dem Array über ein Element, dessen Bedarf an tatsächlicher Chipfläche je nach gewählten Parametern sehr stark variieren kann, da es zwischen $4 \times 4 = 16$ und $31 \times 32 = 992$ Funktionseinheiten umfassen kann. Soll ein tatsächlicher Prozessor basierend auf die GAP-Architektur gefertigt werden, so ist es wichtig, ihn so zu konfigurieren, dass er mit der vorgegebenen Chipfläche die bestmögliche Performance erreichen kann, sie also möglichst effektiv genutzt wird. Eine ähnliche Herausforderung stellt sich, wenn z. B. aufgrund des technischen Fortschritts (vgl. Moore's Law) eine höhere Chipfläche zur Verfügung steht.

In diesem Kapitel werden mit einer DSE optimale Hardware-Parameter für den GAP ermittelt, also das Verhältnis aus Hardware-Aufwand und Performance untersucht. Nach Festlegung des Parameterraumes in Abschnitt 5.2.2 werden die Optimierungsziele definiert (Abschnitt 5.2.3). Dazu wird ein Modell vorgestellt, mit dem die Hardware-Komplexität einer GAP-Konfiguration berechnet werden kann mit dem Ziel, den Hardware-Aufwand zweier Konfigurationen vergleichen zu können. Die Ergebnisse der DSE werden in Abschnitt 5.2.5 erläutert; in Abschnitt 5.2.6 werden die Ergebnisse dieses Kapitels zusammengefasst.

5.2.1. Verwandte Arbeiten

Bislang haben nur Shehan et al. im Artikel [142] bereits eine stark vereinfachte manuelle DSE der Hardware-Parameter des GAP vorgestellt. Es wird kein heuristisches Verfahren verwendet, sondern eine manuelle Exploration durchgeführt. Zudem wird der Befehlszwischenspeicher als fixiert betrachtet, also ausschließlich das Array untersucht. Als einziges Optimierungsziel wird die Performance verwendet und der Hardwareaufwand wird nicht berücksichtigt. Die im Folgenden beschriebene DSE basierend auf [80] weist diese Schwächen nicht auf.

Sheldon und Vahid [144] passen die Parameter einer FPGA-Prozessor-Implementierung so an, dass sie für das auszuführende Programm optimal sind. Dabei wird ein Schätzverfahren verwendet, um die Suchzeit zu reduzieren.

5.2.2. Parameterraum

Die Größe des Arrays ist im GAP das wesentliche Element, das Hardwareaufwand und Performance bestimmt. Zusätzlich kann der Befehlszwischenspeicher in Größe und Organisation passend zu den Anforderungen gewählt werden. Array und Cache bestimmen also den Parameterraum; ihre Parameter werden in Tabelle 5.1 zusammengefasst.

Bei einer erschöpfenden Suche, bei der alle möglichen Konfigurationen simuliert werden, müssten $29 * 28 * 7 * 3 * 9 * 8 = 1.227.744$ Simulationen ausgeführt werden. Sinnvollerweise müssen mehrere Benchmarks verwendet werden, um die Unabhängigkeit der gefundenen Ergebnisse vom ausgeführten Programm sicherzustellen. Werden zehn Benchmarks verwendet und jeder benötigt sechs Minuten, so kann in einer

	Beschreibung	Definitionsbereich
C_r	Array: Zeilen	$\{4, 5, 6, 7, ..., 32\}$
C_c	Array: Spalten	$\{4, 5, 6, 7, ..., 31\}$
C_l	Array: Konfigurationsebenen	$\{1, 2, 4, 8, ..., 64\}$
C_{c1}	Cache: Befehle pro Zeile	$\{4, 8, 16\}$
C_{c2}	Cache: Sätze	$\{32, 64, 128, ..., 8192\}$
C_{c3}	Cache: Zeilen pro Satz	$\{1, 2, 4, 8, ..., 128\}$

Tabelle 5.1.: Parameterraum des GAP

Stunde eine Konfiguration simuliert werden. Bei 32 parallel auf verschiedenen Systemen ausgeführten Simulationen würde die gesamte Auswertung ca. vier Jahre benötigen, das ist eindeutig zu lange. Aus diesem Grund wird ein heuristisches Verfahren verwendet, mit dem nur ein Bruchteil der gesamten Konfigurationen simuliert werden muss.

5.2.3. Optimierungsziele

Es soll die Performance einer Konfiguration des GAP mit dem nötigen Hardwareaufwand der Konfiguration in Zusammenhang gebracht werden. Somit müssen diese beiden Optimierungsziele gemessen werden können.

In Kapitel 2.3.1 wird vorgeschlagen, **Performance** mit dem *IPC*, also der Anzahl der pro Takt ausgeführten Instruktionen zu messen; die Anzahl der Takte wird mit dem taktgenauen Simulator des GAP ermittelt. Da es aber angestrebt wird, alle Optimierungsziele zu minimieren, wird hier und im Folgenden stattdessen der Wert *CPI* verwendet, der die Anzahl an Takten pro Befehl ausdrückt.

Schwieriger ist es, die **Hardware-Komplexität** einer GAP-Konfiguration zu messen oder abzuschätzen. Dazu wird im folgenden Abschnitt 5.2.4 eine Abschätzung vorgeschlagen, mit der die Größe der variablen Teile des GAP im Verhältnis zu einer 32 Bit Integer-ALU ermittelt.

5.2.4. Abschätzung der Hardware-Komplexität des GAP

Wäre eine FPGA-Implementierung oder Ähnliches verfügbar, so könnten verlässliche Zahlen für die benötigten Ressourcen verwendet werden. Leider ist das nicht der Fall, deshalb wird ein Verfahren vorgestellt, mit dem der Hardwarebedarf zweier Konfigurationen des GAP abgeschätzt und verglichen werden kann. Die berechnete Größe wird *Hardware-Komplexität* genannt. Ziel ist es ausschließlich, Vergleichbarkeit zwischen Konfigurationen des GAP herzustellen und nicht, den Hardware-Bedarf des GAP mit anderen Architekturen in Zusammenhang zu bringen.

Die Hardware-Komplexität des GAP setzt sich aus der Komplexität des Prozessor-Frontends H_{front}, der Komplexität des Arrays bestehend aus den ALUs H_{ALUs} und den Speicherzugriffseinheiten H_{LSUs}, der Komplexität des Befehlszwischenspeichers H_{ICache} und des Datenzwischenspeichers H_{DCache}, sowie einiger weiterer Einheiten

Konstante	Wert	Beschreibung
h_{ALU}	1,00	FU mit einer ALU
h_{LSU}	3,50	Speicherzugriffseinheit (LSU)
h_l	0,02	Eine Konfigurationseinheit für eine ALU/LSU
h_r	0,02	Top-Register

Tabelle 5.2.: Elementare Bestandteile des Arrays des GAP und Abschätzung ihrer Komplexität

$H_{constant}$, die aber unabhängig von der Konfiguration nur genaue einmal im GAP vorhanden sind:

$$H = H_{front} + H_{ALUs} + H_{LSUs} + H_{ICache} + H_{DCache} + H_{constant}$$

Da H_{front}, H_{DCache} und $H_{constant}$ konfigurationsunabhängig sind und somit in alle Konfigurationen des GAP die gleiche Größe besitzen, werden sie im Weiteren vernachlässigt. Zur Abschätzung der übrigen Komplexitäten werden in der Literatur Referenzwerte für die Größe der Basiselemente ermittelt. Tabelle 5.2 zeigt die benötigten Elemente und die später festgelegten Werte. Die folgenden drei Quellen stehen zur Verfügung:

- Das Programm **HotSpot** [75] zur thermischen Simulation enthält eine Beschreibung der Flächenaufteilung für den 64-Bit-Prozessor Alpha 21264 [61]. Dieser Grundriss wurde durch genaues Abmessen eines Photos des Chips erstellt.
- **Gupta et al.** stellen in ihrem Artikel [62] Zahlen für die Größe verschiedener Bestandteile eines Prozessors vor, die sie wiederum aus unterschiedlichen Quellen zusammengetragen haben. Davon werden prozessunabhängige Zahlen abgeleitet, mit denen ihre Größe abgeschätzt werden kann.[5]
- Der Simulator **McPat** [94] für Leistungsaufnahme, Zeit- und Temperaturverhalten enthält eine Konfiguration für den 64-Bit-Prozessor Alpha 21364, der den Alpha 21264 erweitert. Die Flächenabschätzung basiert auf Gupta et al. (siehe oben) und Rodrigues [133]

Aus diesen Veröffentlichungen wird die Größe einer 64 Bit Integer-ALU, eines 64 Bit Integer-Registers und eines 64 kB Befehlszwischenspeichers ermittelt. Da der GAP aber ein 32 Bit Prozessor ist, wird die Größe einer 64 Bit Integer-ALU bzw. Integer-Registers halbiert; dieses Vorgehen wird von Gupta et al. [62] unterstützt. Abschließend werden die Zahlen relativ zur Größe einer 32 Bit Integer-ALU normiert (d. h. $h_{ALU} := 1$). Die Ergebnisse zeigt Tabelle 5.3.

Auffallend ist, dass die Werte für einen 64 kB Befehlszwischenspeicher sehr nah beieinander liegen, wohingegen die Werte für ein 32 Bit Integer-Register stark gestreut sind. Dennoch ist ein Integer-Register im Vergleich zu einer Integer-ALU sehr

[5] Die gezeigten Zahlen weisen teilweise eine beträchtliche Varianz auf.

Datenquelle	32 Bit Integer-Register	64 kB Befehlszwischenspeicher
McPat, Alpha 21364	0,031	15,678
HotSpot, Alpha 21264	0,008	16,767
Gupta et al.	0,017	17,131
Mittelwert	0,019	16,525

Tabelle 5.3.: Verhältnis des Hardware-Aufwandes für ein 32 Bit Integer-Register und einem 64 kB Befehlszwischenspeicher im Vergleich zu einer 32 Bit Integer-ALU ($h_{ALU} := 1$)

klein, es wird deshalb $h_r := 0,02$ gewählt. Dem entsprechend wird pro Instruktion, die in einer Konfigurationsebene gespeichert wird, ein Aufwand von $h_l := 0,02$ berechnet. Zwar ist der benötigte Speicher doppelt so groß wie bei einem Register, allerdings ist der sonstige Hardware Aufwand hier wesentlich geringer. Für die Größe einer Speicherzugriffseinheit sind Werte nur in McPat und der Flächenaufteilung des Alpha-Prozessors verfügbar, sie liegen zwischen der 1,5- und der 5,4-fachen Größe einer ALU. Deshalb wird $h_{LSU} := 3,5$ gesetzt.

Mit diesen Konstanten können jetzt H_{ALUs} und H_{LSUs} berechnet werden, wobei C_c, C_r und C_l wie in Tabelle 5.1 definiert die Parameter des GAP-Arrays sind. In beiden Formeln wird zunächst die Komplexität der Funktionseinheiten berechnet und anschließend die Komplexität addiert, die die Konfigurationsebenen verursachen:

$$H_{ALUs} = (C_c * h_r + C_r * C_c * h_{ALU}) + C_r * C_c * C_l * h_l$$
$$H_{LSUs} = C_r * h_{LSU} + C_r * C_l * h_l$$

Mit dem Tool CACTI [154] können Performance- und Flächenwerte für Befehlszwischenspeicher abgeschätzt werden. Für einen Zwischenspeicher, der sehr ähnlich konfiguriert ist wie der im erwähnten Alpha-Prozessor[6], wird eine Fläche von $5,52mm^2$ berechnet. Mit dem Umrechnungsfaktor $16,525/5,52mm^2 = 2,99\frac{1}{mm^2} \approx 3\frac{1}{mm^2}$ kann CACTI verwendet werden, um dynamisch die Größe des Befehlszwischenspeichers H_{ICache} einer GAP-Konfiguration zu berechnen.

Als Beispiel wird die Komplexität H eines GAP mit 12x12x32-Array und 8 kB Befehlszwischenspeicher berechnet:

$$H = H_{ALUs} + H_{LSUs} + H_{ICache}$$
$$H_{ALUs} = (12 * 0,02 + 12 * 12 * 1,00) + 12 * 12 * 32 * 0,02 =$$
$$= 144,24 + 92,16 = 236,40$$

[6]Web-Schnittstelle für CACTI 5.3, 64 kB Gesamtgröße, 128 byte Zeilengröße, 2-fach assoziativ, 1 Speicherbank, 90 nm Fertigungsprozess

$$H_{LSUs} = 12 * 3,50 + 12 * 32 * 0,02 = 49,68$$
$$H_{ICache} = 0,856mm^2 * 3\frac{1}{mm^2} = 2,57$$
$$H = 236,4 + 49,68 + 2,57 = 288,65$$

Bei der Suche im Parameterraum sollen sowohl die Performance des GAP maximiert (also CPI minimiert) wie auch der dazu nötige Hardwareaufwand, der durch die Komplexität H abgeschätzt wird, minimiert werden.

5.2.5. Ergebnisse und Auswertung

Im Artikel [142] werden erstmalig optimale Kombinationen für die Hardware-Parameter des GAP gesucht. Es werden im Folgenden, wie in Artikel [80] begonnen, ähnliche Ziele verfolgt, allerdings wird das Vorgehen wesentlich optimiert.

Erstens wird nicht mehr nur ein Optimierungsziel, nämlich Performance, verfolgt, sondern die Performance wird auch in Zusammenhang zum Hardwarebedarf gestellt. Zweitens wird durch NSGA-II als genetischem Algorithmus eine wesentlich robustere Vorgehensweise im Vergleich zur nicht-erschöpfenden manuellen Exploration verwendet, bei der das Risiko lokale Optima zu finden hoch ist. Mit einem genetischen Algorithmus, der zufällig erzeugte Konfigurationen auswertet, wird dieses Risiko stark reduziert. Drittens werden die Parameter für den Befehlszwischenspeicher untersucht, somit kann beispielsweise gezeigt werden, ob das Array auch bei großen Caches noch einen Mehrwert besitzt.

Zur Auswertung von Konfigurationen wird der taktgenaue Simulator für den GAP entsprechend konfiguriert und es werden zehn Benchmarks[7] ausgeführt. Anschließend wird als Maß für die Performance der Mittelwert über die CPI-Werte der Benchmarks gebildet.

Es wird NSGA-II als genetischer Algorithmus und somit Kern der automatischen Suche im Parameterraum mit FADSE verwendet. Der Algorithmus wird wie von Deb et. al. [47] vorgeschlagen konfiguriert, d. h. die Wahrscheinlichkeit für Rekombination wird auf 0,9 gesetzt, die Wahrscheinlichkeit für Mutation auf 1 geteilt durch die Anzahl an Parametern und der Verteilungsindex auf 20. Als Operatoren werden *Single point crossover* und *Bit flip mutation* sowie *Binary tournament* für die Selektion verwendet. Jede Generation enthält 100 Individuen und die Exploration wird nach 55 Generationen abgebrochen, da ab ca. der 50. Generation keine signifikante Verbesserung mehr zu beobachten ist. Dies zeigt die Entwicklung des Hypervolume, siehe Abbildung 5.5. Die Auswirkungen einer kleineren Generationsgröße werden für den GAP in Artikel [80] untersucht. Auch wird in diesem Artikel erklärt, wie und in welchem Umfang durch Wiederverwendung von Ergebnissen die Exploration mit NSGA-II in FADSE beschleunigen konnte.

Abbildung 5.6(a) zeigt im Zielraum alle Punkte für die 3319 ausgewerteten Konfigurationen. Die „größte“ Konfiguration hat eine Hardware-Komplexität von ca. 10000.

[7] Die Benchmarks sind: netw-dijkstra, auto-qsort, tele-adpcm-file-decode, offi-stringsearch, cons-jpeg-encode, cons-jpeg-decode, tele-gsm-encode, tele-gsm-decode, secu-rijndael-encode-nounroll, secu-rijndael-decode-nounroll

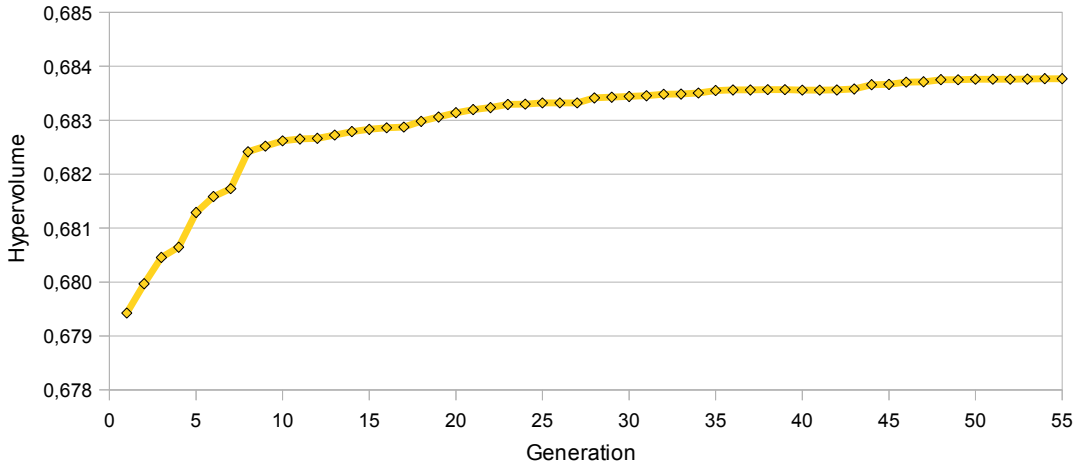


Abbildung 5.5.: Entwicklung des Hypervolume über 55 Generationen bei der DSE der Hardware-Parameter des GAP

Interessant ist aber vor allem der Bereich bis zu einer Hardware-Komplexität von 1000, siehe Abbildung 5.6(b), bzw. der Bereich mit allen nichtdominierten Konfigurationen, die die Pareto-Front approximieren (siehe Abbildung 5.6(c)). Es lassen sich aus diesen Diagrammen die folgenden Schlüsse ziehen:

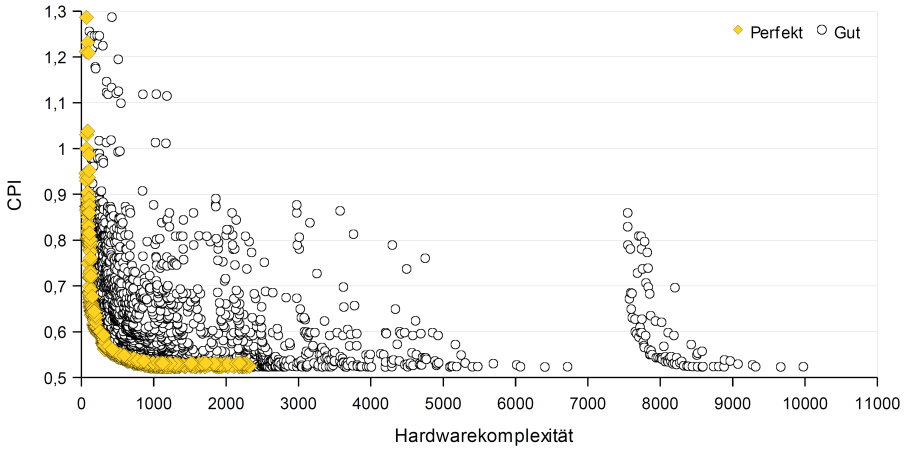
- Für das Performance-Maximum des GAP genügt eine Hardware-Komplexität von ca. 1000. Wird die Komplexität weiter erhöht, so nimmt die Performance nicht weiter zu.

- Nimmt man Performance-Einbußen von 10% in Kauf, so kann die Hardware-Komplexität um 66% reduziert werden, also auf ca. 330. Für eine um 20% reduzierte Performance genügt eine GAP-Konfiguration mit Hardware-Komplexität 170.

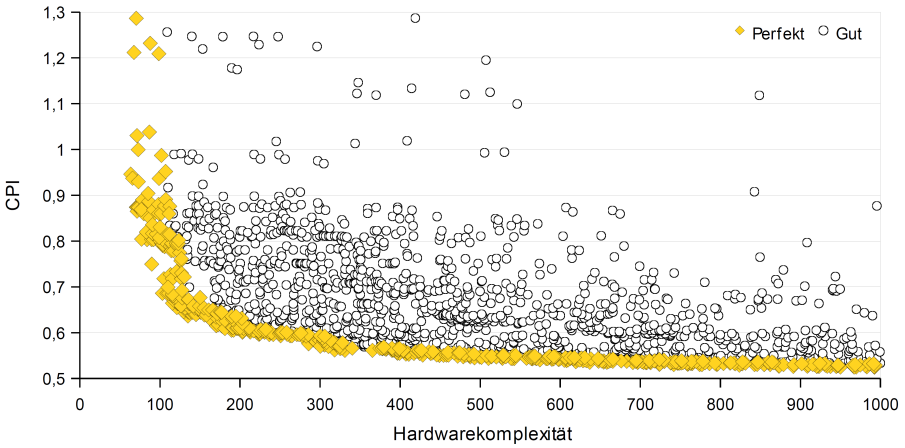
  Somit kann auch mit relativ kleinen Konfigurationen eine unproportional hohe Performance erreicht werden. Erst bei weiterer Reduktion des Hardware-Aufwandes bricht die Performance ein.

- Die mit FADSE automatisch ermittelten Konfigurationen sind besser einzustufen als die manuell gefundenen Konfigurationen, siehe [80]. Mit den automatisch gefundenen Konfigurationen kann bei gleichem Hardware-Aufwand meist eine wesentlich bessere Performance erzielt werden.

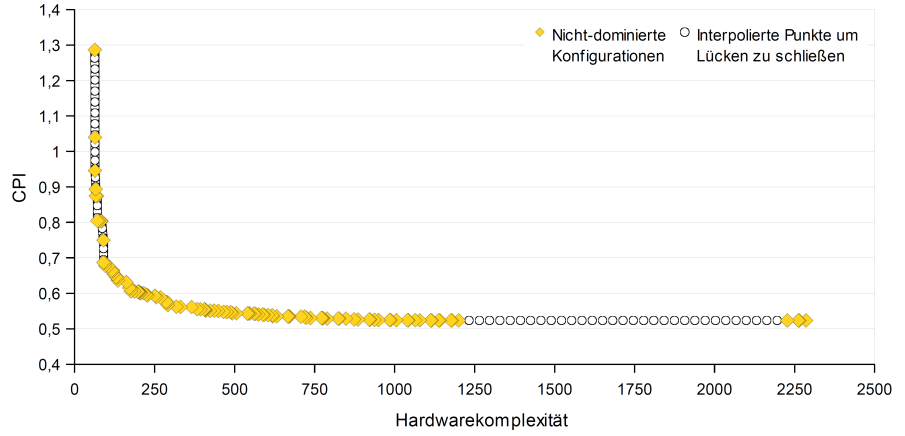
Die Pareto-Approximation zeigt, welche maximale Performance der GAP für die gewählten Benchmarks erreichen kann und wie die Performance in Zusammenhang zur eingesetzten Hardware steht. Allerdings lässt sich daraus noch nicht folgern, wie der GAP konfiguriert werden sollte, damit die eingesetzte Hardware mit hoher Wahrscheinlichkeit effektiv verwendet wird.



(a) Zielraum mit allen in der DSE ausgewerteten Konfigurationen. Das Artefakt rechts zeigt Konfigurationen mit gigantischem Befehlszwischenspeicher. Mit ihm kann hohe Performance erreicht werden, jedoch sind die Konfigurationen nicht effizient.



(b) Ausschnitt des Zielraums mit dem Bereich, in dem die Performance durch Erhöhung der Komplexität gesteigert werden kann



(c) Nichtdominierte Konfigurationen, also die Approximation der Pareto-Front; Lücken werden mit interpolierten Punkten geschlossen um die Abstandsberechnung zu vereinfachen

Abbildung 5.6.: Zielraum der DSE für die Hardware-Parameter des GAP

Dazu werden im Folgenden die von FADSE ausgewerteten Konfigurationen genauer untersucht. Wie in den Abbildungen 5.6(a) und 5.6(b) gezeigt werden alle Konfigurationen in die beiden Klassen *Perfekt* und *Gut* unterteilt. Als perfekt werden diejenigen Konfigurationen bezeichnet, die maximal ε von der angenäherten Pareto-Front (siehe Abbildung 5.6(c)) entfernt sind. Die Variable ε wird so gewählt, dass von den insgesamt 3209 ausgewerteten Konfigurationen ca. 24,5% als perfekt klassifiziert werden[8]. Diese Punkte zeichnen sich dadurch aus, dass sie für die eingesetzte Hardware eine (beinahe) optimale Performance erreichen, also die Hardware möglichst effektiv nutzen und somit effizient sind.

Die Abbildungen 5.7(a)–(c) zeigen, welche Komplexität die Funktionseinheiten, Konfigurationsebenen und Befehlszwischenspeicher besitzen. Teilt man die Gesamtkomplexität auf die drei erwähnten Gebiete auf, so sollte ca. die Hälfte der Hardware für die Konfigurationseinheiten verwendet werden. Anschließend wird möglichst viel Komplexität für die Konfigurationsebenen eingesetzt und der Rest für den Befehlszwischenspeicher. Meist ist seine Größe nicht größer als 512 kB (siehe Abbildung 5.7(d) und Tabelle 5.4).

In den Abbildungen 5.8(a)–(d) werden die einzelnen Parameter des Arrays genauer untersucht. Einfach zu erkennen ist, dass bereits ab einer geringen Hardware-Komplexität von ca. 200 eine hohe Anzahl an Konfigurationsebenen ($2^5 = 32$ bzw. $2^6 = 64$) verwendet wird. Die Anzahl an Zeilen wird für die effektive Nutzung der Hardware fast linear bis auf den Maximalwert erhöht. Überraschend ist, dass die Anzahl an Spalten mit Werten zwischen 4 und 15 relativ niedrig gewählt wird. Das Diagramm 5.8(d) zeigt, dass das optimale Verhältnis zwischen Spalten und Zeilen bei sehr guten Konfigurationen sehr stabil bei ca. zwei Zeilen pro Spalte ist.

5.2.6. Zusammenfassung

Aufgrund des starken Zusammenhangs zwischen den Hardware-Parametern des GAP und der Performance, die damit erreicht werden kann, wurde in diesem Abschnitt eine automatische Suche im Parameterraum (Design Space Exploration, DSE) durchgeführt. Dabei sind die Performance gemessen durch *Clocks per Instruction* (CPI) und die Hardware-Komplexität (siehe Abschnitt 5.2.4) die zu minimierenden Zielfunktionen.

In der ADSE sind 3209 von $1.227.744 \approx 1,2 * 10^6$ möglichen Konfigurationen für jeweils zehn Benchmarks ausgewertet worden, das sind nur 0,29% aller Konfigurationen. Dennoch kann der genetische Algorithmus NSGA-II keinen weiteren Fortschritt mehr erzielen. Da er Parameter zufällig variiert und so weiterhin immer neue Konfigurationen untersucht, ist das Stagnieren des Hypervolume so zu deuten, dass eine sehr gute Approximation der Pareto-Front gefunden wurde.

Anhand der Approximation der Pareto-Front kann man erkennen, dass bereits ab einer Hardware-Komplexität von 1000 die Performance des Prozessors nicht mehr zu-

[8] Details zum Abstandsmaß, also wie der Abstand zwischen einer Konfiguration und der Pareto-Approximation bestimmt wird, enthält Artikel [76].

nimmt. Reduziert man die Hardware-Komplexität auf ca. 330, so reduziert sich die Performance des GAP nur um ca. 10%. Erst ab einer Hardware-Komplexität von ca. 100 bricht die Performance ein.

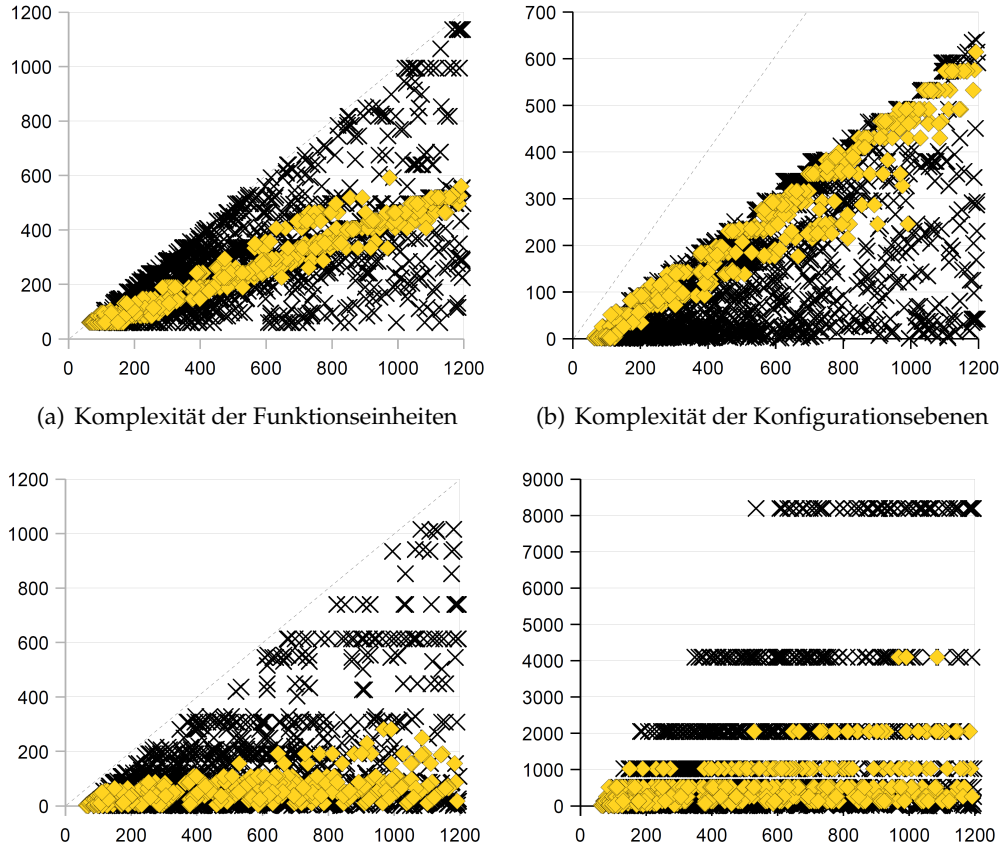
Eine Untersuchung der sehr guten Konfigurationen, die mit FADSE gefunden wurden, zeigt Zusammenhänge zwischen Hardware-Komplexität und der Wahl der Parameter. Die Anzahl der Zeilen des Arrays wird linear zur Komplexität beginnend bei fünf Zeilen für geringe Komplexitäten bis zu 32 Zeilen bei einer Hardware-Komplexität von 1000 erhöht. Das Array sollte ca. doppelt so viele Zeilen wie Spalten besitzen. Die Anzahl der Konfigurationsebenen wird ab einer Komplexität von 200 maximal gewählt.

Sehr gute Konfigurationen setzen ca. 50% der Gesamtkomplexität für Funktionseinheiten ein, möglichst viel für die Konfigurationsebenen und der Befehlszwischenspeicher wird für den Rest verwendet (normalerweise 64 kB bis 512 kB). Somit sind die Konfigurationsebenen wesentlich wichtiger für die Performance als der Befehlszwischenspeicher.

Die Konfigurationen in Tabelle 5.4 sind sehr gut geeignet, um mit möglichst geringem Hardware-Aufwand möglichst hohe Performance zu erzielen.

Zeilen	Spalten	Ebenen	ICache	Komplexität	CPI
8	4	2	8 kB	62,78	1,2863
8	4	2	16 kB	63,54	1,0395
8	4	2	64 kB	66,04	0,8746
8	4	8	64 kB	70,84	0,8046
8	4	32	64 kB	90,04	0,6872
8	7	32	64 kB	129,46	0,6457
8	7	64	256 kB	186,11	0,6051
18	7	32	64 kB	285,66	0,5758
18	7	64	128 kB	383,22	0,5555
18	9	64	128 kB	465,34	0,5478
23	9	64	128 kB	591,84	0,5394
23	12	64	256 kB	776,33	0,5293
32	11	64	512 kB	1005,47	0,5243
28	15	64	512 kB	1141,47	0,5237

Tabelle 5.4.: Effiziente Konfigurationen für den GAP



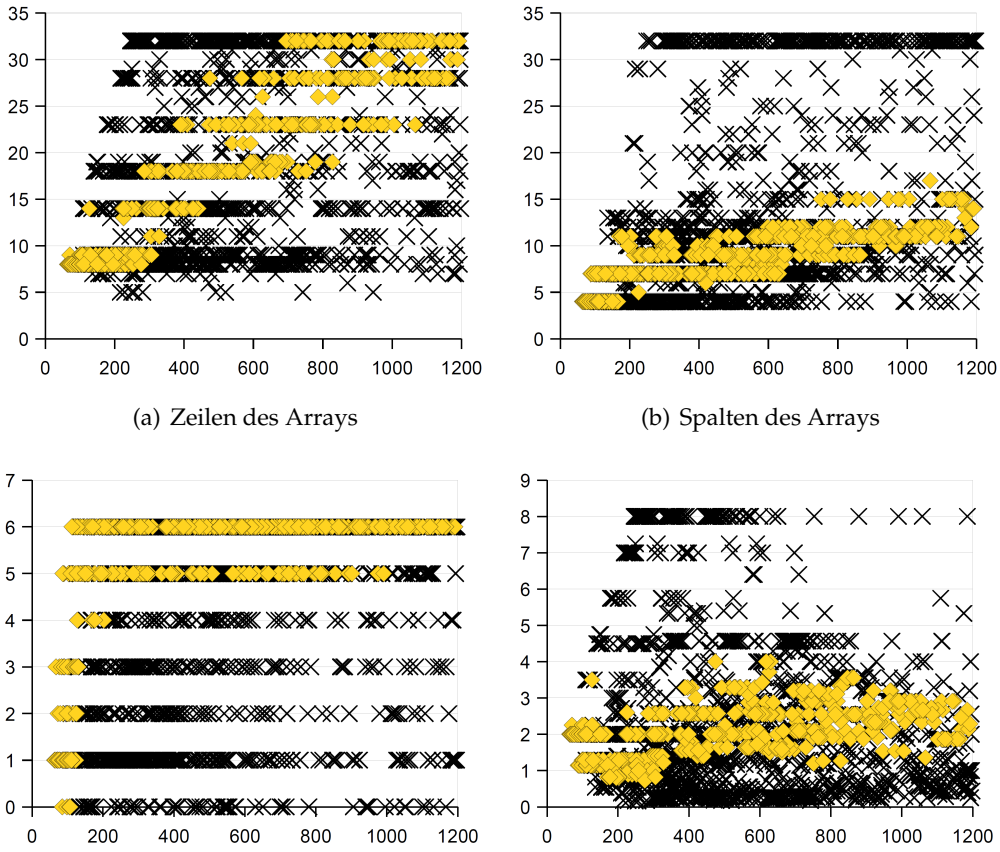
(a) Komplexität der Funktionseinheiten

(b) Komplexität der Konfigurationsebenen

(c) Komplexität des Befehlszwischenspeichers

(d) Größe des Befehlszwischenspeichers (kb)

Abbildung 5.7.: Abbildungen (a)–(c): Aufteilung der Gesamtkomplexität auf die unterschiedlichen Komponenten des GAP. Auf der X-Achse ist die Gesamtkomplexität der Konfigurationen, auf der Y-Achse ist die für die jeweilige Komponente verwendete Komplexität angetragen. Abbildung (d) zeigt die Größe des Befehlszwischenspeichers in Relation zur Hardware-Komplexität. Grau/gelb mit $\Diamond$ eingezeichnet sind perfekte Konfigurationen, mit einem $\times$ markiert sind alle anderen Konfigurationen.



(a) Zeilen des Arrays

(b) Spalten des Arrays

(c) Konfigurationsebenen (2^n)

(d) Zeilen pro Spalte

Abbildung 5.8.: Parameter des Arrays bei verschiedenen Hardware-Komplexitäten (X-Achse); sehr gute Konfigurationen sind in grau/gelb als ◊ eingezeichnet, sonstige Konfiguration mit einem × versehen.

5.3. Optimale Code-Optimierungs- und Hardware-Parameter

Im Artikel [80] und Kapitel 4 wurde gezeigt, dass FADSE trotz Fokus auf Hardware-Parameter sehr gut verwendet werden kann, um optimale Parameter für Code-Optimierungen zu finden. Um die maximale Performance mit dem GAP zu erreichen, wird in diesem Abschnitt parallel nach (a) optimalen Hardware-Konfigurationen, (b) optimalen Kombinationen von vier Code-Optimierungen (siehe Kapitel 4.2 bis 4.5) und (c) optimalen Parametern für diese Code-Optimierungen gesucht.

Zwar ergibt sich aus der gleichzeitigen Untersuchung von Hardware- und Code-Optimierungs-Parametern ein sehr großer Parameterraum, doch erst die Hardware-Parameter und danach die Code-Optimierungen zu optimieren würde nicht zu ähnlich guten Ergebnissen führen. Der Grund dafür ist, dass durch die Code-Optimierungen sich die auszuführenden Programme so verändern können, dass dann die zuvor optimal gewählten Hardware-Parameter nicht mehr die bestmögliche Konfiguration ergeben [1].

Der durchsuchte Parameterraum und die Optimierungsziele werden in Abschnitt 5.3.1 erörtert. Aufgrund seiner Größe und da viele Konfigurationen zu denselben Ergebnissen führen, werden Methoden eingeführt, mit denen die ADSE beschleunigt werden kann (Abschnitt 5.3.2). Die Ergebnisse werden in Abschnitt 5.3.3 ausgewertet. Verwandte Arbeiten stellt Abschnitt 5.3.4 vor; Abschnitt 5.3.5 fasst die Erkenntnisse zusammen.

5.3.1. Parameterraum und Optimierungsziele

Parameterraum

Der Parameterraum umfasst die Hardware-Parameter des GAP (Array und Befehlszwischenspeicher), Parameter zur Aktivierung von Code-Optimierungen sowie die relevanten Parameter für die einzelnen Code-Optimierungen. Als Code-Optimierungen werden die bedingte Ausführung (siehe Kapitel 4.2), die Funktionseinbindung (siehe Kapitel 4.3), die statische Spekulation (siehe Kapitel 4.4) und die verbesserte Ersetzungsstrategie QdLRU für die Ebenen des GAP (siehe Kapitel 4.5) eingesetzt. Alle Optimierungen können automatisch mit GAPtimize auf Programme angewandt werden.

Werden mehrere Code-Optimierungen verwendet, so stellt sich die Frage, in welcher Reihenfolge das geschehen soll, um möglichst gute Ergebnisse zu erzielen. Aufgrund der hohen Anzahl an Möglichkeiten ($n!$ für n Optimierungen[9]) ist die Evaluation aller Möglichkeiten normalerweise keine angemessene Lösung (siehe z. B. [157], [88]).

Allerdings sind bei den untersuchten vier Code-Optimierungen nur die statische Spekulation und das Einbinden von Funktionen in der Reihenfolge austauschbar, die beiden anderen Optimierungen müssen immer am Ende der Kette ausgeführt werden und beeinflussen sich dabei nicht gegenseitig. Deshalb wird zum Parameterraum ein

[9] Ohne wiederholte Anwendung von Optimierungen und Nebenbedingungen

weiterer Parameter hinzugefügt, mit dem die Reihenfolge der Funktionseinbindung und der statischen Spekulation festgelegt werden kann (`do_finlining_first`).

Tabelle 5.5 zeigt den vollständigen Parameterraum. Von den 23 Parametern sind 8 Parameter auf feste Werte gesetzt. Es gibt insgesamt $4872 * 1 * 2 * 2 * (101 * 101 + 1) * (101 * 101 + 1) * (100 * 101 * 1 * 1 * 51 * 1 + 1) * 2 \approx 2,1 * 10^{18}$ Konfigurationen. Für jede Konfiguration werden neun Benchmarks[10] berechnet, dann werden über die Ergebnisse die Mittelwerte ermittelt. Wäre es möglich, jede Konfiguration in einer Sekunde auszuwerten, so wäre eine vollständige Auswertung erst nach ca. 33 Millionen Jahren abgeschlossen. Also ist ein heuristisches Verfahren hier der einzig sinnvolle Weg, gute Konfigurationen zu finden.

Der Parameterraum für den GAP wird mit 32 statt 64 Konfigurationsebenen und einem vergleichsweise kleinen Befehlszwischenspeicher etwas eingeschränkt. Grund dafür ist, dass mit einer hohen Anzahl an Konfigurationsebenen oder einem großen Befehlszwischenspeicher die Auswirkungen der untersuchten Code-Optimierungen sehr klein sind, da sowohl eine hohe Anzahl an Ebenen wie auch ein großer Cache dafür sorgen, dass die eigentlich adressierten Probleme (siehe Kapitel 2.3.3) beinahe verschwinden.

Optimierungsziele

Wie schon in Abschnitt 5.2 werden Konfigurationen gesucht, die die zur Verfügung stehenden Hardware-Ressourcen möglichst effektiv verwenden. Deshalb werden wieder die Hardware-Komplexität der Konfigurationen und die damit erzielte Performance gemessen. Da durch die Code-Optimierungen die Anzahl der ausgeführten Instruktionen verändert werden kann, wird die Performance mit *Clocks per reference instruction* (CPRI, siehe Kapitel 4.3.3) gemessen. Ziel ist es, mit möglichst niedriger Hardware-Komplexität möglichst hohe Performance (also einen niedrigen CPRI-Wert) zu erreichen.

5.3.2. Methoden zur Beschleunigung der DSE

Aufgrund der immensen Größe des Parameterraumes wurde FADSE um einige Methoden erweitert, mit denen die Auswertung von Konfigurationen beschleunigt werden kann. Zur Auswertung einer Konfiguration $(p_0, ..., p_n)$ muss für alle Benchmarks, die untersucht werden sollen, die Konfiguration mit dem Simulator ausgeführt werden. Die Konfiguration wird also an den Benchmark gekoppelt:

$$P^i = (p_0, p_1, p_2, p_3, p_4, p_5, ..., p_n, b_i), \text{ wobei } b_i \text{ ein Benchmark ist}$$

Wie bereits erwähnt kann FADSE eine Datenbank verwenden, um Ergebnisse für die spätere Verwendung zu speichern. Wird P^i in der Datenbank gefunden, d. h. es

[10] Die neun Benchmarks sind: netw-dijkstra, auto-qsort, tele-adpcm-file-decode, offi-stringsearch, cons-jpeg-encode, tele-gsm-encode, tele-gsm-decode, secu-rijndael-encode-nounroll, secu-rijndael-decode-nounroll

Bezeichnung	Min	Max	Werte
Array			**4872**
n_lines	4	32	29
n_columns	4	31	28
n_layers (2^n)	1	32	6
Befehlszwischenspeicher			**1**
c_chunk	8	8	1
c_sets	128	128	1
c_lines	1	1	1
Sprungvorhersage			**2**
do_branch_prediction	0	1	2
Optimierte Ersetzung mit QdLRU			**2**
do_qdlru	0	1	2
Bedingte Ausführung			**10202**
do_pex	0	1	2
pex_max_total	0	100	101
pex_max_loop	0	100	101
Funktionseinbindung			**10202**
do_finlining	0	1	2
finline_length_of_function (Schritt 25)	10000	10000	1
finline_max_caller_count	100	100	1
finline_kpi_insns_per_caller	0	100	101
finline_weight_of_caller	0	100	101
Statische Spekulation			**515101**
do_static_speculation	0	1	2
sspec_min_node_weight (Schritt 10)	10	1000	100
sspec_min_probability_percent	0	100	101
sspec_hot_function_ratio_percent	80	80	1
sspec_max_number_of_insns_to_shift	250	250	1
sspec_max_number_of_blocks_to_shift	0	50	51
sspec_max_allowed_additional_height	0	0	1
Reihenfolge			**2**
do_finlining_first	0	1	2
Gesamt			**$2{,}090 * 10^{18}$**

Tabelle 5.5.: Parameterraum für die parallele Optimierung von Hardware und Code-Optimierungen mit GAP und GAPtimize

gibt in der Datenbank einen Datensatz, bei dem alle Parameter und der Benchmark mit dem gesuchten übereinstimmen, so kann dieser verwendet werden. Es müssen dann weder GAPtimize noch der GAP-Simulator ausgeführt werden.

Dieser Ansatz kann noch stark erweitert werden, da ja verschiedene Code-Optimierungen mit Flags deaktiviert werden können – in diesem Fall haben die Parameter der deaktivierten Code-Optimierung keinen Einfluss mehr. Für die beiden folgenden Konfigurationen P_1^i und P_2^i berechnet GAPtimize, wenn (p_2, p_3, p_4) eine Code-Optimierung steuern, die mit p_1 deaktiviert werden kann, dasselbe Programm:

$$P_1^i = (p_0, \mathbf{0}, \mathbf{12}, \mathbf{10}, \mathbf{42}, p_5, ..., p_n, b_i)$$
$$P_2^i = (p_0, \mathbf{0}, \mathbf{5}, \mathbf{10}, \mathbf{18}, p_5, ..., p_n, b_i)$$

Für FADSE können in der Konfigurationsdatei, in der der Parameterraum und die Zielfunktionen festgelegt werden, mehrere Relationen definiert werden, mit denen Parameter abhängig von anderen Parametern deaktiviert werden können. Sie werden dann bei der Datenbankabfrage nicht mehr mit Werten belegt, sondern es wird `N/A` als Wert verwendet. Eine Relation sieht beispielsweise wie folgt aus:

```
<relation>
    <if parameter="do_pex" value="0">
        <then_invalidate>
            <parameter name="pex_max_total" />
            <parameter name="pex_max_loop" />
        </then_invalidate>
    </if>
</relation>
```

Eine weitere Beschleunigung kann aus der Tatsache gewonnen werden, dass GAPtimize selbst für zwei vollständig unterschiedliche Konfigurationen (beispielsweise P_3^i und P_4^i) nicht zwingend unterschiedliche Programme berechnet. Der Grund dafür ist, dass manche Parameter eine Optimierung beispielsweise so einschränken, dass sie nicht mehr verwendet wird, wodurch dann alle anderen Parameter ebenso wirkungslos werden.

Deshalb wird, sobald der GAP-Simulator ausgeführt wurde, das Verzeichnis mit seinen Ergebnissen in ein Archiv kopiert. Als Dateiname wird eine Prüfsumme verwendet, die über das ausgeführte Programm, eventuelle Eingabedaten und die ausgeführte Simulatorkonfiguration berechnet wird. So kann für andere Konfigurationen nach Ausführung von GAPtimize erst überprüft werden, ob dasselbe Programm schon einmal mit der gewünschten Simulator-Konfiguration ausgeführt wurde. Ist das der Fall, dann wird das Archiv in das Zielverzeichnis entpackt, andernfalls der Simulator gestartet.

Mit diesen drei Mechanismen kann verhindert werden, dass eine Konfiguration doppelt berechnet wird.

Als weitere Methode zur Verkürzung der benötigten Zeit ist *Response surface modelling* [37, 108] zu nennen, das in FADSE derzeit nicht implementiert ist. Auch in den DSE-Algorithmus integrierte *künstliche neuronale Netze* [110, 109, 107] können dazu

verwendet werden, nur mit hoher Wahrscheinlichkeit zielführende Konfigurationen zu untersuchen.

5.3.3. Ergebnisse und Auswertung

Für die Evaluierung wird FADSE mit SMPSO [119] als heuristischer Algorithmus verwendet. FADSE wurde um die drei genannten Methoden zur Beschleunigung erweitert und die Anbindung an GAPtimize dahingehend modifiziert, dass mehrere Code-Optimierungen nacheinander angewandt werden können. Zwischen einzelnen Optimierungsschritten wird, wenn nötig, das modifizierte Programm auf dem GAP-Simulator ausgeführt, sodass dann aktuelle Traces für die erneute Ausführung von GAPtimize vorhanden sind.

Performancegewinn und Komplexitätsreduzierung durch Code-Optimierungen

Die ADSE wird nach 20 Generationen à 100 Individuen abgebrochen, da dann das Hypervolume stagniert. Es werden 1682 unterschiedliche Konfigurationen ausgewertet, das entspricht einem verschwindend geringen Anteil aller Konfigurationen. Als Vergleichswert wird dasselbe Szenario aber ohne Code-Optimierungen verwendet. Abbildung 5.9 zeigt die beiden gefundenen Pareto-Approximationen[11].

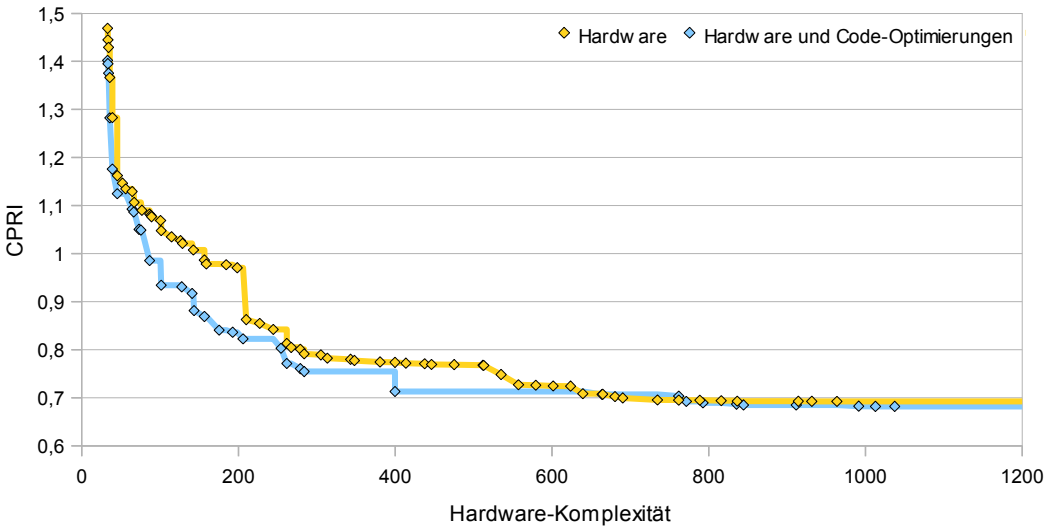


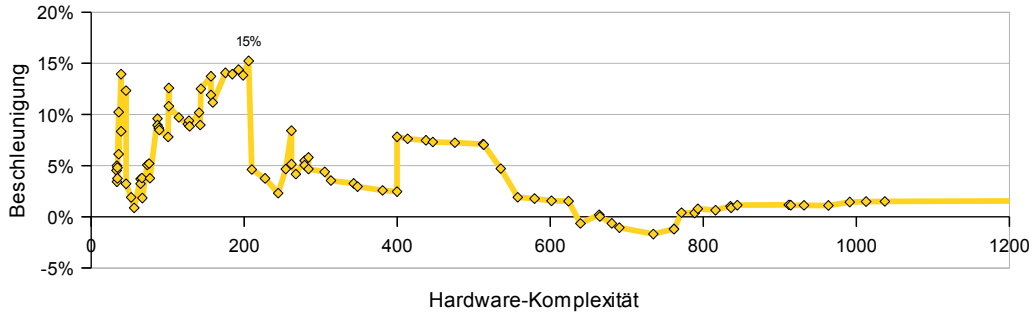
Abbildung 5.9.: Pareto-Approximation mit und ohne Code-Optimierungen

Auf den ersten Blick liegen die beiden Kurven sehr nahe beieinander. Besonders ab einer Hardware-Komplexität von ca. 600 ist nahezu kein Unterschied mehr be-

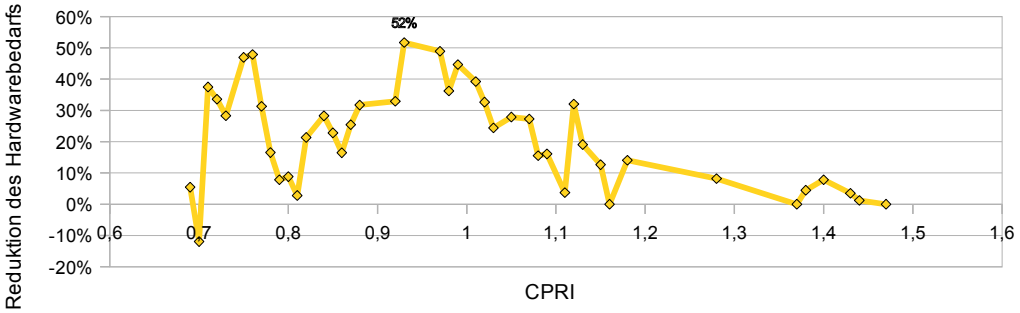
[11] Die gezeigten Kurven haben Stufen, da die verwendete Interpolation pessimistisch ist: Ist in einem Bereich kein Datenpunkt vorhanden, so wird angenommen, dass die erreichbare Performance in diesem Bereich höchstens so gut ist wie die des nächsten Datenpunktes mit geringerer Hardware-Komplexität. (Alternative: Lineare Interpolation)

merkbar. Ursache dafür ist, dass der Einfluss aller Code-Optimierungen abnimmt, wenn mehr Hardware-Ressourcen verfügbar sind, d. h. bei größerem Array und höherer Anzahl an Konfigurationsebenen sinkt beispielsweise der Einfluss der Funktionseinbindung. Außerdem wird immer die durchschnittliche Performance über neun Benchmarks berechnet. Das kann dazu führen, dass wenn für einen Benchmark sehr gute Performance erreicht wird, das durch nur geringe Performance-Einbußen für alle anderen Benchmarks wieder ausgeglichen wird. Für geringere Hardware-Komplexitäten haben die Code-Optimierungen einen wesentlich höheren Einfluss.

Die Erfüllung der Ziele (a) Steigerung der Performance bei gleicher Hardware-Komplexität und (b) Reduktion des Hardware-Bedarfs bei gleicher Performance zeigen die Abbildungen 5.10(a) und (b).



(a) Performance-Gewinn bei gleicher Hardware-Komplexität



(b) Reduktion des Hardwarebedarfs bei gleicher Performance

Abbildung 5.10.: Auswirkungen der Code-Optimierungen

Die maximale Beschleunigung von 15% wird bei einer Hardware-Komplexität von ca. 200 erreicht, einer sehr kleinen Konfiguration des GAP. Durch die Code-Optimierungen wird hier eine Stufe in der Pareto-Approximation ausgeglichen, die dadurch zustande kommt, dass sich die Ausführung einiger weniger Benchmarks ab einer bestimmten Hardware-Komplexität über die Maßen beschleunigt[12]. Die durchschnittliche Beschleunigung bis zu einer Hardware-Komplexität von 600 beträgt 7,0%.

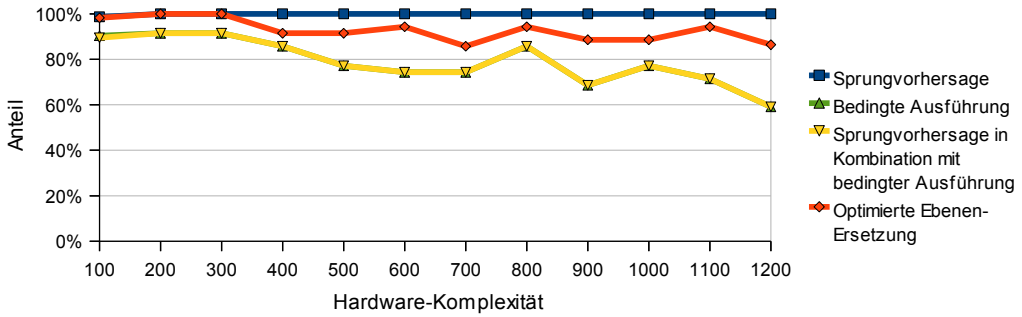
Mit den Code-Optimierungen kann bei gleicher Performance die benötigte Hard-

[12] Auch bei mehrmaliger Ausführung der DSE blieb diese Stufe erhalten.

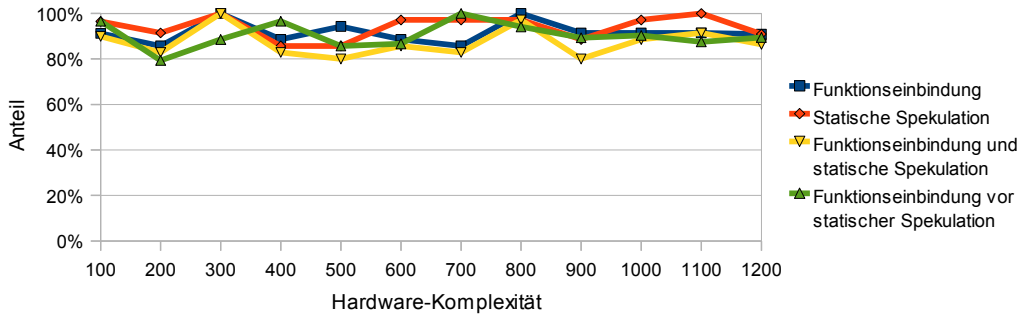
ware-Komplexität um maximal 52% bzw. 21% im Mittel reduziert werden. Die Code-Optimierungen reduzieren also bei gleicher Performance die dafür notwendige Hardware-Komplexität drastisch.

Verwendung der Code-Optimierungen und Wahl der Parameter

Genauso interessant wie die Auswirkungen der Code-Optimierungen ist, welche Optimierungen in den besonders guten Konfigurationen, also denen auf oder sehr nahe bei der Pareto-Approximation, verwendet werden. In Abbildung 5.11(a) wird gezeigt, wie hoch der Anteil der Individuen ist, die bei bestimmten Hardware-Komplexitäten die Sprungvorhersage, die bedingte Ausführung, die bedingte Ausführung in Kombination mit der Sprungvorhersage und die optimierte Ersetzungsstrategie QdLRU verwenden. Selbiges enthält Abbildung 5.11(b) für die Funktionseinbindung, die statische Spekulation, die Kombination beider Verfahren und die Reihenfolge der Anwendung.



(a) Sprungvorhersage, bedingte Ausführung und optimierte Ersetzungsstrategie QdLRU



(b) Funktionseinbindung und statische Spekulation

Abbildung 5.11.: Verwendung der Code-Optimierungen durch GAPtimize bei sehr guten Konfigurationen

Aus den Diagrammen lassen sich die folgenden Fakten ableiten:

- Die Sprungvorhersage ist in jedem Fall aktiviert.
- Die bedingte Ausführung wird nur in Kombination mit der Sprungvorhersage

verwendet[13]. Mit steigender Hardware-Komplexität wird die bedingte Ausführung seltener eingesetzt.

- Die Ersetzungsstrategie QdLRU ist für alle Konfigurationen mit einer Häufigkeit von ca. 90% aktiviert; bei geringer Hardware-Komplexität wird sie bei beinahe allen Konfigurationen verwendet.

- In den meisten guten Konfigurationen werden sowohl die Funktionseinbindung als auch die statische Spekulation verwendet.

- Meist wird die Funktionseinbindung vor der statischen Spekulation durchgeführt. Ursache dafür ist wahrscheinlich, dass sich in den Kopien der Funktionskörper bessere oder weitere Möglichkeiten für die statische Spekulation ergeben.

Man kann schlussfolgern, dass bei den meisten guten Konfigurationen beinahe alle der mit GAPtimize implementierten Code-Optimierungen verwendet werden.

Einen Blick auf die Wahl der einzelnen Parameter erlauben die Grafiken 5.12 und 5.13(a)–(c). Anders als vermutet ist das Verhältnis von Spalten zu Zeilen des Arrays; es steigt anders als in Abschnitt 5.2.5 beschrieben erst auf ca. zwei um dann bald wieder zu fallen. Auch werden mehr Spalten des Arrays verwendet. Die Parameter für die bedingte Ausführung werden mit zunehmender Hardware-Komplexität tendenziell niedriger gewählt, wobei die Standardabweichung, also das Maß für die Streuung der Werte, sehr hoch ist. Dasselbe gilt für die Parameter der Funktionseinbindung. Bei den Parametern für die statische Spekulation sind keine klaren Trends erkennbar.

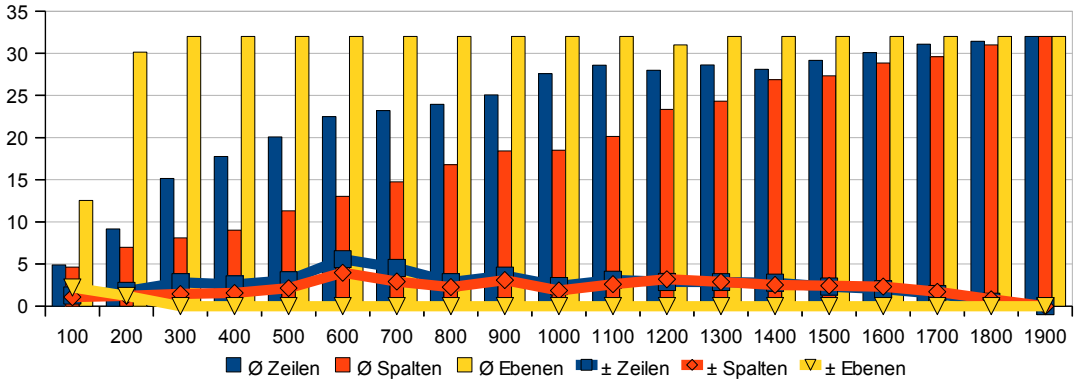
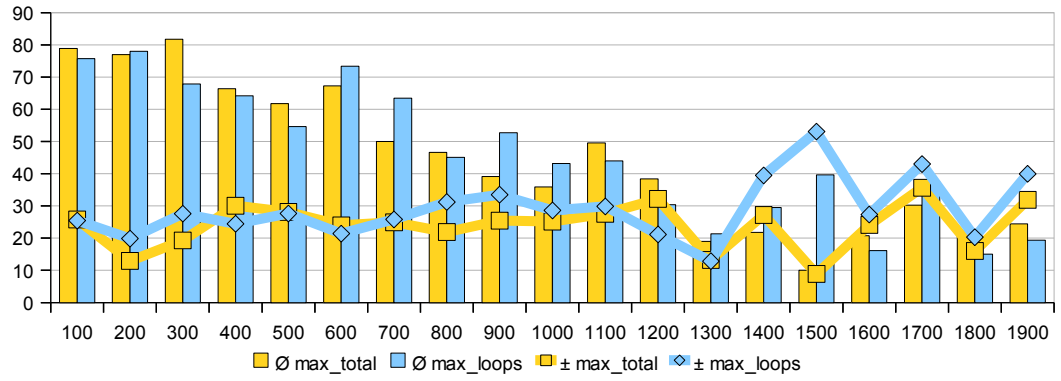
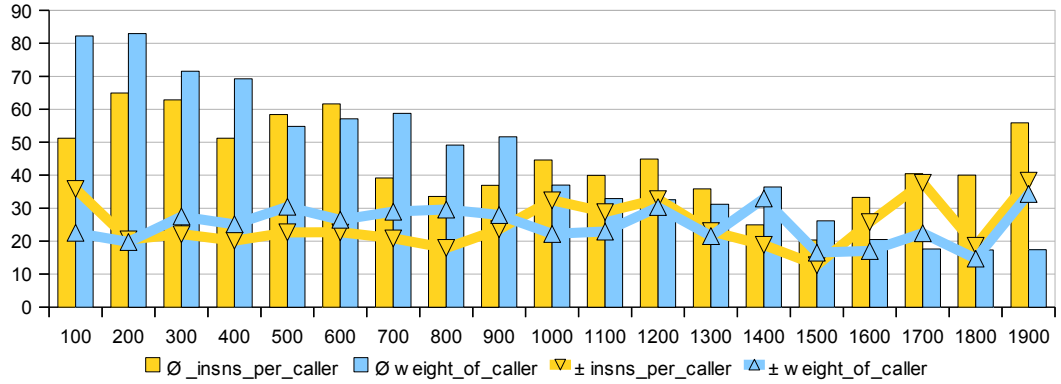


Abbildung 5.12.: Durch FADSE gewählte Array-Parameter für sehr gute Konfigurationen bei gleichzeitiger Optimierung von Hardware- und Software-Parametern (X-Achse: Hardware-Komplexität; es werden Mittelwert ⊘ und Standardabweichung ± dargestellt)

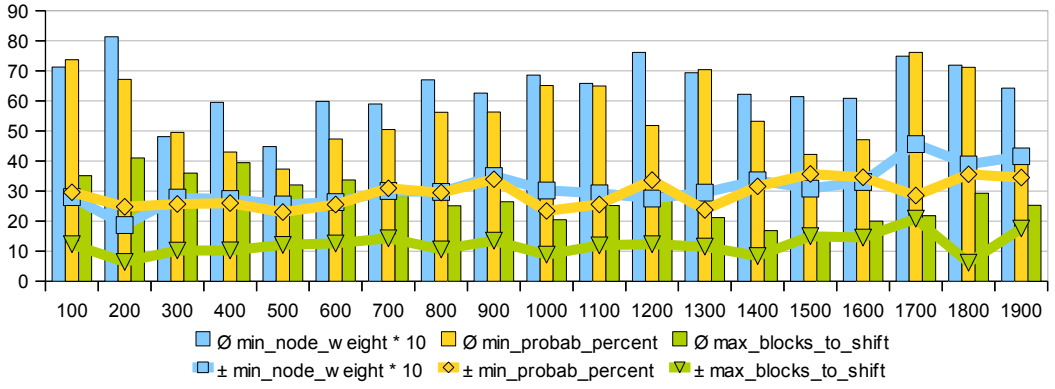
[13]Die beiden Kurven sind im Diagramm deckungsgleich.



(a) Parameter für die bedingte Ausführung



(b) Parameter für die Funktionseinbindung



(c) Parameter für die statische Spekulation

Abbildung 5.13.: Durch FADSE gewählte Parameter für Code-Optimierungen für sehr gute Konfigurationen bei gleichzeitiger Optimierung von Hardware- und Software-Parametern (X-Achse: Hardware-Komplexität; Y-Achse: Mittelwert ⌀ und Standardabweichung ±)

5.3.4. Verwandte Arbeiten

Für den GAP wurden mehrere Code-Optimierungen zusammen mit den Hardware-Parametern noch nie gemeinsam untersucht. Einzig im Artikel [80] werden die Parameter für die Funktionseinbindung (siehe Kapitel 4.3) zusammen mit den Hardware-Parametern untersucht. Die meisten Ähnlichkeiten zeigen Arbeiten, in denen (a) parallel Hardware- und Code-Optimierungs-Parameter gesucht, (b) automatisch Compiler-Optimierungen konfiguriert oder (c) automatisch Compiler-Optimierungen ausgewählt werden.

Agosta et al. [1] suchen parallel nach für eine Gruppe von Benchmarks optimalen Parametern für einen *Soft-Core*, also einen Prozessor auf einem FPGA, und dazu passenden optimalen Programmtransformationen. Es werden mehrere Zielfunktionen verwendet, beispielsweise Energiebedarf und Laufzeit. Außerdem wird gezeigt, dass die Ergebnisse der parallelen Optimierung besser sind, als wenn erst gute Parameter für die Hardware und anschließend für die Software ermittelt werden.

Mit der Optimierung von Compiler-Heuristiken durch Techniken des maschinellen Lernens beschäftigen sich beispielsweise Monsifrot et al. [114]. Sie optimieren das Abrollen von Schleifen im Wesentlichen durch Vorhersage, ob die Optimierung wahrscheinlich zu guten Ergebnissen führt. Stephenson et al. [148] verwenden ebenso maschinelles Lernen.

Im Artikel [72] untersuchen Hoste et al., inwieweit mit einem genetischen Algorithmus (abgeleitet von SPEA2) neue Optimierungslevel in Konkurrenz zu den Schaltern `-O3`, `-Os` etc. eines Standard-Compilers gefunden werden können. Die gefundenen Level zeigen selbst für unbekannte Programme bessere Ergebnisse, sind also übertragbar. Ähnliche Ziele verfolgen Almagor et al.[6], wobei hier pro Programm optimiert wird ($\rightarrow$ adaptiv, siehe Abschnitt 5.4).

Es handelt sich bei der DSE dem Grunde nach um eine iterative Vorgehensweise, da Schritt für Schritt optimiert wird. Auch zu *Iterative Compilation* gibt es einige Veröffentlichungen, z. B. von Knijnenburg et al. [85].

Eine DSE kann ebenso verwendet werden, um den Einfluss von Programm-Optimierungen auf die WCET zu untersuchen und dann die WCET von Programmen zu reduzieren, siehe Artikel [99].

5.3.5. Zusammenfassung

Um untersuchen zu können, welche Auswirkungen die Code-Optimierungen, die in GAPtimize implementiert sind, auf die Ausführung von Programmen auf dem GAP haben, wird eine ADSE mit dem Algorithmus SMPSO durchgeführt. Da der zugrundeliegende Parameterraum sehr groß ist, wurde FADSE erst um zusätzliche Methoden erweitert, die die Auswertung von Individuen beschleunigen. Es wird mit 1682 von ca. $2,1 * 10^{18}$ Konfigurationen nur ein verschwindend geringer Anteil des gesamten Parameterraumes ausgewertet.

Für Konfigurationen mit Hardware-Komplexität unter 600 kann mit den Code-Optimierungen durchgehend eine bessere Performance im Vergleich zur Ausführung ohne vorherige Code-Optimierung erreicht werden. Ab einer Komplexität von 600 ha-

ben die Code-Optimierungen keine nennenswerten Auswirkungen mehr. Die maximale Beschleunigung ist 15%, der Durchschnitt bis zu einer Komplexität von 600 ist 7,0%. Mit den Code-Optimierungen kann die gleiche Performance mit wesentlich geringerer Hardware-Komplexität erreicht werden; sie kann maximal um 52% und im Durchschnitt um 21% reduziert werden.

Eine genaue Analyse der Verwendung der Code-Optimierungen bei sehr guten Konfigurationen zeigt, dass die Sprungvorhersagetechnik des GAP immer sinnvoll ist. Die bedingte Ausführung sollte nur in Kombination mit der Sprungvorhersage eingesetzt werden und in der DSE wird sie mit wachsender Komplexität seltener verwendet. Die Funktionseinbindung und die statische Spekulation werden sehr häufig gewählt, wobei immer mindestens eine der Optimierungen aktiv ist. Werden beide Optimierungen angewandt, so sollte die Funktionseinbindung vor der statischen Spekulation durchgeführt werden. Eine Veränderung mit steigender Komplexität ist nicht erkennbar. Die Technik QdLRU zur optimierten Ersetzung ist für alle Konfigurationsgrößen gewinnbringend.

5.4. Adaptive Optimierung der Hardware- und Code-Optimierungs-Parameter

Die wesentlichen Einflussfaktoren auf die Performance des GAP (und aller anderen Prozessoren) sind die verwendete Hardware-Konfiguration, das auszuführende Programm und letztendlich seine Eingabedaten (vgl. Definition 1, Seite 30). In den Abschnitten 5.2 und 5.3 werden mit einer DSE optimale Parameter erst für die Hardware-Parameter des GAP allein und dann in Verbindung mit Code-Optimierungen gesucht, wobei immer mehrere Benchmarks zur Bewertung einer Konfiguration verwendet werden. Die damit erzielten Ergebnisse lassen sich (wahrscheinlich mit Einschränkungen) auf Programme übertragen, die nicht zur getesteten Benchmark-Auswahl zählen.

Optimiert man nur ein einzelnes Programm[14], so können sowohl die Hardware- als auch die Parameter für die Code-Optimierungen so gewählt werden, dass sie optimal für das Zielprogramm sind. Mit dem Verzicht auf die Übertragbarkeit der Ergebnisse gewinnt man weitere Chancen. Es werden die Parameter so gewählt, dass sie unter Berücksichtigung aller Umgebungswerte optimal sind; somit werden sie *adaptiv* entsprechend den Definitionen im Artikel [71] gewählt.

5.4.1. DSE als Werkzeug für eine adaptive Optimierung

Um optimale Werte für die Hardware- und Code-Optimierungs-Parameter zu bestimmen, kann – wie schon in Abschnitt 5.3 vorgestellt – eine automatische DSE durchgeführt werden, z. B. mit FADSE und den Algorithmen NSGA-II oder SMPSO. Der einzige Unterschied ist, dass statt für neun Benchmarks nur noch für einen einzelnen optimiert wird. Alternative Ansätze zu einer DSE werden in Abschnitt 5.4.3 erwähnt.

In der Praxis müsste für jeden Benchmark, der optimiert werden soll, die DSE neu begonnen werden, was sehr viel Zeit in Anspruch nimmt. Wie beispielsweise in Grafik 5.6(a) (Seite 131) zu sehen, werden bei einer DSE (besonders am Anfang) viele Punkte ausgewertet, die weit von der Pareto-Kurve entfernt sind. Um die Zeit zu reduzieren, die nötig ist, um die Parameter adaptiv für einen Benchmark zu finden, ist es also wichtig, die DSE stark zu beschleunigen.

Geht man davon aus, dass seitens eines Herstellers oder einer sonstigen unterstützenden Partei Interesse daran bestünde, dass adaptive Optimierung verwendet wird, so könnten sie mit den beiden folgenden Verfahren und entsprechenden Vorarbeiten helfen, die Zeit für die Optimierung stark zu verkürzen:

- **Start mit besten Individuen aus dem allgemeinen Fall**
  Theoretisch sollten die Ergebnisse für einen einzelnen Benchmark schon relativ gut sein, wenn die Konfigurationen auf und nahe der Pareto-Approximation, die bei der Optimierung nach dem Mittelwert gefunden wurden, verwendet werden. FADSE besitzt für NSGA-II die Möglichkeit, eine DSE zu starten, indem als initiale Population (vgl. Abbildung 5.2, Seite 120) Individuen aus einer Textdatei

[14] Alternatives Szenario: Optimierung für eine bestimmte Domäne.

gelesen werden. Startet man die adaptive Optimierung mit der letzten Generation aus der allgemeinen Optimierung, so ist schon eine gute Basis vorhanden, die dann weiter optimiert werden kann.

- **Beeinflussung des Mutationsoperators mit Transformationsregeln**
  Aus der Klassifikation der in einer Referenz-DSE ausgewerteten Individuen in gute und perfekte Individuen lässt sich ein automatisch erstellter Entscheidungsbaum ableiten. Von diesem Baum können Regeln abgeleitet werden, mit denen die Parameter von Individuen so verändert werden können, dass sie wahrscheinlich näher an die Pareto-Front geschoben werden.

  Dieses Vorgehen, das eigentlich einen erweiterten Mutationsoperator bereitstellt (vgl. Abbildung 5.2 auf Seite 120 für NSGA-II) und mit dem sich mit FADSE sowohl SMPSO als auch NSGA-II beschleunigen lassen, wird im Artikel [76] näher erklärt.

  Die adaptive Optimierung kann somit durch Bereitstellung eines Regelsystems für den allgemeinen Fall, also basierend auf die Optimierung mehrerer Benchmarks, verkürzt werden.

Natürlich helfen auch die bereits im Abschnitt 5.3.2 vorgestellten Methoden zur Beschleunigung der DSE weiterhin dabei, die Zeit für die Parametersuche zu reduzieren.

5.4.2. Ergebnisse und Auswertung

Um den Einfluss der adaptiven Optimierung zu bestimmen wird für jeden der neun im vorherigen Abschnitt 5.3 verwendeten Benchmarks separat eine Optimierung der Hardware- und Code-Optimierungs-Parameter durchgeführt. Der Parameterraum ist derselbe wie für den allgemeinen Fall (siehe Tabelle 5.5) und die Zielfunktionen bleiben gleich.

Anschließend werden für alle Benchmarks die Pareto-Approximationen errechnet und Lücken mit der schon erwähnten pessimistischen Abschätzung aufgefüllt. Dann wird für alle neun Benchmarks der Mittelwert errechnet. Das Ergebnis ist in Abbildung 5.14 zu sehen. Wie erwartet sind die Ergebnisse mit Adaptivität für alle Hardware-Komplexitäten besser als bei Optimierung nach dem Mittelwert. Der Einfluss der Code-Optimierungen ist auch bei adaptiver Anwendung der Code-Optimierungen für geringe Hardware-Komplexitäten bis ca. 600 höher, danach bleibt die Pareto-Kurve aber konstant unter der Kurve ohne Code-Optimierungen.

Wie in Abbildung 5.15(a) zu sehen ist beträgt der maximale Performance-Gewinn bei adaptiven Optimierungen 29% bei einem Mittelwert von 17% bis zu einer Komplexität von 600 (14% über alle Werte). Die durchschnittliche Beschleunigung ist also mehr als doppelt so hoch wie bei Optimierung nach dem Mittelwert.

Der Hardware-Bedarf, der nötig ist, um eine vorgegebene Performance zu erreichen, kann mit den adaptiven Code-Optimierungen um maximal 65% verringert werden, im Mittel um 42%. Somit ist die Reduktion des Hardware-Bedarfs durch die adaptive Optimierung der Code-Optimierungs-Parameter ungefähr doppelt so hoch (vgl. Abbildung 5.15(b)).

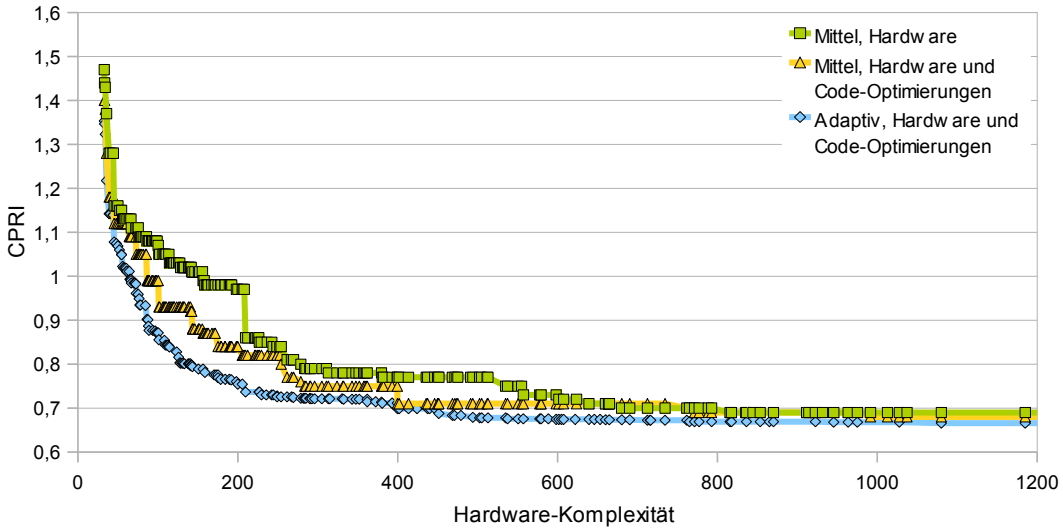
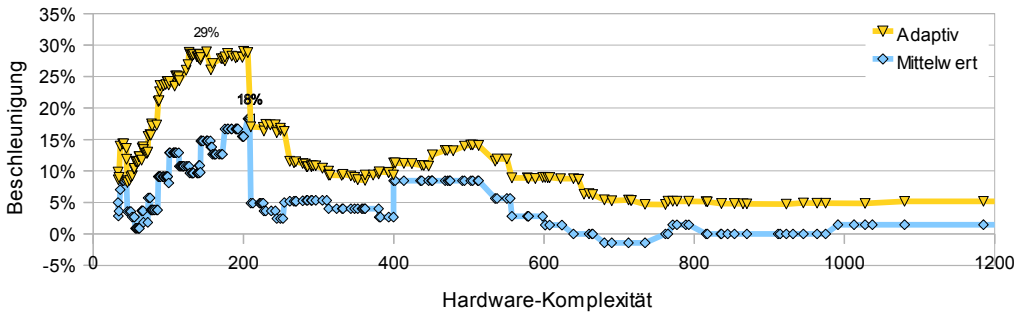
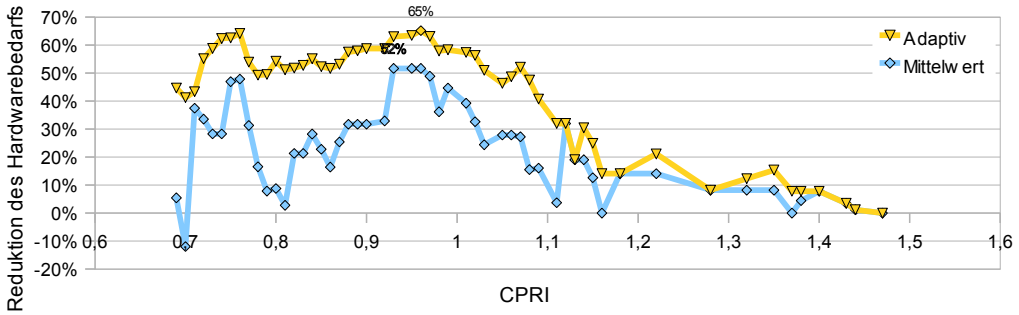


Abbildung 5.14.: Vergleich der Pareto-Approximationen mit und ohne Code-Optimierungen, adaptiv verwendet und durch Mittelwertbildung



(a) Performance-Gewinn bei gleicher Hardware-Komplexität



(b) Reduktion des Hardwarebedarfs bei gleicher Performance

Abbildung 5.15.: Auswirkungen der Code-Optimierungen bei adaptiver Verwendung

5.4.3. Verwandte Arbeiten

Es gibt relativ viele Veröffentlichungen zu adaptiver Programmoptimierung. Meist wird dazu ein Compiler verwendet, es handelt sich dann um *Adaptive Compilation*. Einer der Gründe mag sein, dass der mögliche Performance-Gewinn bei Spezialisierung auf ein einzelnes Programm sehr groß sein kann. Die Wahl der Optimierungen hat also einen großen Einfluss. Für den GAP gibt es bisher allerdings keine Untersuchungen in dieser Richtung.

Ein wesentlicher Nachteil der adaptiven Optimierung nach vorgestelltem Schema ist die lange Zeit, die benötigt wird, um zu einem guten Ergebnis zu kommen. Der Grund dafür ist, dass das zu optimierende Programm sehr oft ausgeführt werden muss, was viel Zeit kostet. Deshalb werden häufig Modelle verwendet, mit denen in sehr kurzer Zeit für ein Programm sehr wahrscheinlich gute Optimierungen gewählt werden können. Es lassen sich zwei Grundideen isolieren: (a) Häufige Ausführung mit Unterstützung durch Vorhersage-Techniken und (b) direkte Optimierung des Programms mit bereits vorhandenem Wissen.

Mit Performance-Zählern arbeiten Cavazos et al. [25, 52]. Sie verwenden sie zur Bestimmung passender Code-Optimierungen und erklären, dass Performance-Counter besser geeignet seien als Eigenschaften des auszuführenden Programms, die mit statischer Analyse gewonnen werden können.

Dubach et al. [51] leiten ein erweitertes Modell her, mit dem sie für ein Programm basierend auf den Parametern der Plattform und der Performance-Counter des Programms bestimmen können, welche Compiler-Optimierungen des GCC sinnvoll wären. Es werden zwei Drittel des insgesamt möglichen Potenzials ausgeschöpft.

„Globalisiert“ wird diese Idee von Fursin und Temam [58], die die verwendete Wissensdatenbank zentral organisieren[15] und für jedermann zugänglich machen. Somit können Nutzer von den Ergebnissen anderer Nutzer profitieren. Es werden statische Programm-Features in Betracht gezogen (vgl. Leather et al. [92]) und es kommen Milepost GCC [57], ein selbst-optimierender Compiler, sowie das *Continuous Collective Compilation Framework (CCC)* zum Einsatz.

Eine Gruppe um Keith D. Cooper beschäftigt sich mit adaptiven Code-Optimierungen. Todd Waterman stellt in seiner Dissertation [168] eine adaptive Heuristik für die Einbindung von Funktionen vor (vgl. Abschnitt 4.3). Cooper und Waterman untersuchen für einen MIPS-Prozessor den Compiler MIPSpro und zeigen, dass mit adaptiver Kompilierung hohe Performancegewinne erreicht werden können. In den Artikeln [41] und [38] werden Systeme zur adaptiven Optimierung vorgeschlagen, wobei hier sehr viele Evaluationen durch Schätzungen ersetzt werden.

Einen Schritt weiter gehen Triantafyllis et al. [158]. Sie schlagen vor, Parameter nicht nur optimal pro Programm, sondern auch schon bei der Kompilierung eines Programms beispielsweise pro Funktion, zu variieren. Hier ist es ebenso wichtig, die für die Optimierung benötige Zeit zu reduzieren.

[15] Webseite: `http://ctuning.org`

5.4.4. Zusammenfassung

Ähnlich wie in Abschnitt 5.3 wird eine automatische Suche im Parameterraum verwendet, um optimale Parameter für den GAP und GAPtimize zu bestimmen. Allerdings wird jetzt für einen einzelnen statt für eine Gruppe von Benchmarks optimiert. Somit kann auf alle „Umgebungsparameter“ reagiert werden, die Optimierung ist daher *adaptiv*. Da die DSE für jeden Benchmark separat durchgeführt werden muss, werden zwei weitere Möglichkeiten zu ihrer Beschleunigung vorgestellt.

Die gefundenen Ergebnisse sind nochmals besser, der maximale Performance-Gewinn ist 29% (Mittelwert bis Hardware-Komplexität 600 ist 17%) und die maximale Hardware-Einsparung ist 65% bei einem Mittelwert von 42%.

5.5. Zusammenfassung und Fazit

In diesem Kapitel 5 wird mit einer automatischen Suche im Parameterraum der Zusammenhang von Performance und Hardware-Bedarf für den GAP untersucht. Dazu werden schrittweise erst nur die Hardware-Parameter optimiert, dann die Hardware-Parameter und die Parameter für die im Kapitel 4 vorgestellten und in GAPtimize implementierten Code-Optimierungen. Abschließend wird diese Betrachtung mit je nur einem Benchmark durchgeführt; es wird also eine adaptive Optimierung verwendet.

Zu diesem Zweck wird das *Framework für Automatic Design Space Exploration* (FADSE) so erweitert, dass Geschwindigkeit und Robustheit erhöht werden. FADSE wird dann an GAP und GAPtimize gekoppelt.

Für den GAP wird eine Abschätzung der Hardware-Komplexität des Arrays und des Befehlszwischenspeichers vorgestellt, somit lässt sich die Größe dieser beiden variablen Bereiche quantifizieren. Seine maximale Hardware-Komplexität liegt bei ca. 10000. Bereits ein Zehntel genügt aber, um die maximale Performance zu erreichen. Eine um 10% geringere Performance kann schon mit einer Komplexität von 330 erreicht werden. Somit kann auch mit geringer Hardware-Komplexität vergleichsweise hohe Performance erreicht werden.

Eine Untersuchung der Parameter für die gefundenen sehr guten Konfigurationen zeigt deutliche Zusammenhänge zwischen den Parametern des Arrays. Bemerkenswert ist, dass mehr Konfigurationsebenen scheinbar sinnvoller für die Beschleunigung des GAP sind als ein größerer Befehlszwischenspeicher.

Die Untersuchung der Code-Optimierungen hat gezeigt, dass mit ihnen besonders die Hardware-Komplexität, die benötigt wird, um eine gewisse Performance zu erreichen, stark reduziert werden kann. Außerdem werden bei allen sehr guten Konfigurationen meistens alle Code-Optimierungen, die für den GAP entwickelt wurden, verwendet.

Wird die Optimierung der Hardware- und Code-Optimierungs-Parameter adaptiv durchgeführt, so können Performance-Gewinn und Komplexitäts-Reduktion weiter ausgebaut werden. Zudem hat sich gezeigt, dass FADSE auch für den sehr großen Parameterraum, den die Hardware- und Software-Parameter aufspannen, noch in der Lage ist, in kurzer Zeit sehr gute Ergebnisse zu finden.

6. Zusammenfassung und Ausblick

6.1. Zusammenfassung

Durch Fortschritte in der Technik, die verwendet wird, um Prozessoren zu fertigen, können diese immer komplexer aufgebaut sein. Diese zusätzlich verfügbaren Ressourcen effektiv zu nutzen ist besonders für einfädige Programme schwierig, da die meisten skalierbaren Architekturen auf mehrfädige Programme ausgerichtet sind.

Der Grid-ALU-Prozessor (GAP) ist eine neuartige Prozessorarchitektur, deren Ziel explizit die Beschleunigung einfädiger Programme ist. Zusätzlich ist das zentrale Element der Architektur, ein Array aus Funktionseinheiten, skalierbar, es kann also für beinahe jede zur Verfügung stehende Chipfläche ein passender Prozessor erzeugt werden.

Da sequentielle Programme heutzutage noch immer sehr häufig verwendet werden und die Weiterverwendung bestehenden Codes meist nicht vermeidbar ist, kann man davon ausgehen, dass es in naher Zukunft noch weitere Prozessorarchitekturen zur Beschleunigung einfädiger Programme geben wird. Ihnen allen wird gemein sein, dass sie bestimmte Eigenschaften aufweisen, die die vorhandenen *Legacy*-Programme aber nicht nutzen können, da sie nicht dafür optimiert wurden.

Um diese Herausforderung zu meistern wird am Beispiel des GAP erklärt, wie Programmoptimierung für eine neuartige Prozessorarchitektur durchgeführt werden kann. Anschließend werden plattformspezifische Optimierungen vorgestellt und in einer gleichzeitigen Optimierung der Hardware- und Software-Parameter möglichst optimale Konfigurationen für den GAP und die Code-Optimierungen gesucht. So kann automatisch die für jedes Hardware-Budget bestmögliche Performance ermittelt werden, die Architektur also quasi automatisch evaluiert werden.

Für den GAP werden die wesentlichen Faktoren isoliert, die die Programmausführung negativ beeinflussen können. Das sind im Wesentlichen Fehlzugriffe auf den Befehlszwischenspeicher, häufige Neukonfigurationen des Arrays, eine niedrige Parallelität unter den Befehlen und somit eine schlechte Ausnutzung des Arrays, sowie nicht oder schwer vorhersagbare Sprünge. Sie werden mit Programmeigenschaften in Zusammenhang gebracht.

Um plattformspezifische Optimierungen anzuwenden wird der *Post-Link-Optimizer* GAPtimize entwickelt, mit dem komplett übersetzte Programme ohne Einbeziehung des Quellcodes optimiert werden können. Mit GAPtimize können auch Informationen über die Programmausführung berücksichtigt werden; somit können Feedback-gesteuerte und iterative Optimierungen durchgeführt werden.

In GAPtimize sind dazu vier Optimierungen implementiert, namentlich bedingte Ausführung, Funktionseinbindung auf Programmebene, statische Kontrollfluss-

Spekulation und eine optimierte Ersetzungsstrategie für die Konfigurationsebenen des GAP. Jedes der Performance-Hemmnisse wird, wie Abbildung 6.1 zeigt, von mindestens einer Optimierung adressiert. Außerdem werden drei weitere Optimierungen und ihre potenziellen Auswirkungen vorgestellt.

Optimierung	GAP-spezifisch	In GAPtimize implementiert	Fehlzugriffe Befehls-zwischenspeicher	Neukonfigurationen	Parallelität der Befehle	Kritische Sprünge
Ersetzungsstrategie QdLRU	X	X	X	X		
Bedingte Ausführung	X	X	X	x		X
Statische Spekulation	X	X	X	x	X	
Einbindung von Funktionen		X	X	x	x	X
Loop Unrolling					X	
Software Pipelining	x				X	
Verkürzung des kritischen Pfades	x			x	X	

Abbildung 6.1.: Übersicht über Optimierungen und die von ihnen beeinflussten Performance-Hemmnisse

Die GAP-Architektur verfügt über einige Hardware-Parameter, die einen hohen Einfluss auf die benötigt Chipfläche des GAP haben. Zudem stehen die verwendeten Hardware-Ressourcen in einem starken Zusammenhang zur Performance. Mit einer automatischen Suche im Parameterraum werden Konfigurationen für den GAP so ermittelt, dass sie möglichst hohe Performance mit möglichst geringem geschätzten Hardware-Aufwand erreichen. Dabei zeigt sich, dass bereits mit Konfigurationen deutlich unter der höchsten möglichen Ausbaustufe das Performance-Maximum erreicht werden kann. Besonders der Einfluss des Befehlszwischenspeichers ist im Vergleich zu den Konfigurationsebenen gering. Nimmt man Abstriche in Höhe von 10% der maximal möglichen Performance in Kauf, so genügt ein Drittel der Komplexität, die für das Performance-Maximum nötig ist.

Außerdem wird demonstriert, dass die Hardware-Parameter zusammen mit den Parametern für die Code-Optimierungen mit einem Tool für die automatische Suche im Parameterraum untersucht werden können. Die in GAPtimize eingebauten Optimierungen können deutlich dazu beitragen, die Performance und Hardware-Effizienz des GAP zu verbessern.

6.2. Ausblick

In dieser Arbeit wurde davon ausgegangen, dass Legacy-Code bereits als übersetztes Programm vorliegt. Dadurch kann exakt dasselbe Programm auf einer Basis-Architektur (z. B. superskalarer Prozessor) und auf dem GAP ausgeführt werden. Das ist ein großer Vorteil bei der Evaluierung des GAP, aber teilweise hinderlich für Pro-

grammoptimierungen. Der Grund dafür ist, dass viele Chancen für Optimierungen bereits vergeben sind, weil z. B. Scheduling und Registerzuweisung sowie Befehlsselektion schon ausgeführt wurden, und nur schwer rückgängig gemacht werden können. Interessant wäre es, als „gemeinsamen Nenner“ Programme in C zu verwenden, die dann mit unterschiedlichen Compilern auf die jeweilige Plattform angepasst werden können. Natürlich sollte dieser Quellcode genauso wie jetzt die Binärdatei als unveränderlich gesehen werden. Für den GAP wäre es für die weitere Forschung eigentlich unumgänglich, eine aktuelle Version des GCC zur Verfügung zu haben. Untersucht werden sollte dann auch, welchen Einfluss die einzelnen Optimierungen des GCC auf die Performance des GAP im Vergleich zu einem Referenzprozessor haben.

Bei der automatischen Suche im Parameterraum wurden Performance und Hardware-Komplexität als Zielfunktionen verwendet. Sinnvoll wäre es, die Leistungsaufnahme und die Wärmeentwicklung dabei zu beachten sowie die maximale Taktfrequenz, mit der der GAP in der gewünschten Konfiguration betrieben werden kann. Dazu müsste der vorhandene Simulator deutlich erweitert werden und zumindest ein Teil des GAP in einer Hardware-Beschreibungssprache umgesetzt werden, sodass der kritische Pfad des Designs bestimmt werden kann.

Außerdem könnte sich der GAP auch für die Ausführung von Programmen mit harten Echtzeitschranken eignen. In Experimenten wurde bereits herausgefunden, dass seine Performance ohne Befehlszwischenspeicher zwar niedriger ist aber immer noch akzeptabel. Der Grund dafür ist, dass die Konfigurationsebenen eigentlich genügen, um Instruktionen schnell für die Ausführung zur Verfügung zu stellen. Die Konfigurationsebenen sind aber wesentlich besser analysierbar als ein Befehlszwischenspeicher. Zudem stellt die bedingte Befehlsausführung eine Möglichkeit dar, auf eine schwer zu analysierende Sprungvorhersagetechnik zu verzichten.

A. Anhang

A.1. Datei mit Ergebnissen des Simulators

Bei Konfiguration des Simulators mit 12 Spalten und 12 Zeilen sowie 16 Konfigurationsebenen werden für ein Beispielprogramm folgende Ergebnisse ermittelt:

```
Number of Lines                          : 12
Number of Columns                        : 12
Number of Layers                         : 16
Memory Latency                           : 24
Cache Configuration - Chunk size         : 8
Cache Configuration - Sets               : 128
Cache Configuration - Lines              : 1
Cache Size in Byte                       : 8192
Cache Size in KByte                      : 8
Duration of Simulation                   : 7
number of clock cycles                   : 446016
hitrate layers                           : 0.059567
hitrate loops                            : 0.933140
hitrate total                            : 0.992708
layer-system layer hits                  : 2181
layer-system loop hits                   : 34166
layer-system hits                        : 36347
layer-system layer misses                : 267
number of interface stall cycles         : 434032
instruction per clock cycle IPC          : 0.798904
Hitrate                                  : 99%
instruction cache miss                   : 379
instruction cache hit                    : 434512
miss_prediction                          : 1218
Reads from memory                        : 12642
Writes to memory                         : 20771
Fetches from instruction cashe           : 378
Number of ALU mapped instructions        : 957968
Number placed instructions               : 3039
Number of executed instructions in ALU   : 356324
Number of taken branches in ALU          : 36761
Number of branches that flush the ALU    : 228
Number of Loops inside the ALU           : 36347
Array Usability                          : 0.067593
Configurations with Length  1            : 1087
Configurations with Length  2            : 84
Configurations with Length  3            : 2075
```

```
[...]
Configurations with Length 12              : 48
Configurations with AVG Length             : 4.821298
Configurations with Length  1 in Loop      : 1043
Configurations with Length  2 in Loop      : 37
Configurations with Length  3 in Loop      : 2047
[...]
Configurations with Length 12 in Loop      : 9
Configurations with AVG Length in Loop     : 4.816683
Lines with  0 used FUs                     : 0
Lines with  1 used FUs                     : 35428
Lines with  2 used FUs                     : 3352
Lines with  3 used FUs                     : 137628
Lines with  4 used FUs                     : 102
Lines with  5 used FUs                     : 14
Lines with  6 used FUs                     : 1
Lines with  7 used FUs                     : 1
[...]
Lines with 33 used FUs                     : 0
Lines with AVG used FUs                    : 2.580438
Lines with  0 used FUs in loop             : 0
Lines with  1 used FUs in loop             : 35100
Lines with  2 used FUs in loop             : 2534
Lines with  3 used FUs in loop             : 137353
Lines with  4 used FUs in loop             : 32
Lines with  5 used FUs in loop             : 0
[...]
Lines with 33 used FUs in loop             : 0
Lines with AVG used FUs in loop            : 2.584605
Configurations with  0 used LSs            : 44
Configurations with  1 used LSs            : 1147
Configurations with  2 used LSs            : 34873
Configurations with  3 used LSs            : 20
[...]
Configurations with 12 used LSs            : 4
Configurations with AVG used LSs           : 2.047140
Configurations with  0 branches            : 2
Configurations with  1 branches            : 35348
Configurations with  2 branches            : 1166
Configurations with  3 branches            : 48
[...]
Configurations with 12 branches            : 0
Configurations with AVG branches           : 1.040149
Configurations with  0 used Cols           : 0
Configurations with  1 used Cols           : 4
Configurations with  2 used Cols           : 17
Configurations with  3 used Cols           : 1087
[...]
Configurations with 12 used Cols           : 15
Configurations with AVG Cols               : 9.510708
```

```
Configurations with  0 used Cols in Loop  : 0
Configurations with  1 used Cols in Loop  : 0
Configurations with  2 used Cols in Loop  : 0
Configurations with  3 used Cols in Loop  : 1052
[...]
Configurations with 12 used Cols in Loop  : 8
Configurations with AVG Cols in loop      : 9.539465
Max used Lines                            : 12
The max Number of Lines used in Loop      : 12
counter_branches_avg                      : 0.215740
counter_branches_taken_avg                : 0.208240
counter_branches_nottaken_avg             : 0.007500
counter_ialus_avg                         : 1.568661
counter_mult_avg                          : 0.371433
counter_memory_avg                        : 0.424604
counter_memory_read_avg                   : 0.375682
counter_memory_write_avg                  : 0.048922
Max number of FUs used in configuration   : 30
Number of Lines in Max-FU-Conf            : 12
Takt for Max-FU-Conf                      : 180
Max number of COLs used in configuration  : 14
Number of Lines in Max-COL-Conf           : 11
Takt for Max-COL-Conf                     : 442280
Max number of LSs used in configuration   : 12
Number of Lines in Max-LS-Conf            : 12
Takt for Max-LS-Conf                      : 436280
The max Number of columns used in Loop    : 11
```

A.2. Übersicht über dynamische Blocklängen der verwendeten Benchmarks

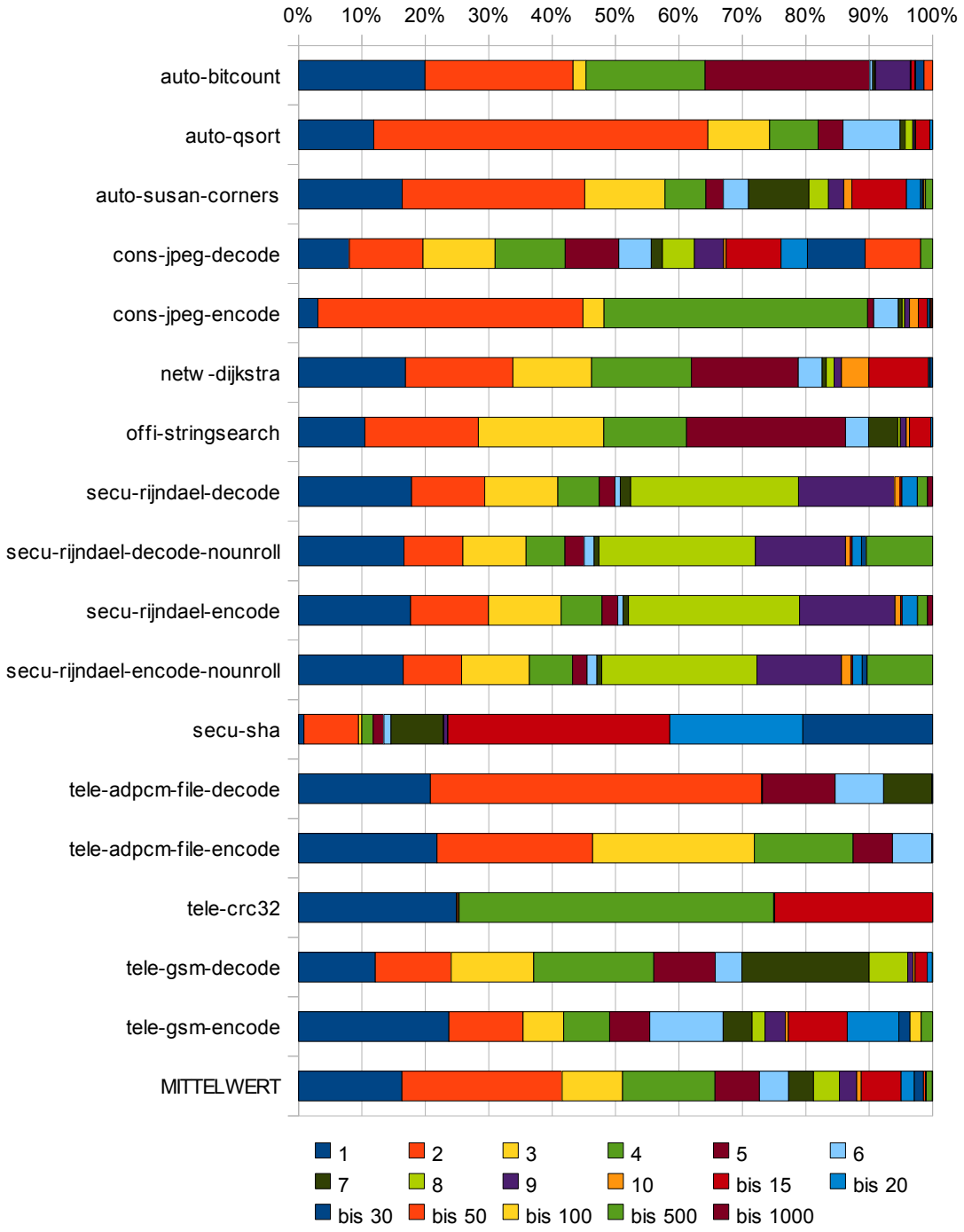
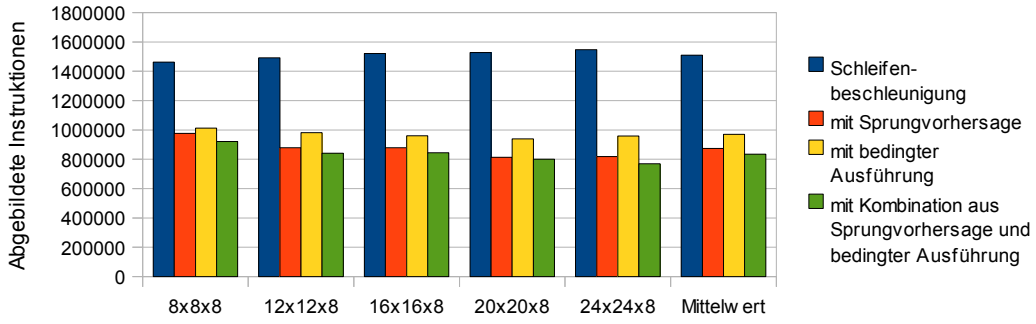


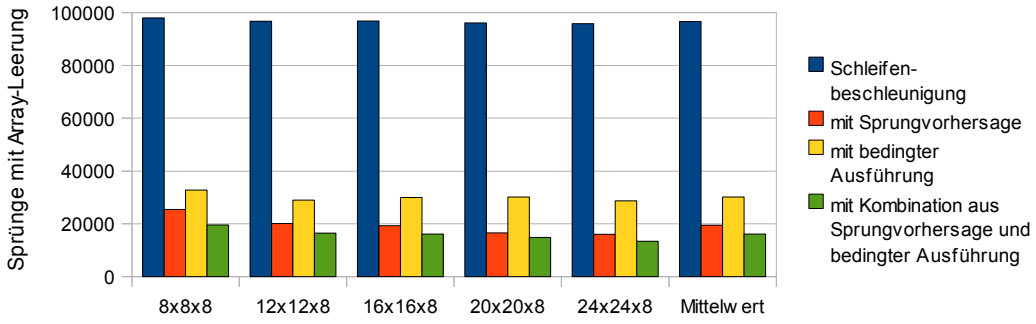
Abbildung A.1.: Blocklängen bei der Ausführung (dynamisch) von 17 ausgewählten MiBench-Benchmarks

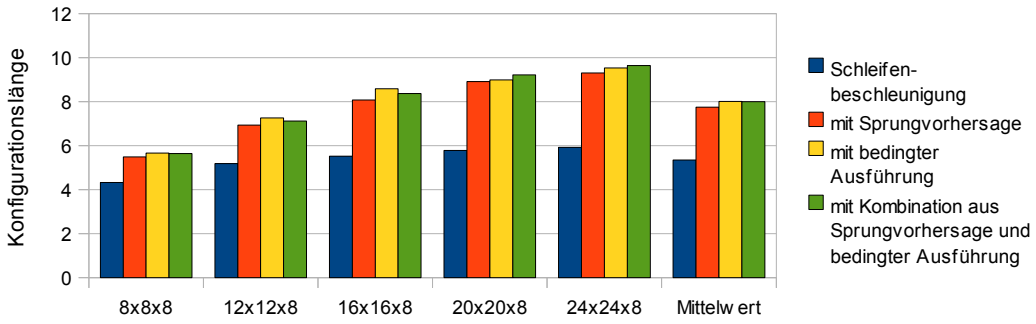
A.3. Beispiel für YAML-Datei zur Steuerung von GAPtimizeYaml

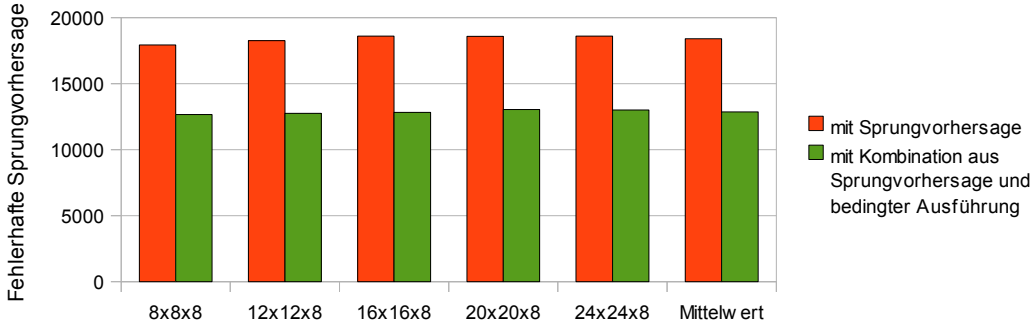
```
--- !stepstepgui.datastructures.YamlParameters
path_dump_directory: c:\_benchmarks\
path_target_directory: c:\gap_und_fadse\ssearch\
path_stream_file: ~
path_compressed_stream_file: c:\_benchmarks\ssearch\ssearch.strc
do_branch_prediction: &1 false
do_generate_statistics: *1
do_reduce_false_ddeps: *1
do_generate_temp_dumps: *1
do_reschedule_single_blocks: *1
do_pex: *1
pex_max_length: 8
pex_max_length_in_loops: 32
do_function_inlining: true
finline_kpi_insns_per_caller: 68
finline_length_of_function: 6161
finline_max_caller_count: 16
finline_weight_of_caller: 58
do_static_speculation: *1
sspec_min_node_weight: 4711
sspec_min_probability_percent: 4711
sspec_hot_function_ratio_percent: 4711
sspec_max_number_of_insns_to_shift: 4711
sspec_max_number_of_blocks_to_shift: 4711
sspec_max_allowed_additional_height: 4711
```

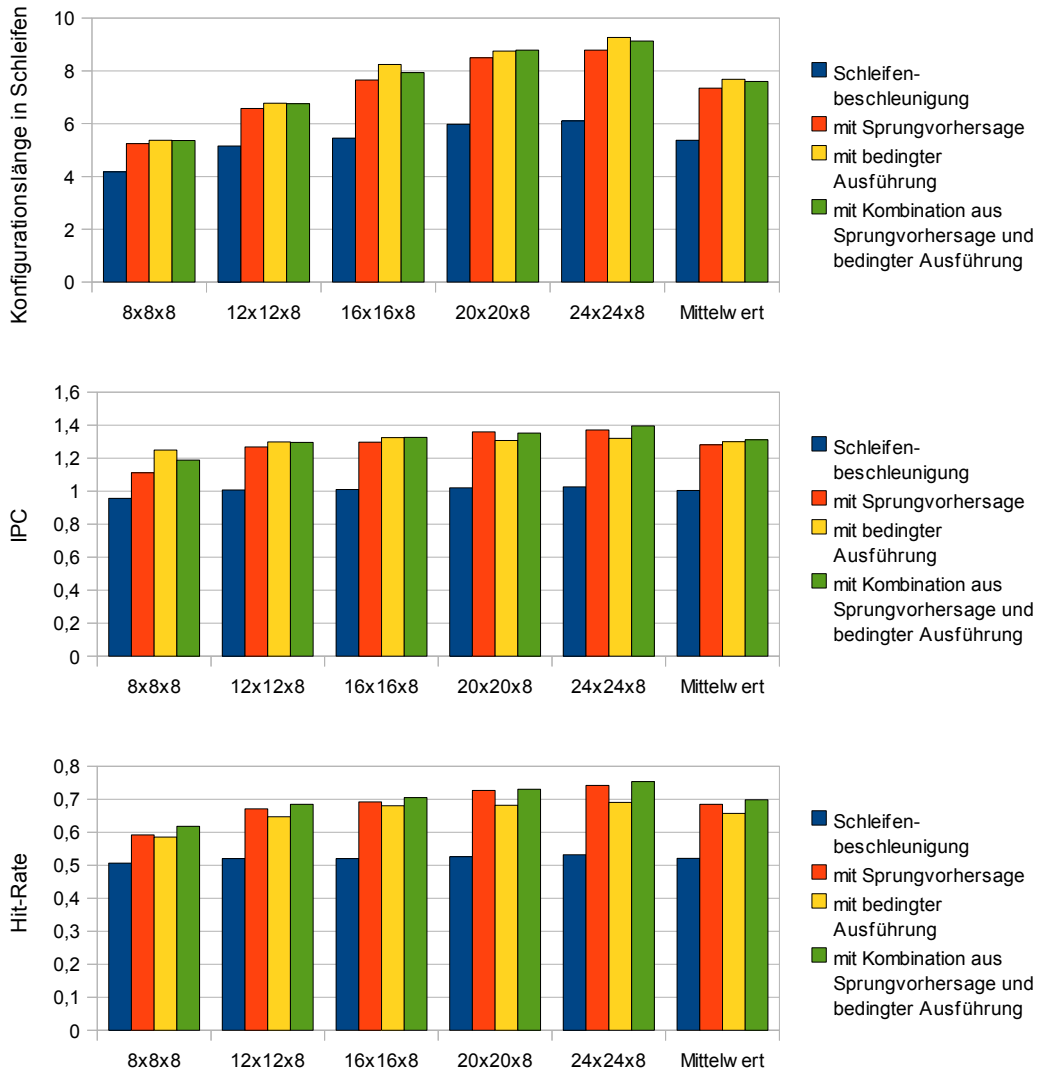
A.4. Weitere Diagramme zur bedingten Ausführung










Konfigurationslänge in Schleifen
10
8
6
4
2
0
8x8x8
12x12x8
16x16x8
20x20x8
24x24x8
Mittelw ert
Schleifen-beschleunigung
mit Sprungvorhersage
mit bedingter Ausführung
mit Kombination aus Sprungvorhersage und bedingter Ausführung
IPC
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
8x8x8
12x12x8
16x16x8
20x20x8
24x24x8
Mittelw ert
Schleifen-beschleunigung
mit Sprungvorhersage
mit bedingter Ausführung
mit Kombination aus Sprungvorhersage und bedingter Ausführung
Hit-Rate
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
8x8x8
12x12x8
16x16x8
20x20x8
24x24x8
Mittelw ert
Schleifen-beschleunigung
mit Sprungvorhersage
mit bedingter Ausführung
mit Kombination aus Sprungvorhersage und bedingter Ausführung

A.5. Konfiguration des SimpleScalar für Vergleiche

```
sim-outorder \
        -redir:sim dijkstra_bpred-dumprunner_sim.log \
        -redir:prog dijkstra_bpred-dumprunner_prog.log \
        -cache:il1 il1:128:64:1:l -cache:il1lat 2  -cache:il2 none \
        -cache:dl1 dl1:1024:32:1:l -cache:dl1lat 2  -cache:dl2 none \
        -mem:lat 24 1 -mem:width 32 \
        -tlb:itlb none -tlb:dtlb none \
        -bpred bimod -bpred:ras 0 -bpred:btb 1 1 \
        -res:memport 1 -fetch:ifqsize 8  -decode:width 4  -issue:width 4  \
        -issue:inorder false -commit:width 4  -res:ialu 4  -ruu:size 16  \
        -fetch:mplat 4 \
        dijkstra.ss input.dat



-redir:sim     rijndael-encode-nounroll_bpred-auto_sim.log
                       # redirect simulator output to file
-redir:prog    rijndael-encode-nounroll_bpred-auto_prog.log
                       # redirect simulated program output to file

-fetch:ifqsize      8 # instruction fetch queue size (in insts)
-fetch:mplat        4 # extra branch mis-prediction latency
-fetch:speed        1 # speed of front-end of machine relative
                      # to execution core
-bpred          bimod # branch predictor type
                      # {nottaken|taken|perfect|bimod|2lev|comb}
-bpred:bimod     2048 # bimodal predictor config (<table size>)
-bpred:ras          0 # return address stack size (0 for no return stack)
-bpred:btb        1 1 # BTB config (<num_sets> <associativity>)

-decode:width       4 # instruction decode B/W (insts/cycle)
-issue:width        4 # instruction issue B/W (insts/cycle)
-issue:inorder  false # run pipeline with in-order issue
-issue:wrongpath true # issue instructions down wrong execution paths
-commit:width       4 # instruction commit B/W (insts/cycle)
-ruu:size          16 # register update unit (RUU) size
-lsq:size           8 # load/store queue (LSQ) size

-cache:dl1    dl1:1024:32:1:l
                      # l1 data cache config, i.e., {<config>|none}
-cache:dl1lat       2 # l1 data cache hit latency (in cycles)
-cache:dl2       none # l2 data cache config, i.e., {<config>|none}
-cache:dl2lat       6 # l2 data cache hit latency (in cycles)

-cache:il1     il1:128:64:1:l
                      # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1lat       2 # l1 instruction cache hit latency (in cycles)
-cache:il2       none # l2 instruction cache config
-cache:il2lat       6 # l2 instruction cache hit latency (in cycles)

-mem:lat         24 1 # memory access latency (<first_chunk> <inter_chunk>)
-mem:width         32 # memory access bus width (in bytes)

-tlb:itlb                none # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb        none # data TLB config, i.e., {<config>|none}

-res:ialu           4 # number of integer ALU's available
-res:imult          1 # number of integer multiplier/dividers available
-res:memport        1 # number of memory system ports available (to CPU)
```

A.6. Performance-Counter zur Funktionseinbindung

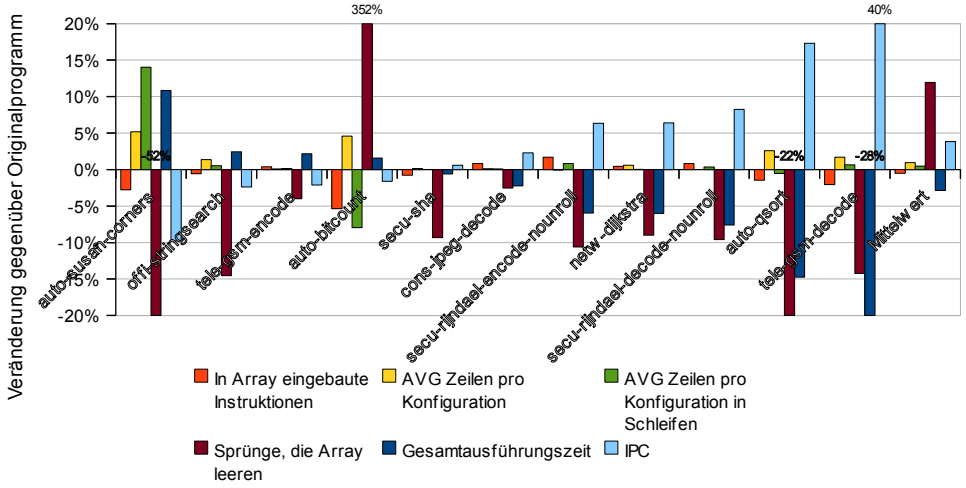


Abbildung A.2.: GAP: Veränderung wichtiger Performance-Counter durch Funktionseinbindung

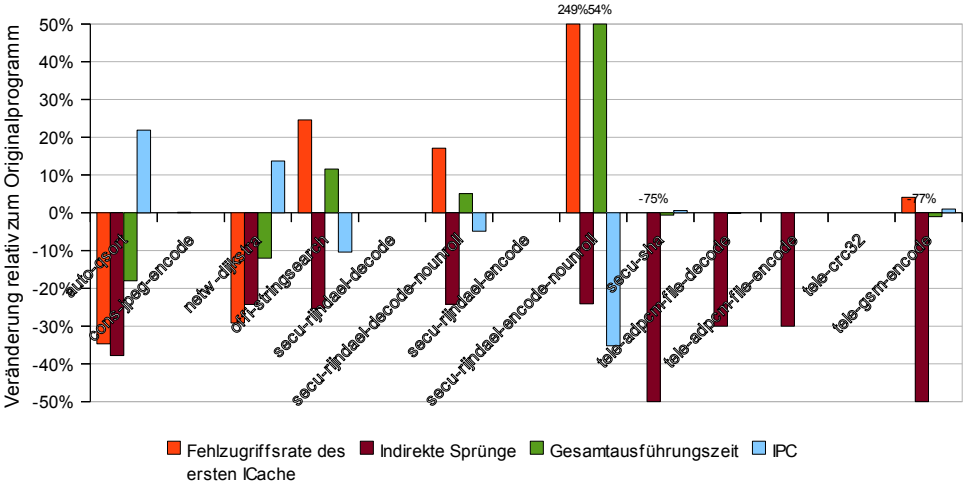


Abbildung A.3.: SimpleScalar: Veränderung wichtiger Performance-Counter für SimpleScalar durch die Funktionseinbindung mit GAPtimize

B. Literaturverzeichnis

[1] AGOSTA, G. ; PALERMO, G. ; SILVANO, C. : Multi-objective co-exploration of source code transformations and design space architectures for low-power embedded systems. In: *Proceedings of the 2004 ACM symposium on Applied computing*. New York, NY, USA : ACM, 2004 (SAC '04). – ISBN 1–58113–812–1, S. 891–896

[2] AHO, A. V. ; LAM, M. S. ; SETHI, R. ; ULLMANN, J. D.: *Compiler. Prinzipien, Techniken und Tools (Pearson Studium): Prinzipien, Techniken und Werkzeuge*. 2. Auflage. PEARSON STUDIUM, 2008. – ISBN 3827370973

[3] *Kapitel* Software Pipelining. In: ALLAN, V. H. .. ; ALLAN, S. J. ..: *The compiler design handbook: optimizations and machine code generation*. CRC Press, 2003. – ISBN 978–0–8493–1240–3, S. 689–737

[4] ALLEN, F. E.: Control flow analysis. In: *Proceedings of a symposium on Compiler optimization*. New York, NY, USA : ACM, 1970, S. 1–19

[5] ALLEN, J. R. ; KENNEDY, K. ; PORTERFIELD, C. ; WARREN, J. : Conversion of control dependence to data dependence. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA : ACM, 1983 (POPL '83). – ISBN 0–89791–090–7, S. 177–189

[6] ALMAGOR, L. ; COOPER, K. D. ; GROSUL, A. ; HARVEY, T. J. ; REEVES, S. W. ; SUBRAMANIAN, D. ; TORCZON, L. ; WATERMAN, T. : Finding effective compilation sequences. In: *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA : ACM, 2004 (LCTES '04). – ISBN 1–58113–806–7, S. 231–239

[7] AMDAHL, G. M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. New York, NY, USA : ACM, 1967, S. 483–485

[8] ANCKAERT, B. ; VANDEPUTTE, F. ; DE BUS, B. ; DE SUTTER, B. ; DE BOSSCHERE, K. : Link-time optimization of IA64 binaries. In: *Euro-Par 2004 Parallel Processing* Springer, 2004, S. 284–291

[9] ARNOLD, M. ; FINK, S. ; SARKAR, V. ; SWEENEY, P. F.: A comparative study of static and profile-based heuristics for inlining. In: *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*. New York, NY, USA : ACM, 2000 (DYNAMO '00). – ISBN 1–58113–241–7, S. 52–64

[10] BELADY, L. A.: A study of replacement algorithms for a virtual-storage computer. In: *IBM Systems* 5 (1966), Nr. 2, S. 78–101

[11] BERMUDO, N. ; KRALL, A. ; HORSPOOL, N. : Control flow graph reconstruction for assembly language programs with delayed instructions. In: *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*, 2005, S. 107–116

[12] BERNSTEIN, D. ; RODEH, M. : Global instruction scheduling for superscalar machines. In: *SIGPLAN Not.* 26 (1991), Nr. 6, S. 241–255. – ISSN 0362–1340

[13] BOEHM, O. ; CITRON, D. ; HABER, G. ; KLAUSNER, M. ; LEVIN, R. : Aggressive Function Inlining with Global Code Reordering / IBM. IBM Research Division, Haifa Research Laboratory, Mt. Carmel 31905, Haifa, Israel, November 2006 (H-0247). – IBM Research Report

[14] BRINKSCHULTE, U. ; UNGERER, T. : *Mikrocontroller und Mikroprozessoren (eXamen.press)*. Springer, 2007. – ISBN 3540468013

[15] BRUENING, D. ; GARNETT, T. ; AMARASINGHE, S. : An infrastructure for adaptive dynamic optimization. In: *CGO '03: Proceedings of the international symposium on Code generation and optimization*, IEEE Computer Society, 2003. – ISBN 076951913X, S. 265–275

[16] BURGER, D. ; AUSTIN, T. : The SimpleScalar tool set, version 2.0. In: *ACM SIGARCH Computer Architecture News* 25 (1997), June, Nr. 3, S. 13–25

[17] BURGER, D. ; KECKLER, S. W. ; MCKINLEY, K. S. ; DAHLIN, M. ; JOHN, L. K. ; LIN, C. ; MOORE, C. R. ; BURRILL, J. H. ; MCDONALD, R. G. ; YODE, W. : Scaling to the End of Silicon with EDGE Architectures. In: *IEEE Computer* 37 (2004), Nr. 7, S. 44–55

[18] CALBOREAN, H. ; VINTAN, L. : Toward an efficient automatic design space exploration frame for multicore optimization. In: *ACACES 2010 poster Abstracts*. Terassa, Spain, Jul. 2010. – ISBN 978 90 382 1631 7, S. 135–138

[19] CALBOREAN, H. ; JAHR, R. ; UNGERER, T. ; VINTAN, L. : Optimizing a Superscalar System using Multi-objective Design Space Exploration. In: *Proceedings of the 18th International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania* Bd. 1. Calea Grivitei, nr. 132, 78122, Sector 1, Bucuresti, May 24-27 2011, S. 339–346. – ISSN 2066-4451

[20] CALBOREAN, H. ; VINTAN, L. : An automatic design space exploration framework for multicore architecture optimizations. In: *Roedunet International Conference (RoEduNet), 2010 9th*. Sibiu, Romania, June 2010. – ISBN 978–1–4244–7335–9, S. 202–207

[21] CALBOREAN, H. A.: *Multi-Objective Optimization of Advanced Computer Architectures using Domain-Knowledge*, Lucian Blaga University of Sibiu, Hermann Oberth Engineering Faculty, Computer Science Department, Diss., September 2011

[22] CALLAHAN, T. J. ; WAWRZYNEK, J. : Adapting software pipelining for reconfigurable computing. In: *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA : ACM, 2000 (CASES '00). – ISBN 1–58113–338–3, S. 57–64

[23] CALLAND, P. ; DARTE, A. ; ROBERT, Y. ; VIVIEN, F. : On the Removal of Anti and Output Dependences. In: *Application-Specific Systems, Architectures and Processors, IEEE International Conference on* 0 (1996), S. 353. – ISSN 1063–6862

[24] CARR, R. W. ; HENNESSY, J. L.: WSCLOCK - a simple and effective algorithm for virtual memory management. In: *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles*. New York, NY, USA : ACM Press, 1981. – ISSN 0163–5980, S. 87–95

[25] CAVAZOS, J. ; FURSIN, G. ; AGAKOV, F. ; BONILLA, E. ; P, M. F. ; TEMAM, O. O.: Rapidly selecting good compiler optimizations using performance counters. In: *In Proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO*, 2007, S. 185–197

[26] CAVAZOS, J. ; O'BOYLE, M. F. P.: Automatic Tuning of Inlining Heuristics. In: *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 1–59593–061–2, S. 14+

[27] CHAO, L.-F. ; LAPAUGH, A. : Rotation scheduling: a loop pipelining algorithm. In: *Proceedings of the 30th international Design Automation Conference*. New York, NY, USA : ACM, 1993 (DAC '93). – ISBN 0–89791–577–1, S. 566–572

[28] CHEN, W. Y. ; CHANG, P. P. ; CONTE, T. M. ; HWU, W. W.: The Effect of Code Expanding Optimizations on Instruction Cache Design. In: *IEEE Trans. Comput.* 42 (1993), September, S. 1045–1057. – ISSN 0018–9340

[29] CHEUNG, W. ; EVANS, W. ; MOSES, J. : Predicated Instructions for Code Compaction. In: *Software and Compilers for Embedded Systems* Bd. 2826. Springer Berlin / Heidelberg, 2003, S. 17–32. – 10.1007/978-3-540-39920-9_3

[30] CHOI, Y. ; KNIES, A. ; GERKE, L. ; NGAI, T.-F. : The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor. In: *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2001 (MICRO 34). – ISBN 0–7695–1369–7, S. 182–191

[31] CHUANG, W. ; CALDER, B. : Predicate prediction for efficient out-of-order execution. In: *Proceedings of the 17th annual international conference on Supercomputing*. New York, NY, USA : ACM, 2003 (ICS '03). – ISBN 1–58113–733–8, S. 183–192

[32] CIFUENTES, C. ; VAN EMMERIK, M. : Recovery of Jump Table Case Statements from Binary Code. In: *Proceedings of the 7th International Workshop on Program Comprehension*. Washington, DC, USA : IEEE Computer Society, 1999 (IWPC '99). – ISBN 0–7695–0179–6, S. 192–

[33] CLARK, N. ; HORMATI, A. ; MAHLKE, S. : VEAL: Virtualized Execution Accelerator for Loops. In: *Proc. of the International Symposium on Computer Architecture*, 2008, S. 389–400

[34] CLEVELAND, L. ; AKIYAMA, Y. : Detecting loop structure in assembly code. In: *AFIPS Conference Proceedings; vol. 55 1986 National Computer Conference*. Arlington, VA, USA : AFIPS Press, 1986. – ISBN 0–88283–049–X, S. 259–265

[35] COELLO, C. A. C. ; VELDHUIZEN, D. A. V. ; LAMONT, G. B.: *Evolutionary Algorithms for Solving Multi-Objective Problems*. 1. Springer, 2002. – ISBN 0306467623

[36] COHN, R. ; GOODWIN, D. ; LOWNEY, P. G. ; RUBIN, N. : Spike: an optimizer for alpha/NT executables. In: *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*. Berkeley, CA, USA : USENIX Association, 1997, S. 3–3

[37] COOK, H. ; SKADRON, K. : Predictive design space exploration using genetically programmed response surfaces. In: *Proceedings of the 45th annual Design Automation Conference*. New York, NY, USA : ACM, 2008 (DAC '08). – ISBN 978–1–60558–115–6, S. 960–965

[38] COOPER, K. D. ; GROSUL, A. ; HARVEY, T. J. ; REEVES, S. ; SUBRAMANIAN, D. ; TORCZON, L. ; WATERMAN, T. : ACME: adaptive compilation made efficient. In: *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA : ACM, 2005. – ISBN 1595930183, S. 69–77

[39] COOPER, K. D. ; HARVEY, T. J. ; WATERMAN, T. : Building a control-flow graph from scheduled assembly code / Rice University. 2002 (TR02-399). – Forschungsbericht

[40] COOPER, K. D. ; HARVEY, T. J. ; WATERMAN, T. : An adaptive strategy for inline substitution. In: *CC'08/ETAPS'08: Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*. Berlin, Heidelberg : Springer-Verlag, 2008. – ISBN 3–540–78790–9, 978–3–540–78790–7, S. 69–84

[41] COOPER, K. D. ; SUBRAMANIAN, D. ; TORCZON, L. : Adaptive Optimizing Compilers for the 21st Century. In: *The Journal of Supercomputing* 23 (2001), May, Nr. 1, S. 7–22

[42] COOPER, K. D. ; TORCZON, L. : *Engineering a Compiler*. Morgan Kaufmann, 2004. – ISBN 1–55860–699–8

[43] COOPER, K. D. ; WATERMAN, T. : Investigating Adaptive Compilation Using the MIPSpro Compiler. In: *In Proc. of the Symp. of the Los Alamos Computer Science Institute*, 2003

[44] COTOFANA, S. ; VASSILIADIS, S. : On the Design Complexity of the Issue Logic of Superscalar Machines. In: *EUROMICRO Conference* 1 (1998), S. 10277. – ISSN 1089–6503

[45] DE BUS, B. ; DE SUTTER, B. ; VAN PUT, L. ; CHANET, D. ; DE BOSSCHERE, K. : Link-time optimization of ARM binaries. In: *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA : ACM, 2004 (LCTES '04). – ISBN 1–58113–806–7, S. 211–220

[46] DE SUTTER, B. ; DE BUS, B. ; DE BOSSCHERE, K. : Link-time binary rewriting techniques for program compaction. In: *ACM Trans. Program. Lang. Syst.* 27 (2005), September, S. 882–945. – ISSN 0164–0925

[47] DEB, K. ; PRATAP, A. ; AGARWAL, S. ; MEYARIVAN, T. : A fast and elitist multiobjective genetic algorithm: NSGA-II. In: *Evolutionary Computation, IEEE Transactions on* 6 (2002), Nr. 2, S. 182–197. – ISSN 1089–778X

[48] DENNING, P. : Thrashing: its causes and prevention. In: *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. New York, NY, USA : ACM, 1968, S. 915–922

[49] DENNING, P. J.: The locality principle. In: *Communications of the ACM* 48 (2005), Nr. 7, S. 19–24. – ISSN 0001–0782

[50] DESMET, V. ; GIRBAL, S. ; RAMIREZ, A. ; TEMAM, O. ; VEGA, A. : ArchExplorer for Automatic Design Space Exploration. In: *IEEE Micro* 30 (2010), Nr. 5, S. 5 –15. – ISSN 0272–1732

[51] DUBACH, C. ; JONES, T. ; BONILLA, E. ; FURSIN, G. ; O'BOYLE, M. : Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In: *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009. – ISSN 1072–4451, S. 78–88

[52] DUBACH, C. ; CAVAZOS, J. ; FRANKE, B. ; FURSIN, G. ; O'BOYLE, M. F. ; TEMAM, O. : Fast compiler optimisation evaluation using code-feature based performance prediction. In: *Proceedings of the 4th international conference on Computing frontiers*. New York, NY, USA : ACM, 2007 (CF '07). – ISBN 978–1–59593–683–7, S. 131–142

[53] DURILLO, J. J. ; NEBRO, A. J. ; LUNA, F. ; DORRONSORO, B. ; ALBA, E. : jMetal: A Java Framework for Developing Multi-Objective Optimization Metaheuristics / Departamento de Lenguajes y Ciencias de la Computacion, University of Malaga. E.T.S.I. Informatica, Campus de Teatinos, Dez. 2006 (ITI-2006-10). – Forschungsbericht

[54] EECKHOUT, L. ; VANDIERENDONCK, H. ; BOSSCHERE, K. D.: How Input Data Sets Change Program Behaviour. In: *In Workshop on Computer-Architecture Evaluation Using Commercial Workloads, held in conjunction with HPCA-8*, 2002

[55] ERTL, M. A. ; KRALL, A. : Removing Anti Dependences by Repairing. In: *Proceedings of the 6th International Conference on Compiler Construction*. London, UK : Springer-Verlag, 1996. – ISBN 3–540–61053–7, S. 33–43

[56] FISHER, J. : Trace Scheduling: A Technique for Global Microcode Compaction. In: *IEEE Transactions on Computers* 30 (1981), S. 478–490. – ISSN 0018–9340

[57] FURSIN, G. ; KASHNIKOV, Y. ; MEMON, A. ; CHAMSKI, Z. ; TEMAM, O. ; NAMOLARU, M. ; YOM-TOV, E. ; MENDELSON, B. ; ZAKS, A. ; COURTOIS, E. ; BODIN, F. ; BARNARD, P. ; ASHTON, E. ; BONILLA, E. ; THOMSON, J. ; WILLIAMS, C. ; O'BOYLE, M. : Milepost GCC: Machine Learning Enabled Self-tuning Compiler. In: *International Journal of Parallel Programming* 39 (2011), S. 296–327. – ISSN 0885–7458

[58] FURSIN, G. ; TEMAM, O. : Collective optimization: A practical collaborative approach. In: *ACM Transactions on Architecture and Code Optimization* 7 (2010), December, S. 20:1–20:29. – ISSN 1544–3566

[59] GREENLEY, D. ; BAUMAN, J. ; CHANG, D. ; CHEN, D. ; ELTEJAEIN, R. ; FEROLITO, P. ; FU, P. ; GARNER, R. ; GREENHILL, D. ; GREWAL, H. ; HOLDBROOK, K. ; KIM, B. ; KOHN, L. ; KWAN, H. ; LEVITT, M. ; MATURANA, G. ; MRAZEK, D. ; NARASIMHAIAH, C. ; NORMOYLE, K. ; PARVEEN, N. ; PATEL, P. ; PRABHU, A. ; TREMBLAY, M. ; WONG, M. ; YANG, L. ; YARLAGADDA, K. ; YU, R. ; YUNG, R. ; ZYNER, G. : UltraSPARC: the next generation superscalar 64-bit SPARC. In: *Compcon '95.'Technologies for the Information Superhighway', Digest of Papers.*, 1995. – ISSN 1063–6390, S. 442–451

[60] GREGG, D. : Comparing Tail Duplication with Compensation Code in Single Path Global Instruction Scheduling. In: *CC '01: Proceedings of the 10th International Conference on Compiler Construction*. London, UK : Springer-Verlag, 2001. – ISBN 3–540–41861–X, S. 200–212

[61] GRONOWSKI, P. E. ; BOWHILL, W. J. ; PRESTON, R. P. ; GOWAN, M. K. ; ALLMON, Y. L. ; MEMBER, A. : High-performance microprocessor design. In: *IEEE Journal of Solid-State Circuits* 33 (1998), S. 676–686

[62] GUPTA, S. ; KECKLER, S. W. ; BURGER, D. : Technology Independent Area and Delay Estimates for Microprocessor Building Blocks / University of Texas at Austin. 2000 (TR2000-05). – Forschungsbericht

[63] GUTHAUS, M. R. ; RINGENBERG, J. S. ; ERNST, D. ; AUSTIN, T. M. ; MUDGE, T. ; BROWN, R. B.: MiBench: A free, commercially representative embedded benchmark suite. In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. Washington, DC, USA : IEEE Computer Society, 2001. – ISBN 0–7803–7315–4, S. 3–14

[64] HABER, G. ; KLAUSNER, M. ; EISENBERG, V. ; MENDELSON, B. ; GUREVICH, M. : Optimization opportunities created by global data reordering. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. Washington, DC, USA : IEEE Computer Society, 2003 (CGO '03). – ISBN 0–7695–1913–X, S. 228–237

[65] HAGOG, M. ; ZAKS, A. : Swing modulo scheduling for gcc. In: *Proceedings of the 2004 GCC Developers Summit* (2004), S. 55–64

[66] HARRIS, L. C. ; MILLER, B. P.: Practical analysis of stripped binary code. In: *SIGARCH Comput. Archit. News* 33 (2005), December, S. 63–68. – ISSN 0163–5964

[67] HAUCK, S. ; FRY, T. ; HOSLER, M. ; KAO, J. : The Chimaera Reconfigurable Functional Unit. In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 1997)*, 1997, S. 87–96

[68] HAUSER, J. ; WAWRZYNEK, J. : Garp: a MIPS processor with a reconfigurable coprocessor. In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 1997)*, 1997, S. 12–

[69] HENNESSY, J. L. ; PATTERSON, D. A.: *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2006. – ISBN 0123704901

[70] HERLIHY, M. ; MOSS, J. E. B.: Transactional memory: architectural support for lock-free data structures. In: *Proceedings of the 20th annual international symposium on computer architecture*. New York, NY, USA : ACM, 1993 (ISCA '93). – ISBN 0–8186–3810–9, S. 289–300

[71] HERRMANN, K. ; WERNER, M. ; MÜHL, G. : A Methodology for Classifying Self-Organizing Software Systems. In: *International Transactions on Systems Science and Applications* 2 (2006), Nr. 1, S. 41–50

[72] HOSTE, K. ; EECKHOUT, L. : Cole: compiler optimization level exploration. In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA : ACM, 2008 (CGO '08). – ISBN 978–1–59593–978–4, S. 165–174

[73] HOSTE, K. ; PHANSALKAR, A. ; EECKHOUT, L. ; GEORGES, A. ; JOHN, L. K. ; DE BOSSCHERE, K. : Performance prediction based on inherent program similarity. In: *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. New York, NY, USA : ACM, 2006 (PACT '06). – ISBN 1–59593–264–X, S. 114–122

[74] HSU, W. C. ; CHEN, H. ; YEW, P. C. ; CHEN, D.-Y. : On the Predictability of Program Behavior Using Different Input Data Sets. In: *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*. Washington, DC, USA : IEEE Computer Society, 2002 (INTERACT '02). – ISBN 0–7695–1534–7, S. 45–

[75] HUANG, W. ; STAN, M. ; SKADRON, K. : Parameterized physical compact thermal modeling. In: *Components and Packaging Technologies, IEEE Transactions on* 28 (2005), December, Nr. 4, S. 615–622. – ISSN 1521–3331

[76] JAHR, R. ; CALBOREAN, H. ; VINTAN, L. ; UNGERER, T. : Boosting Design Space Explorations with Existing or Automatically Learned Knowledge. In: SCHMITT, J. (Hrsg.): *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance* Bd. 7201. Springer Berlin / Heidelberg, 2012. – ISBN 978–3–642–28539–4, S. 221–235

[77] JAHR, R. ; SHEHAN, B. ; UHRIG, S. ; UNGERER, T. : The Grid ALU Processor. In: *Proceedings of ACACES 2008 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems*. L'Aquila, Italy : Academia Press, Ghent, 2008. – ISBN 978–90–382–1288–3, S. 325–328

[78] JAHR, R. ; SHEHAN, B. ; UHRIG, S. ; UNGERER, T. : Optimized Replacement in the Configuration Layers of the Grid Alu Processor. In: *Proceedings of the Second International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC'11)*, KIT Scientific Publishing, 2011. – ISBN 978–3–86644–626–7, S. 9–16

[79] JAHR, R. ; SHEHAN, B. ; UHRIG, S. ; UNGERER, T. : Static Speculation as Post-link Optimization for the Grid Alu Processor. In: GUARRACINO, M. (Hrsg.) ; VIVIEN, F. (Hrsg.) ; TRÄFF, J. (Hrsg.) ; CANNATORO, M. (Hrsg.) ; DANELUTTO, M. (Hrsg.) ; HAST, A. (Hrsg.) ; PERLA, F. (Hrsg.) ; KNÜPFER, A. (Hrsg.) ; DI MARTINO, B. (Hrsg.) ; ALEXANDER, M. (Hrsg.): *Euro-Par 2010 Parallel Processing Workshops* Bd. 6586. Springer Berlin / Heidelberg, 2011, S. 145–152

[80] JAHR, R. ; UNGERER, T. ; CALBOREAN, H. ; VINTAN, L. : Automatic Multi-Objective Optimization of Parameters for Hardware and Code Optimizations. In: WALEED W. SMARI, J. P. M. (Hrsg.): *Proceedings of the 2011 International Conference on High Performance Computing & Simulation (HPCS 2011)*, IEEE, July 2011, S. 308–316. – ISBN 978-1-61284-381-0

[81] JIA, Z. J. ; PIMENTEL, A. ; THOMPSON, M. ; BAUTISTA, T. ; NUNEZ, A. : NASA: A generic infrastructure for system-level MP-SoC design space exploration. In: *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, 2010, S. 41–50

[82] KANG, S. ; KUMAR, R. : Magellan: a search and machine learning-based framework for fast multi-core design space exploration and optimization. In: *Proceedings of the conference on Design, automation and test in Europe*. Munich, Germany : ACM, 2008. – ISBN 978–3–9810801–3–1, S. 1432–1437

[83] KERAMIDAS, G. ; PETOUMENOS, P. ; KAXIRAS, S. : Where replacement algorithms fail: a thorough analysis. In: *CF '10: Proceedings of the 7th ACM international conference on Computing frontiers*. New York, NY, USA : ACM, 2010. – ISBN 978–1–4503–0044–5, S. 141–150

[84] KIM, H. ; MUTLU, O. ; STARK, J. ; PATT, Y. N.: Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution. In: *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2005 (MICRO 38). – ISBN 0–7695–2440–0, S. 43–54

[85] KNIJNENBURG, P. ; KISUKI, T. ; O'BOYLE, M. : Iterative Compilation. In: DEPRETTERE, E. (Hrsg.) ; TEICH, J. (Hrsg.) ; VASSILIADIS, S. (Hrsg.): *Embedded Processor Design Challenges* Bd. 2268. Springer Berlin / Heidelberg, 2002. – ISBN 978–3–540–43322–4, S. 171–187

[86] KÜNZLI, S. ; THIELE, L. ; ZITZLER, E. : Modular design space exploration framework for embedded systems. In: *Computers and Digital Techniques, IEE Proceedings* - 152 (2005), mar, Nr. 2, S. 183–192. – ISSN 1350–2387

[87] KRUEGEL, C. ; ROBERTSON, W. ; VALEUR, F. ; VIGNA, G. : Static disassembly of obfuscated binaries. In: *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*. Berkeley, CA, USA : USENIX Association, 2004 (SSYM'04), S. 18–18

[88] KULKARNI, P. A. ; WHALLEY, D. B. ; TYSON, G. S. ; DAVIDSON, J. W.: Practical exhaustive optimization phase order exploration and evaluation. In: *ACM Trans. Archit. Code Optim.* 6 (2009), April, S. 1:1–1:36. – ISSN 1544–3566

[89] LAM, M. : Software pipelining: an effective scheduling technique for VLIW machines. In: *SIGPLAN Not.* 23 (1988), June, S. 318–328. – ISSN 0362–1340

[90] LAM, M. S. ; WILSON, R. P.: Limits of control flow on parallelism. In: *Proceedings of the 19th annual international symposium on Computer architecture*. New York, NY, USA : ACM, 1992 (ISCA '92). – ISBN 0–89791–509–7, S. 46–57

[91] LATTNER, T. M.: *An Implementation of Swing Modulo Scheduling with Extensions for Superblocks*. Urbana, IL, Computer Science Dept., University of Illinois at Urbana-Champaign, Diplomarbeit, June 2005. – *Siehe* `http://llvm.cs.uiuc.edu`.

[92] LEATHER, H. ; BONILLA, E. ; O'BOYLE, M. : Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In: *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*. Washington, DC, USA : IEEE Computer Society, 2009 (CGO '09). – ISBN 978–0–7695–3576–0, S. 81–91

[93] LEE, M.-H. ; SINGH, H. ; LU, G. ; BAGHERZADEH, N. ; KURDAHI, F. J. ; FILHO, E. M. ; ALVES, V. C.: Design and Implementation of the MorphoSys Reconfigurable ComputingProcessor. In: *Journal of VLSI Signal Processing Systems* 24 (2000), March, Nr. 2–3

[94] LI, S. ; AHN, J. H. ; STRONG, R. D. ; BROCKMAN, J. B. ; TULLSEN, D. M. ; JOUPPI, N. P.: McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA : ACM, 2009. – ISBN 978–1–60558–798–1, S. 469–480

[95] LIN, D. : *Compiler support for predicated execution in superscalar processors*, Department of Computer Science, University of Illinois, Urbana IL, Diplomarbeit, 1992

[96] LLOSA, J. : *Reducing The Impact Of Register Pressure On Software Pipelined Loops*, UPC. Universitat Politècnica de Catalunya, Diss., 1996

[97] LLOSA, J. : Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In: *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA : IEEE Computer Society, 1996 (PACT '96), S. 80–

[98] LOCKHEED MARTIN CORPORATION: JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM. 2008 (2RDU00001 Rev C). – Forschungsbericht

[99] LOKUCIEJEWSKI, P. ; MARWEDEL, P. : *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer, 2011. – I–XVII, 1–260 S. – ISBN 978–90–481–9928–0

[100] LOWNEY, P. G. ; FREUDENBERGER, S. M. ; KARZES, T. J. ; LICHTENSTEIN, W. D. ; NIX, R. P. ; O'DONNELL, J. S. ; RUTTENBERG, J. : The multiflow trace scheduling compiler. In: *J. Supercomput.* 7 (1993), Nr. 1-2, S. 51–142. – ISSN 0920–8542

[101] LUK, C.-K. ; MUTH, R. ; PATIL, H. ; COHN, R. ; LOWNEY, G. : Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In: *Code Generation and Optimization, IEEE/ACM International Symposium on* 0 (2004), S. 15. ISBN 0–7695–2102–9

[102] LYSECKY, R. ; STITT, G. ; VAHID, F. : Warp Processors. In: *ACM Trans. Des. Autom. Electron. Syst.* 11 (2006), Nr. 3, S. 659–681. – ISSN 1084–4309

[103] MAEBE, J. ; RONSSE, M. ; BOSSCHERE, K. D.: DIOTA: Dynamic instrumentation, optimization and transformation of applications. In: *IN PROC. 4TH WORKSHOP ON BINARY TRANSLATION (WBT'02)*, 2002

[104] MAHLKE, S. ; HANK, R. ; BRINGMANN, R. ; GYLLENHAAL, J. ; GALLAGHER, D. ; HWU, W.-M. : Characterizing the impact of predicated execution on branch prediction. In: *Microarchitecture, 1994. MICRO-27. Proceedings of the 27th Annual International Symposium on*, 1994. – ISSN 1072–4451, S. 217–227

[105] MAHLKE, S. A. ; HANK, R. E. ; MCCORMICK, J. E. ; AUGUST, D. I. ; HWU, W.-M. W.: A comparison of full and partial predicated execution support for ILP processors. In: *Proceedings of the 22nd annual international symposium on Computer architecture*. New York, NY, USA : ACM, 1995 (ISCA '95). – ISBN 0–89791–698–0, S. 138–150

[106] MAHLKE, S. A. ; LIN, D. C. ; CHEN, W. Y. ; HANK, R. E. ; BRINGMANN, R. A.: Effective compiler support for predicated execution using the hyperblock. In: *Proceedings of the 25th annual international symposium on Microarchitecture*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1992 (MICRO 25). – ISBN 0–8186–3175–9, S. 45–54

[107] MARIANI, G. ; PALERMO, G. ; SILVANO, C. ; ZACCARIA, V. : Multi-processor system-on-chip Design Space Exploration based on multi-level modeling techniques. In: *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS '09. International Symposium on*, 2009, S. 118 –124

[108] MARIANI, G. ; BRANKOVIC, A. ; PALERMO, G. ; JOVIC, J. ; ZACCARIA, V. ; SILVANO, C. : A correlation-based design space exploration methodology for multi-processor systems-on-chip. In: *Proceedings of the 47th Design Automation Conference*. New York, NY, USA : ACM, 2010 (DAC '10). – ISBN 978–1–4503–0002–5, S. 120–125

[109] MARIANI, G. ; PALERMO, G. ; SILVANO, C. ; ZACCARIA, V. : Meta-model Assisted Optimization for Design Space Exploration of Multi-Processor Systems-on-Chip. In: *Proceedings of the 2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. Washington, DC, USA : IEEE Computer Society, 2009 (DSD '09). – ISBN 978–0–7695–3782–5, S. 383–389

[110] MARIANI, G. ; PALERMO, G. ; ZACCARIA, V. ; SILVANO, C. : An Efficient Design Space Exploration Methodology for Multi-Cluster VLIW Architectures based on Artificial Neural Networks. In: *Proc. IFIP International Conference on Very Large Scale Integration VLSI - SoC 2008*. Rhodes Island, Greece, October 13-15 2008

[111] MCDONALD, R. ; BURGER, D. ; KECKLER, S. W. ; SANKARALINGAM, K. ; NAGARAJAN, R. : TRIPS Processor Reference Manual / Department of Computer Sciences, The University of Texas at Austin. 2005 (TR-05-19). – Forschungsbericht

[112] MEGIDDO, N. ; MODHA, D. S.: Outperforming LRU with an Adaptive Replacement Cache Algorithm. In: *Computer* 37 (2004), Nr. 4, S. 58–65. – ISSN 0018–9162

[113] MEI, B. ; VERNALDE, S. ; VERKEST, D. ; DE MAN, H. ; LAUWEREINS, R. : Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling. In: *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*. Washington, DC, USA : IEEE Computer Society, 2003 (DATE '03). – ISBN 0–7695–1870–2, S. 10296–

[114] MONSIFROT, A. ; BODIN, F. ; QUINIOU, R. : A Machine Learning Approach to Automatic Production of Compiler Heuristics. In: *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. London, UK : Springer-Verlag, 2002. – ISBN 3–540–44127–1, S. 41–50

[115] MUTH, R. : Register Liveness Analysis of Executable Code / Department of Computer Science, University of Arizona. 1998. – Forschungsbericht

[116] MUTH, R. ; DEBRAY, S. K. ; WATTERSON, S. A. ; DE BOSSCHERE, K. : alto: a link-time optimizer for the Compaq Alpha. In: *Software - Practice and Experience* 31 (2001), Nr. 1, S. 67–101

[117] MUTH, R. M.: *Alto: a platform for object code modification*, The University of Arizona, Diss., 1999. – AAI9946835

[118] NAMOLARU, M. ; COHEN, A. ; FURSIN, G. ; ZAKS, A. ; FREUND, A. : Practical aggregation of semantical program properties for machine learning based optimization. In:

Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems. New York, NY, USA : ACM, 2010 (CASES '10). – ISBN 978–1–60558–903–9, S. 197–206

[119] NEBRO, A. ; DURILLO, J. ; GARCIA-NIETO, J. ; COELLO, C. A. ; LUNA, F. ; ALBA, E. : Smpso: A new pso-based metaheuristic for multi-objective optimization. In: *Proceedings of the IEEE Symposium Series on Computational Intelligence*, 2009, S. 66–73

[120] NIKOLOV, H. ; THOMPSON, M. ; STEFANOV, T. ; PIMENTEL, A. ; POLSTRA, S. ; BOSE, R. ; ZISSULESCU, C. ; DEPRETTERE, E. : Daedalus: toward composable multimedia MP-SoC design. In: *Proceedings of the 45th annual Design Automation Conference*. New York, NY, USA : ACM, 2008 (DAC '08). – ISBN 978–1–60558–115–6, S. 574–579

[121] PARK, H. ; FAN, K. ; MAHLKE, S. A. ; OH, T. ; KIM, H. ; KIM, H.-s. : Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA : ACM, 2008 (PACT '08). – ISBN 978–1–60558–282–5, S. 166–176

[122] PARK, J. C. H. ; SCHLANSKER, M. S.: On predicated execution. Hewlett Packard Laboratories, 1991 (HPL-91-58). – Forschungsbericht

[123] PERELMAN, E. ; HAMERLY, G. ; VAN BIESBROUCK, M. ; SHERWOOD, T. ; CALDER, B. : Using SimPoint for accurate and efficient simulation. In: *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA : ACM, 2003 (SIGMETRICS '03). – ISBN 1–58113–664–1, S. 318–319

[124] PISTER, M. ; KÄSTNER, D. : Generic software pipelining at the assembly level. In: *Proceedings of the 2005 workshop on Software and compilers for embedded systems*. New York, NY, USA : ACM, 2005 (SCOPES '05). – ISBN 1–59593–207–0, S. 50–61

[125] PRICE, C. : *MIPS IV Instruction Set, Revision 3.2*. September 1995

[126] PROBST, M. ; KRALL, A. ; SCHOLZ, B. : Register liveness analysis for optimizing dynamic binary translation. In: *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. Washington, DC, USA : IEEE Computer Society, 2002. – ISSN 1095–1350, S. 35–44

[127] QUIÑONES, E. ; PARCERISA, J.-M. ; GONZALEZ, A. : Selective predicate prediction for out-of-order processors. In: *Proceedings of the 20th annual international conference on Supercomputing*. New York, NY, USA : ACM, 2006 (ICS '06). – ISBN 1–59593–282–8, S. 46–54

[128] QUINLAN, D. : ROSE: Compiler support for object-oriented frameworks. In: *Parallel Processing Letters* 10 (2000), Nr. 2-3, S. 215–226

[129] QURESHI, M. K. ; JALEEL, A. ; PATT, Y. N. ; STEELY, S. C. ; EMER, J. : Adaptive insertion policies for high performance caching. In: *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–706–3, S. 381–391

[130] RAJAN, K. ; RAMASWAMY, G. : Emulating Optimal Replacement with a Shepherd Cache. In: *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–3047–8, S. 445–454

[131] RAU, B. R.: Iterative modulo scheduling: an algorithm for software pipelining loops. In: *Proceedings of the 27th annual international symposium on Microarchitecture*. New York, NY, USA : ACM, 1994 (MICRO 27). – ISBN 0–89791–707–3, S. 63–74

[132] RAZDAN, R. ; SMITH, M. D.: A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994, S. 172–80

[133] RODRIGUES, A. F.: Parametric Sizing for Processors / Sandia National Laboratories. 2007. – Forschungsbericht

[134] ROSS QUINLAN, J. : *C4.5: Programs for Machine Learning (Morgan Kaufmann Series in Machine Learning)*. 1. Morgan Kaufmann, 1993. – ISBN 1558602380

[135] ROTENBERG, E. ; BENNETT, S. ; SMITH, J. E.: Trace cache: a low latency approach to high bandwidth instruction fetching. In: *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 1996. – ISBN 0–8186–7641–8, S. 24–35

[136] SCHLANSKER, M. ; KATHAIL, V. : Critical path reduction for scalar programs. In: *Microarchitecture, 1995. Proceedings of the 28th Annual International Symposium on*, 1995, S. 57–69

[137] SCHWARZ, B. ; DEBRAY, S. ; ANDREWS, G. : Disassembly of Executable Code Revisited. In: *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*. Washington, DC, USA : IEEE Computer Society, 2002, S. 45–

[138] SCHWARZ, B. ; DEBRAY, S. ; ANDREWS, G. ; LEGENDRE, M. : PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In: *In Proc. 2001 Workshop on Binary Translation (WBT-2001*, 2001

[139] SEN, R. ; SRIKANT, Y. N.: Executable Analysis using Abstract Interpretation with Circular Linear Progressions. In: *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*. Washington, DC, USA : IEEE Computer Society, 2007 (MEMOCODE '07). – ISBN 1–4244–1050–9, S. 39–48

[140] SHEHAN, B. ; JAHR, R. ; UHRIG, S. ; UNGERER, T. : Enhancing the Grid Alu processor for a better exploitation of the functional units. In: *Mixed Design of Integrated Circuits and Systems (MIXDES), 2010 Proceedings of the 17th International Conference*, 2010, S. 106–111

[141] SHEHAN, B. ; JAHR, R. ; UHRIG, S. ; UNGERER, T. : Optimization and evaluation of the reconfigurable Grid Alu Processor. In: *Computer Architecture and Digital Systems (CADS), 2010 15th CSI International Symposium on*, 2010, S. 11–18

[142] SHEHAN, B. ; JAHR, R. ; UHRIG, S. ; UNGERER, T. : Reconfigurable Grid Alu Processor: Optimization and Design Space Exploration. In: *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, 2010, S. 71 –79

[143] SHEHAN, B. : *Dynamic Coarse Grained Reconfigurable Architectures*, University of Augsburg, Diss., 2010

[144] SHELDON, D. ; VAHID, F. : Making good points: application-specific pareto-point generation for design space exploration using statistical methods. In: *Proceeding of the ACM/-SIGDA international symposium on Field programmable gate arrays*. New York, NY, USA : ACM, 2009 (FPGA '09). – ISBN 978–1–60558–410–2, S. 123–132

[145] SILVANO, C. ; FORNACIARI, W. ; VILLAR, E. : *Multi-Objective Design Space Exploration of Multiprocessor Soc Architectures: The Multicube Approach*. Springer, 2011. – ISBN 9781441988362

[146] SNAVELY, N. ; DEBRAY, S. ; ANDREWS, G. : Unscheduling, Unpredication, Unspeculation: Reverse Engineering Itanium Executables. In: *Reverse Engineering, Working Conference on* 0 (2003), S. 4. – ISSN 1095–1350

[147] STEPHENSON, M. ; AMARASINGHE, S. : Predicting Unroll Factors Using Supervised Classification. In: *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*, IEEE Computer Society, 2005. – ISBN 076952298X, S. 123–134

[148] STEPHENSON, M. ; AMARASINGHE, S. ; MARTIN, M. ; O'REILLY, U.-M. : Meta optimization: improving compiler heuristics with machine learning. In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. New York, NY, USA : ACM, 2003 (PLDI '03). – ISBN 1–58113–662–5, S. 77–90

[149] SUTTER, B. D. ; BUS, B. D. ; BOSSCHERE, K. D. ; KEYNGNAERT, P. ; DEMOEN, B. : On the Static Analysis of Indirect Control Transfers in Binaries. In: ARABNIA, H. R. (Hrsg.): *PDPTA*, CSREA Press, 2000, S. 1013–1019

[150] TAYLOR, M. B. ; KIM, J. S. ; MILLER, J. E. ; WENTZLAFF, D. ; GHODRAT, F. ; GREENWALD, B. ; HOFFMANN, H. ; JOHNSON, P. ; LEE, J.-W. ; LEE, W. ; MA, A. ; SARAF, A. ; SENESKI, M. ; SHNIDMAN, N. ; STRUMPEN, V. ; FRANK, M. ; AMARASINGHE, S. P. ; AGARWAL, A. : The Raw microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. In: *IEEE Micro* 22 (2002), Nr. 2, S. 25–35

[151] TAYLOR, M. B. ; LEE, W. ; MILLER, J. E. ; WENTZLAFF, D. ; BRATT, I. ; GREENWALD, B. ; HOFFMANN, H. ; JOHNSON, P. ; KIM, J. S. ; PSOTA, J. ; SARAF, A. ; SHNIDMAN, N. ; STRUMPEN, V. ; FRANK, M. ; AMARASINGHE, S. P. ; AGARWAL, A. : Evaluation of the Raw microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In: *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, IEEE Computer Society, june 2004. – ISBN 0–7695–2143–6, S. 2–13

[152] THORNTON, J. E.: *Design of a Computer—The Control Data 6600*. Scott Foresman & Co, 1970

[153] THORNTON, J. E.: Parallel operation in the control data 6600. In: *AFIPS '64 (Fall, part II): Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*. New York, NY, USA : ACM, 1965, S. 33–40

[154] THOZIYOOR, S. ; AHN, J. H. ; MONCHIERO, M. ; BROCKMAN, J. B. ; JOUPPI, N. P.: A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In: *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. Washington, DC, USA : IEEE Computer Society, 2008. – ISBN 978–0–7695–3174–8, S. 51–62

[155] TIRUMALAI, P. ; LEE, M. : A heuristic for global code motion. In: *ICYCS'93: Proceedings of the third international conference on Young computer scientists*. Beijing, China, China : Tsinghua University Press, 1993. – ISBN 7–302–00888, S. 109–115

[156] TOMASULO, R. M.: An efficient algorithm for exploiting multiple arithmetic units. In: *IBM J Res Dev* 11 (1967), Nr. 1, S. 25–33

[157] TOUATI, S.-A.-A. ; BARTHOU, D. : On the decidability of phase ordering problem in optimizing compilation. In: *Proceedings of the 3rd conference on Computing frontiers*. New York, NY, USA : ACM, 2006 (CF '06). – ISBN 1–59593–302–6, S. 147–156

[158] TRIANTAFYLLIS, S. ; VACHHARAJANI, M. ; VACHHARAJANI, N. ; AUGUST, D. I.: Compiler optimization-space exploration. In: *In Proceedings of the international symposium on Code generation and optimization*, IEEE Computer Society, 2003, S. 204–215

[159] TYSON, G. S.: The effects of predicated execution on branch prediction. In: *Proceedings of the 27th annual international symposium on Microarchitecture*. New York, NY, USA : ACM, 1994 (MICRO 27). – ISBN 0–89791–707–3, S. 196–206

[160] UHRIG, S. ; MAIER, S. ; KUZMANOV, G. ; UNGERER, T. : Coupling of a Reconfigurable Architecture and a Multithreaded Processor Core with Integrated Real-Time Scheduling. In: *13th Reconfigurable Architectures Workshop (RAW 2006), Rhodos, Greece*, 2006, S. 209–2012

[161] UHRIG, S. ; SHEHAN, B. ; JAHR, R. ; UNGERER, T. : A Two-Dimensional Superscalar Processor Architecture. In: *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World:*, 2009. – ISBN 978–0–7695–3862–4, S. 608–611

[162] UHRIG, S. ; SHEHAN, B. ; JAHR, R. ; UNGERER, T. : The Two-dimensional Superscalar GAP Processor Architecture. In: *International Journal on Advances in Systems and Measurements* 3 (2010), September, Nr. 1 and 2, S. 71–81. – ISSN 1942–261x

[163] VASSILIADIS, S. ; WONG, S. ; COTOFANA, S. D.: The MOLEN pmu-coded Processor. In: *In in 11th International Conference on Field-Programmable Logic and Applications (FPL), Springer-Verlag Lecture Notes in Computer Science (LNCS) Vol. 2147*, 2001, S. 275–285

[164] VASSILIADIS, S. ; WONG, S. ; GAYDADJIEV, G. ; BERTELS, K. ; KUZMANOV, G. ; PANAINTE, E. M.: The MOLEN Polymorphic Processor. In: *IEEE Transactions on Computers* 53 (2004), Nr. 11, S. 1363–1375. – ISSN 0018–9340

[165] VENKATARAMANI, G. ; NAJJAR, W. ; KURDAHI, F. ; BAGHERZADEH, N. ; BOHM, W. ; HAMMES, J. : Automatic compilation to a coarse-grained reconfigurable system-opn-chip. In: *ACM Trans. Embed. Comput. Syst.* 2 (2003), November, S. 560–589. – ISSN 1539–9087

[166] VINTAN, L. ; GELLERT, A. ; FLOREA, A. ; OANCEA, M. ; EGAN, C. : Understanding Prediction Limits Through Unbiased Branches. In: JESSHOPE, C. (Hrsg.) ; EGAN, C. (Hrsg.): *Advances in Computer Systems Architecture* Bd. 4186. Springer Berlin / Heidelberg, 2006, S. 480–487

[167] WARTER, N. ; LAVERY, D. ; HWU, W. : The benefit of predicated execution for software pipelining. In: *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on* Bd. 1, 1993, S. 497–506

[168] WATERMAN, T. : *Adaptive compilation and inlining*. Houston, TX, USA, Rice University, Diss., 2006. – Adviser-Cooper, Keith D.

[169] WENTZLAFF, D. ; AGARWAL, A. : Constructing Virtual Architectures on a Tiled Processor. In: *Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA : IEEE Computer Society, 2006 (CGO '06). – ISBN 0–7695–2499–0, S. 173–184

[170] WOLF, M. ; MAYDAN, D. ; CHEN, D.-K. : Combining loop transformations considering caches and scheduling. In: *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*, 1996, S. 274 –286

[171] YEH, T.-Y. ; PATT, Y. N.: A comparison of dynamic branch predictors that use two levels of branch history. In: *Proceedings of the 20th annual international symposium on Computer architecture*. New York, NY, USA : ACM, 1993 (ISCA '93). – ISBN 0–8186–3810–9, S. 257–266

[172] YI, J. ; JOSHI, A. ; SENDAG, R. ; EECKHOUT, L. ; LILJA, D. : Analyzing the Processor Bottlenecks in SPEC CPU 2000. In: *Proceedings of the 2006 SPEC Benchmark Workshop*, 2006

[173] ZHAO, P. ; AMARAL, J. N.: To Inline or Not to Inline? Enhanced Inlining Decisions. In: *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003, S. 405–419

[174] ZITZLER, E. ; LAUMANNS, M. ; THIELE, L. : SPEA2: Improving the Strength Pareto Evolutionary Algorithm / Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH), Zurich, Switzerland. 2001 (103). – Forschungsbericht

Lebenslauf von Ralf Jahr

Persönliche Daten

Name:	Ralf Werner Jahr
Geburtsdatum:	11.11.1982
Geburtsort:	Augsburg

Ausbildung

1989 – 1993	Johann-Peter-Ring-Grundschule Friedberg/Ottmaring
1993 – 2002	Wernher-von-Braun-Gymnasium Friedberg (WVBG), Abschluss: Abitur
2002 – 2007	Studium der Angewandten Informatik mit Anwendungsfach Betriebswirtschaftslehre an der Universität Augsburg, Abschluss: Diplom-Informatiker Thema der Diplomarbeit: „Organic Computing in den Oracle Core-Technologien“
seit 2007	Wissenschaftlicher Angestellter am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme, Universität Augsburg

Veröffentlichungen als Erstautor

2012

- JAHR, R. ; CALBOREAN, H. ; VINTAN, L. ; UNGERER, T. : Boosting Design Space Explorations with Existing or Automatically Learned Knowledge. In: SCHMITT, J. (Hrsg.): *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance* Bd. 7201. Springer Berlin / Heidelberg, 2012. – ISBN 978–3–642–28539–4, S. 221–235

2011

- JAHR, R. ; UNGERER, T. ; CALBOREAN, H. ; VINTAN, L. : Automatic Multi-Objective Optimization of Parameters for Hardware and Code Optimizations. In: WALEED W. SMARI, J. P. M. (Hrsg.): *Proceedings of the 2011 International Conference on High Performance Computing & Simulation (HPCS 2011)*, IEEE, July 2011, S. 308–316. – ISBN 978-1-61284-381-0
- JAHR, R. ; SHEHAN, B. ; UHRIG, S. ; UNGERER, T. : Optimized Replacement in the Configuration Layers of the Grid Alu Processor. In: *Proceedings of the Second International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC'11)*, KIT Scientific Publishing, 2011. – ISBN 978–3–86644–626–7, S. 9–16

2010

- JAHR, R. ; SHEHAN, B. ; UHRIG, S. ; UNGERER, T. : Static Speculation as Post-link Optimization for the Grid Alu Processor. In: GUARRACINO, M. (Hrsg.) ; VIVIEN, F. (Hrsg.) ; TRÄFF, J. (Hrsg.) ; CANNATORO, M. (Hrsg.) ; DANELUTTO, M. (Hrsg.)

; HAST, A. (Hrsg.) ; PERLA, F. (Hrsg.) ; KNÜPFER, A. (Hrsg.) ; DI MARTINO, B. (Hrsg.) ; ALEXANDER, M. (Hrsg.): *Euro-Par 2010 Parallel Processing Workshops* Bd. 6586. Springer Berlin / Heidelberg, 2011, S. 145–152

2008

- JAHR, R. ; SHEHAN, B. ; UHRIG, S. ; UNGERER, T. : The Grid ALU Processor. In: *Proceedings of ACACES 2008 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems*. L'Aquila, Italy : Academia Press, Ghent, 2008. – ISBN 978–90–382–1288–3, S. 325–328

Weitere Veröffentlichungen

2011

- CALBOREAN, H. ; JAHR, R. ; UNGERER, T. ; VINTAN, L. : Optimizing a Superscalar System using Multi-objective Design Space Exploration. In: *Proceedings of the 18th International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania* Bd. 1. Calea Grivitei, nr. 132, 78122, Sector 1, Bucuresti, May 24-27 2011, S. 339–346. – ISSN 2066-4451

2010

- UHRIG, S. ; SHEHAN, B. ; JAHR, R. ; UNGERER, T. : The Two-dimensional Superscalar GAP Processor Architecture. In: *International Journal on Advances in Systems and Measurements* 3 (2010), September, Nr. 1 and 2, S. 71–81. – ISSN 1942–261x
- SHEHAN, B. ; JAHR, R. ; UHRIG, S. ; UNGERER, T. : Reconfigurable Grid Alu Processor: Optimization and Design Space Exploration. In: *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, 2010, S. 71 –79
- SHEHAN, B. ; JAHR, R. ; UHRIG, S. ; UNGERER, T. : Optimization and evaluation of the reconfigurable Grid Alu Processor. In: *Computer Architecture and Digital Systems (CADS), 2010 15th CSI International Symposium on*, 2010, S. 11–18
- SHEHAN, B. ; JAHR, R. ; UHRIG, S. ; UNGERER, T. : Enhancing the Grid Alu processor for a better exploitation of the functional units. In: *Mixed Design of Integrated Circuits and Systems (MIXDES), 2010 Proceedings of the 17th International Conference*, 2010, S. 106–111

2009

- UHRIG, S. ; SHEHAN, B. ; JAHR, R. ; UNGERER, T. : A Two-Dimensional Superscalar Processor Architecture. In: *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World:*, 2009. – ISBN 978–0–7695–3862–4, S. 608–611