Parameterautotuning für kompilierte Anwendungen durch Verwendung von Debugger-Werkzeugen

Fabian Mikula

Bayreuth Reports on Parallel and Distributed Systems

No. 14, März 2022

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

Bachelorarbeit

Parameterautotuning für kompilierte Anwendungen durch Verwendung von Debugger-Werkzeugen

---

Fabian Mikula

AI2 – Parallele und verteilte Systeme
Universität Bayreuth

Juni 10, 2021

Universität Bayreuth

AI2 – Parallele und verteilte Systeme
Institut für Informatik
Lehrstuhl Angewandte Informatik II - Parallele und verteilte Systeme

Bachelorarbeit

Parameterautotuning für kompilierte Anwendungen durch Verwendung von Debugger-Werkzeugen

Parameter auto-tuning for compiled applications by using debugger tools

Fabian Mikula

1. Prüfer PD Dr. Matthias Korch
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth

2. Prüfer Prof. Dr. Thomas Rauber
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth

Betreuer PD Dr. Matthias Korch

Abgabedatum: Juni 10, 2021

Fabian Mikula
Robert-Koch-Straße 7
95447 Bayreuth
Matrikelnummer: 1564120

Bachelorarbeit
Bachelorarbeit, Juni 10, 2021
Prüfer: PD Dr. Matthias Korch und Prof. Dr. Thomas Rauber
Betreuer: PD Dr. Matthias Korch

Universität Bayreuth
Lehrstuhl Angewandte Informatik II - Parallele und verteilte Systeme
Institut für Informatik
AI2 – Parallele und verteilte Systeme
Universitätsstrasse 30
95447 Bayreuth
Deutschland

Zusammenfassung

Um die Laufzeit einer Anwendung auf einem Rechner zu verbessern, sind Parameter-Autotuner eine gute Wahl[vgl. NK14]. Dafür werden unter anderem die Parameter angepasst, deren Änderung keinerlei Auswirkungen auf die Resultate des Programms haben. Diese Arbeit stellt einen alternativen Ansatz vor, der es ermöglicht bereits kompilierte Programme durch Verwendung von Debugger-Werkzeugen zu optimieren. Dafür wird in der Arbeit mit den theoretischen Grundlagen begonnen, die den Aufbau und die speziellen Formate von Binärdateien darstellen. Auf Basis dessen wird dann geklärt, wie man mit Hilfe der Debug-Informationen, die Speicheradressen der zu optimierenden Variablen erhält. Nachdem die Speicheradressen bekannt sind, wird erklärt, wie man durch Ptrace-Anweisungen die Inhalte der Speicheradressen modifizieren kann. Als nächstes werden Breakpoints näher beleuchtet und wie diese durch die zuvor genannte Ptrace-Library gehandhabt werden können. Nachdem die theoretischen Grundlagen behandelt wurden, wird die Programmarchitektur des Optimierers vorgestellt. Dieses Kapitel beinhaltet unter anderem die Vorstellung der Nutzerschnittstelle, sowie die Arbeitsweisen der Optimierungsvarianten. Anschließend wird dieser Ansatz mit anderen Autotunern verglichen und dessen Funktionalität durch Laufzeitmessungen demonstriert. Abgeschlossen wird die Arbeit durch ein Fazit und einen Ausblick.

Abstract

To improve the runtime of an application on a computer, parameter auto-tuners are a good choice[vgl. NK14]. In this case, the parameters can be adjusted, whose change has no effect on the results of the program. This work presents an alternative approach, which makes it possible to optimize already compiled programs by the use of debugger tools. Therefore the work begins with the theoretical bases, which represent the structure and the special formats of binary files. Based on this, it can be explained how to gain the memory addresses of parameters and variables

with the debug information. After the memory addresses are known, it is explained how to modify the contents of these memory addresses by ptrace instructions. Next, breakpoints are discussed in more detail and how they can be handled through the previously mentioned ptrace library. After the explanation of the theoretical basics, the program architecture of the optimizer is shown. This chapter presents among other things the usage of the user interface, as well as the working methods of the optimization variants. Subsequently, this approach is compared with other auto tuners and its functionality is demonstrated by runtime measurements. The thesis is concluded by a conclusion and an outlook.

Inhaltsverzeichnis

1 Motivation

Aufgrund der unterschiedlichen Hardware von Computern bietet es sich an, die Laufzeit durch Parametertuning zu verbessern[vgl. NK14]. Dabei werden Parameterkonfigurationen optimiert. In diesem Fall beschreiben Parameterkonfigurationen die Parameter, die geändert werden können, ohne dass sich das Ergebnis der Funktion oder des Programms ändert. Vielmehr bezieht sich deren Einfluss auf die Performance des Programms[vgl. RA18]. Ein Beispiel hierfür wäre die Blockgröße einer Schleife, die dabei die Laufzeit des Programms, durch Ausnutzung der daraus resultierenden Cache-Effekte, verbessert[vgl Wike]. Es kann aber auch die Anzahl der Threads erhöht werden, um beispielsweise auf leistungsfähigeren Prozessoren eine kürzere Ausführungszeit zu erreichen[vgl. Wikg], oder auf schwächeren Prozessoren die zuvor eingestellte Threadanzahl zu verringern, um eine Überbelastung des Prozessors zu verhindern. Diese kann beispielsweise durch zu viele Lock-Anfragen innerhalb eines parallelen Programmteils geschehen. Wenn die Zahl der Threads in diesem Fall reduziert wird, finden weniger Anfragen statt und es kommt zu einer geringeren Ausführungszeit der Anwendung, durch geringere Wartezeiten der Threads an den Lock-Stellen[vgl. SASL]. Eine Möglichkeit der Optimierung wäre, einen Online-Parameterautotuner für Programme zu benutzen. Diese verwenden beispielsweise Code-Generierung, die den ursprünglichen Programm-Code durch den neu generierten und potentiell schnelleren Code ersetzt[vgl. NK14; AT11].

Diese Arbeit stellt jedoch einen alternativen Ansatz zum Online-Parameterautotuning vor. Hierbei bedeutet Online-Tuning, dass die Optimierung während der Ausführung des Programms stattfindet[NK14]. Dieser Ansatz sieht vor, dass kompilierte Programme optimiert werden, sodass möglichst wenige Änderungen am Programm selbst vorgenommen werden müssen. Realisiert wurde dies mit Hilfe von Debugger-Werkzeugen, zu denen auch Breakpoints gehören. Breakpoints werden von Debuggern verwendet, um kritische Stellen zu markieren, an denen dann angehalten wird[vgl. Wikb]. Der Optimierer, der in dieser Arbeit thematisiert wird, macht sich die Arbeitsweise der Breakpoints zu Nutze, um Optimierungsstellen im kompilierten Programm zu markieren. An diesen Stellen werden dann die vom Nutzer angegebenen Parameter und Variablen während der Laufzeit optimiert. Dafür müssen die Breakpoints vom Nutzer in einer Schleife positioniert werden, um dann pro Iteration die vom Nutzer spezifizierten Parameter und Variablen auf Basis der Laufzeit

zwischen dem ersten und dem zweiten Betreten des Breakpoints zu optimieren. Ein Breakpoint kann zum Beispiel durch die Angabe der Zeilennummer und der zugehörigen Code-Datei im Optimierer gesetzt werden. Eine weitere Möglichkeit wäre den Breakpoint im C-Programmiercode, des zu optimierenden Programms zu setzen, indem man die betreffende Stelle durch ein Label markiert. Für jeden Breakpoint muss dann noch geklärt werden, welche „Werte“ optimiert werden. Dazu werden unter anderem die Variablen-Namen, die zugehörigen Methoden-Namen und der Variablen-Typ (z.B. Parameter) im Programmiercode benötigt. All diese Informationen werden in der Nutzerschnittstelle des Optimierers angegeben, die durch eine YAML-Konfigurationsdatei repräsentiert wird.

Diese Arbeit deckt zu Beginn alle nötigen theoretischen Grundlagen ab. Darunter befinden sich der Aufbau und das Format von Binärdateien, gefolgt von der Form und der Interpretation der benötigten Debug-Informationen, bis hin zur Beschreibung der Nutzerschnittstelle und der Programmarchitektur. Im Anschluss wird dieser Optimierungs-Ansatz mit anderen Autotunern hinsichtlich der Nutzungsmöglichkeiten verglichen und dessen Funktionalität durch Laufzeitmessungen eines Anwendungsfalls gezeigt. Abschließend wird die Arbeit durch ein Fazit und einen Ausblick beendet.

2 Binärdateiformate und Debuginformationen

2.1 ELF-Format

Das ELF-Format wurde von der TIS(=Tool Interface Standards Comittee) als Standard-Format für Objekt-Dateien ausgewählt, um für diese eine plattformunabhängige Schnittstelle zu bieten[vgl. TIS95]. Objekt-Dateien, sind Dateien, die aus Maschinen-Code in binärer Form bestehen und als Ausgaben eines Assemblers oder eines Compilers hervorgehen[vgl. Wikf]. Diese werden dann von einem Linker zu einer einzigen Objekt-Datei zusammengefasst, die dann unter normalen Umständen auch ausführbar ist[vgl. Pro]. Hierbei wird beim ELF-Format zwischen diesen drei Haupttypen von Objekt-Dateien unterschieden[vgl. TIS95]:

1. relocatable files: Diese Dateien enthalten Code und Daten, die mit anderen Objekt-Dateien verlinkt werden muss, um dann eine ausführbare Objekt-Datei(=executable file), oder eine gemeinsam genutzte Objektdatei (=shared object file) zu erhalten[vgl. TIS95].

2. executable files: Diese Dateitypen können ausgeführt werden und beinhalten alle dafür nötigen Informationen[vgl. TIS95].

3. shared object files: Beinhalten Code und alle weiteren Informationen, die für das statische und dynamische Linken benötigt werden[vgl. TIS95].

Die Hauptaufgabe des Formats besteht darin, den Aufbau einer Objekt-Datei zu spezifizieren. Dies wird vom ELF-Header übernommen, der sich immer am Anfang derartiger Dateien befindet[vgl. TIS95]. Die folgende Graphik 2.1 zeigt einen schematischen Aufbau einer Objekt-Datei, daraus ist ersichtlich, dass diese Segmente, oder Sektionen enthält[vgl. TIS95]. Es ist ebenfalls möglich, dass eine Objekt-Datei, neben Sektionen, ebenfalls Segmente beinhalten kann. Dies führt dazu, dass die Datei zeitgleich geparst und interpretiert werden kann[TIS95; Wika]. Für weiterführende Informationen über den Aufbau von Objekt-Dateien kann die ELF-Dokumentation([TIS95]) zu Rate gezogen werden. Eine Sektion des ELF-Formats

beinhaltet Debugger-Informationen, die durch ein eigenes Format beschrieben werden, dem DWARF-Format, das im nächsten Kapitel näher beleuchtet wird.

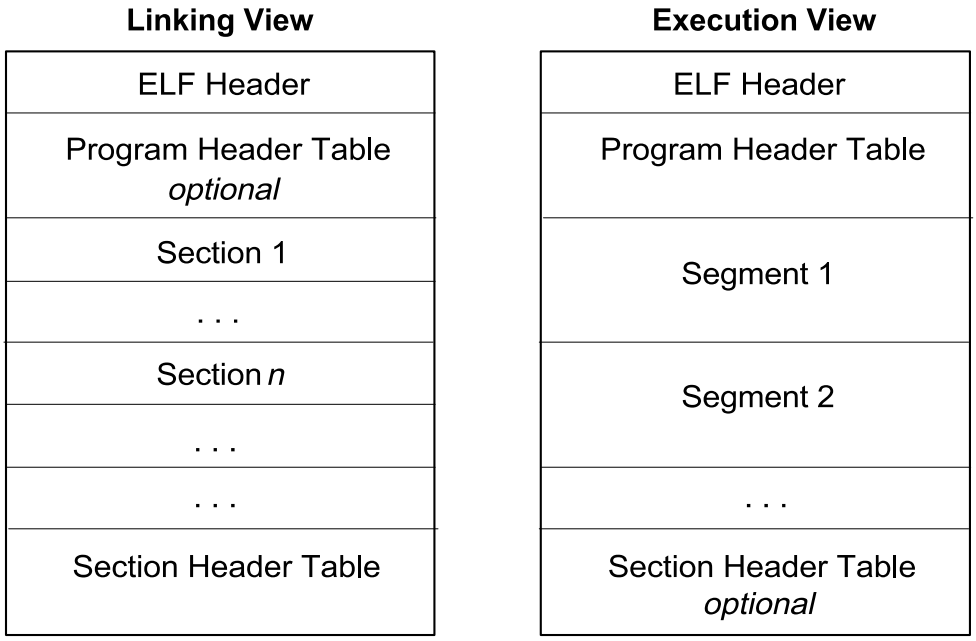

Abb. 2.1: Aufbau einer Objekt-Datei im ELF-Format[TIS95]

2.2 DWARF-Format

Das gesamte Kapitel 2.2 dient dazu, die wichtigsten Aspekte des DWARF-Formats zusammenzufassen, die für das zugrundeliegende Thema dieser Arbeit relevant sind und basiert deshalb vollständig auf der DWARF-Dokumentation([Gro05]).

2.2.1 Allgemeiner Aufbau

Die Hauptaufgabe des DWARF-Formats besteht darin, die Informationen, von Compilern, Assemblern und Linkern, die für das Debuggen von Anwendungen benötigt werden, in einer standardisierten und Debugger- bzw. Programmiersprachen unabhängigen Form zu speichern. Darüber hinaus wurde darauf geachtet, das Format so zu gestalten, dass eine Erweiterbarkeit auf neue Programmiersprachen und deren Eigenheiten gegeben ist. Repräsentiert werden die Informationen durch DIE's(=Debugging Information Entry), die jeweils durch einen Tag einer Klasse zugeordnet werden und einer Reihe von Attributen, die die besonderen Eigenschaften des einzelnen DIE's spezifizieren. Organisiert werden die DIE's in einer Baumstruk-

tur, wobei unter Berücksichtigung spezieller Beziehungen zwischen den DIE's, die Struktur keinen Baum mehr ergibt, sondern einen Graphen. Ein Beispiel hierfür wäre ein DIE, der eine Variable repräsentiert und ein anderer DIE, der den Typ dieser Variable festlegt, sich jedoch nicht in dem Teilbaum des „Variablen-DIE's" befindet. Derartige Beziehungen zwischen den DIE's werden über Variablen ermöglicht, deren Inhalte als Pointer auf den jeweils anderen DIE fungieren[vgl. Gro05]. Für diese Arbeit sind jedoch nur zwei spezielle Sektionen relevant, die im Folgenden näher betrachtet werden. Für zusätzliche Informationen über das DWARF-Format kann die DWARF-Dokumentation([Gro05]) zu Rate gezogen werden.

2.2.2 Debug-Information

Dieses Kapitel beschäftigt sich mit den relevanten DIE-Klassen des DWARF-Formats, die für den Prozess der Speicheradressfindung von globalen und lokalen Variablen, sowie Funktionen und Labels essentiell sind. Dem Kapitel zuvor kann man entnehmen, dass das DWARF-Format in einer Baumstruktur organisiert ist. Das Wurzel-Element dieser Struktur bildet der Compilation-Unit-Entry, folgend als Wurzel-DIE bezeichnet. Der Wurzel-DIE repräsentiert im Normalfall die Objekt-Datei selbst. Wie bereits im Kaptitel 2.1 erwähnt, kann ein executable aus mehreren Objekt-Dateien bestehen[vgl. Pro], so ist es ebenfalls möglich, dass es auch mehrere Wurzel-DIE's in der Baumstruktur geben kann. Die „globalen Objekte" einer Objekt-Datei, zum Beispiel globale Funktionen und globale Variablen werden als direkte Kind-DIE's des Wurzel-DIE's dargestellt.

Die Baum-Struktur der DIE-Einträge zielt darauf ab, einem DIE a das Recht zu gewähren eine beliebige Anzahl von anderen DIE's zu besitzen, die dann als Kinder des DIE's a in der Baum-Struktur dargestellt werden[vgl. Gro05]. Somit werden dann die DIE's, die Parameter, lokale Variablen, sowie Labels einer Funktion repräsentieren, als direkte Kinder des DIE's der Funktion selbst gespeichert. Da nun geklärt wurde, wie die DIE's in der Baum-Struktur organisiert sind, werden abschließend die einzelnen DIE-Klassen näher betrachtet:

- Funktionen werden durch den Tag DW_TAG_subprogram spezifiziert und die Start-Adresse, bzw. die erste Maschinen-Instruktion der Funktion wird durch das Attribut DW_AT_low_pc beschrieben. Dies trifft jedoch nur auf globale Funktionen zu.

- Variablen, ob global oder lokal, werden durch den Tag DW_TAG_variable der entsprechenden Klasse zugewiesen. Die Speicheradresse der globalen Variable wird durch das Attribut DW_AT_location aufgelistet. Bei lokalen Variablen wird

statt der Adresse ein Offset angegeben, dessen Verwendung im Kapitel 3.2.2 behandelt wird.

- Labels werden durch den Tag DW_TAG_label klassifiziert und dessen Speicheradresse wird durch das Attribut DW_AT_low_pc angegeben.

- Parameter werden durch den Tag DW_TAG_formal_parameter spezifiziert und der Offset wird, wie bei den lokalen Variablen, durch das Attribut DW_AT_location angegeben.

Für alle Vertreter der Klassen gilt, dass der Name der Funktion, Variable, Parameter, oder des Labels durch das Attribut DW_AT_name gekennzeichnet wird[vgl. Gro05].

2.2.3 Line-Number-Information

Dieses Kapitel wird nicht das Dekodieren der Zeilennummer-Informationen im Detail erklären, sondern vielmehr einen groben Überblick geben, welchen Nutzen die Informationen haben, wie diese in der Objekt-Datei gespeichert sind und welche Technik angewendet werden muss, um an die relevanten Informationen zu gelangen. Die Zeilennummer-Informationen (engl. Line-Number-Information) werden bei vielen Debuggern dazu verwendet, Breakpoints möglichst anschaulich zu setzen, da diese Informationen eine Brücke zwischen dem unleserlichen Assembler-Code und dem gut lesbaren Programmier-Code des Benutzers bilden. Mit Hilfe dieses Wissens, kann beispielsweise der Benutzer durch Angabe der Zeilen-Nummer im Programmier-Code einen Breakpoint an der korrespondierenden Maschinen-Instruktion im Assembler-Code setzen. Was Breakpoints genau sind und wie diese im Assembler-Code gesetzt werden, wird im Kapitel 5 genauer behandelt.

Der einfachste Ansatz diese Informationen zu speichern, wäre eine Matrix (vgl Abb. 6.1). Zumal diese sehr groß werden kann, wurden einige Optimierungen angewendet. Zu den beiden wichtigsten Optimierungen gehören zum Beispiel das Löschen sich duplizierender Einträge in der Matrix und eine Byte-Code-Sprache einzuführen. Die dazu definierte Byte-Code-Sprache enthält die optimierte Matrix und wird mit Hilfe eines Zustandsautomaten dekodiert, der letztlich auch die Matrix rekonstruiert.

Dateiname	Zeilennummer	Spaltennummer	Ist Statement	...
test.c	10	3	True	...
...	...	...	...	...

Abb. 2.2: Schematischer Aufbau der Zeilennummer-Informations-Matrix[vgl. Gro05]

Der Zustandsautomat besitzt eigene Register und dessen Verhalten wird durch Opcodes, die im Byte-Code neben den eigentlichen Zeilennummern-Informationen enthalten sind, gesteuert. Die Opcodes geben an, wie die nachfolgenden Informationen im Byte-Code zu verarbeiten sind, oder welche Operation als nächstes auf den Registern auszuführen ist[vgl. Gro05]. Für weitere Informationen über den Zustandsautomaten und die Interpretation der Opcodes sollte die DWARF-Dokumentation([Gro05]) verwendet werden. Der Vorteil des Zustandsautomaten und des Byte-Codes liegt darin, dass die Sprache bzw. der Automat und folglich auch die daraus resultierende Matrix um einige Spalten erweitert werden kann[vgl. Gro05].

3 Speicheradresskalkulation

3.1 Funktionen und globale Variablen

Dieses Kapitel beschäftigt sich damit, wie man mit Hilfe der Debug-Informationen, die im DWARF-Format (vgl. Kapitel 2.2 & [Gro05]) vorliegen, an die endgültigen Speicheradressen der Funktionen und globalen Variablen gelangt, die für den Optimierungsprozess benötigt werden. Dieser Prozess wird anhand eines Pseudocodes erklärt, der es ermöglicht die entsprechenden DWARF-DIE's in der Baumstruktur zu finden. Im Optimierer wurde die Bibliothek libdwarf (Dokumentation: [Gro02]) für die Programmiersprache C verwendet. Libdwarf vereinfacht in diesem Fall das Parsen der Baumstruktur, wird aber im weiteren Verlauf der Arbeit nicht weiter thematisiert. Die im Pseudocode verwendeten Methoden-Namen und deren erklärte Arbeitsweise ähneln den Methoden-Namen, die Libdwarf verwendet, um einen einfachen Bezug zum Quellcode zu ermöglichen. Jedoch wurde der Datenzugriff im Pseudocode vereinfacht um die Lesbarkeit zu erhöhen.

Algorithm 1 Find „global object address“ in DWARF structure

```
1: function FINDGLOBALDIEMEMADDRESS(searchedName, searchedTag)
2:     for all CU-headers do
3:         cudie ←dwarf_siblingof(nodie)          ▷ nodie is uninitalized
4:         child ←dwarf_child(cudie)
5:         repeat
6:             if child.tag = searchedTag and child.name=searchedName then
7:                 return readLocation(child)
8:         until (child ←dwarf_siblingof(child)) != NULL
```

Da die DWARF-DIE's in einer Baum-Struktur angeordnet sind[vgl. Gro05], wird als erstes der Wurzel-DIE benötigt, den man erhält, wenn dessen Header erfolgreich ausgelesen wurde und anschließend die Funktion dwarf_siblingof() ausgeführt wird, die den eigentlichen Wurzel-DIE zurückgibt[vgl. Gro02]. Aufgrund der Tatsache, dass alle „globalen Informationen“ in der Baumstruktur die Höhe eins besitzen, müssen in Folge dessen nur alle direkten Kind-DIE's des Wurzel-DIE's durchsucht werden[vgl. Gro05]. Den ersten Kind-DIE eines beliebigen DIE's erhält mit mit der Methode dwarf_child()[vgl. Gro02]. Um nun alle Kind-DIE's auf diesem Level zu erreichen, kann die zuvor genannte Funktion dwarf_siblingof() solange ausgeführt werden, bis kein Geschwister-DIE mehr vorhanden sind. Wie der Name

der Funktion besagt, wird der nächste Geschwister-DIE des angegebenen DIEs zurückgegeben[vgl. Gro02]. Geschwister-DIE's sind alle DIE's, die das gleiche Level besitzen und direkte Kind-DIE's eines Vater-DIE's sind[vgl. Gro05]. Stimmt das Namens-Attribut (DW_AT_name: im Pseudocode name)[vgl. Gro05] und die Klasse (DW_TAG_subprogram | DW_TAG_variable: im Pseudocode tag)[vgl. Gro05] mit dem Namen und der Klasse der gesuchten Funktion oder der gesuchten globalen Variablen überein, so hat man den gesuchten DIE gefunden. Nun gilt es, die Speicheradresse der Funktion oder der Variablen zu ermitteln, wie die Methode readLocation() im Pseudocode andeutet. Die gesuchten Informationen sind in speziellen Attributen im DIE hinterlegt, die mit entsprechenden Methoden aus der Libdwarf-Bibliothek auf Verfühbarkeit überprüft und falls vorhanden auch Ausgelesen werden können[vgl. Gro02; Gro05]. Gleiches gilt auch für die Klasse und den Namen eines DIE's[vgl. Gro05].

3.2 Lokale Variablen

3.2.1 Der Call-Stack

Das Ziel dieses Kapitels ist es, die grundlegende Funktionsweise eines Call-Stacks zu behandeln, um die notwendigen Grundlagen der Speicheradresskalkulation für lokale Variablen und Parameter abzudecken. Der Zweck des Call-Stacks besteht in erster Linie darin, Funktionsaufrufe in einem Programm zu ermöglichen[vgl. Tob; Ran11; Webd]. Dafür werden die Funktionsparameter, die Rücksprungadresse, gesicherte Registerinhalte und lokale Variablen auf dem Call-Stack gespeichert[vgl. Tob; Ran11]. So wird die Gesamtheit aller Informationen, die für einen einzelnen Funktionsaufruf auf dem Call-Stack gespeichert werden als Stack-Frame bezeichnet[vgl. Tob; Ran11]. Zur Veranschaulichung dient die Abb. 3.1, die den schematischen Aufbau eines Call-Stacks zeigt.

Als Vorbereitung für einen Funktionsaufruf werden als erstes dessen Parameter auf den Stack gespeichert[vgl Zey17]. Um zu gewährleisten, dass nach dem Funktionsende die richtige Instruktion ausgeführt wird, muss der aktuelle PC auf dem Stack gespeichert werden[vgl. Ran11, S:219ff]. Diese Adresse wird auch Rücksprungadresse genannt[vgl. Ran11, S:219ff]. Falls ein Frame-Pointer verwendet wird, so wird dieser anschließend auf dem Stack gespeichert[vgl Zey17]. Danach werden die sogenannten „callee-save-registers“ auf dem Stack gespeichert, da diese nach IA32-Konvention (32Bit-Architektur) nicht von der aufrufenden Funktion gesichert werden[vgl. Ran11; Zey17]. Die „callee-save-registers“ sind ebenfalls für 64Bit-Architekturen existent[vgl. Mic20]. Nachdem nun die notwendigen Register

gesichert worden sind, können die lokalen Variablen auf den Stack gespeichert werden[vgl. Ran11; Zey17].

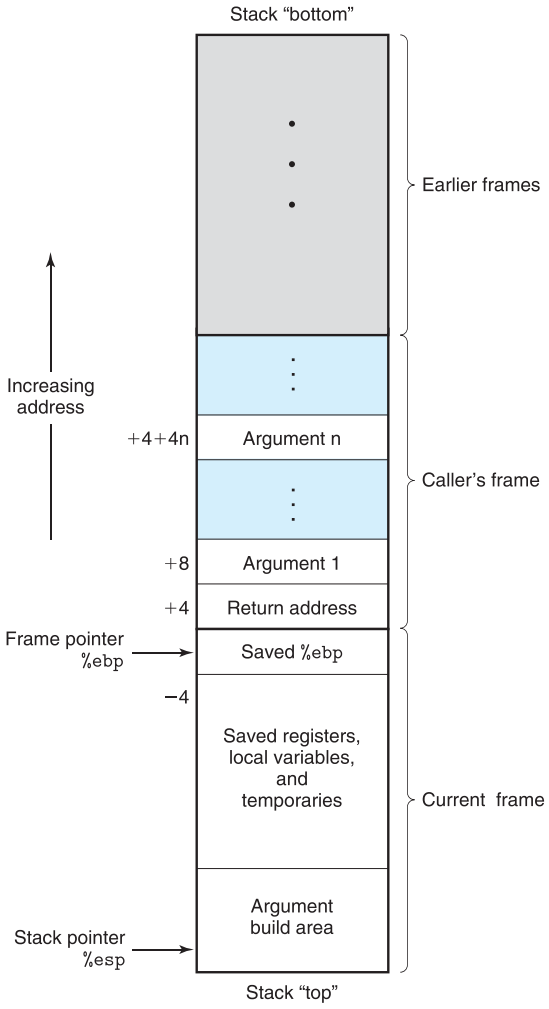

Abb. 3.1: Schematischer Aufbau eines Call-Stacks [Ran11, S:220]

Da jetzt der grundlegende Aufbau eines Callstacks geklärt ist, kann mit der Speicheradresskalukulation von Parametern und lokalen Variablen fortgefahren werden. Dafür spielen zwei Register, die zur Verwaltung des Call-Stacks dienen, eine wichtige Rolle. Das erste Register ist der Stack-Pointer. Der Stack-Pointer zeigt auf das obere Ende vom Stack, indem dessen Adresse im Stack-Pointer Register[vgl. Webd] auch %esp (für x86-Architektur), oder %rsp (für x64-Architektur)[vgl. Ben11b] gespeichert wird. Das wichtigste Register für die Speicheradresskalkulation ist der Frame-Base-Pointer. In der 64-Bit Architektur ist dieser im %rbp Register (%ebp für x86-Architektur) gespeichert[vgl. Ben11b]. Dieser zeigt auf den Beginn des Stack-Frames und wird benutzt, um Parameter und lokale Variablen mit Hilfe eines Offsets zu lokalisieren[vgl. Ran11, S:219ff.]. Auch im DWARF-Fromat wird die

Position von Parametern und lokalen Variablen durch einen Offset in Bezug auf einen logischen Frame-Base-Pointer angegeben(vgl. Kapitel 2.2 & [Gro05]). Somit berechnet sich die Adresse einer lokalen Variable oder eines Parameters mit Hilfe der DWARF-Informationen wie folgt[vgl. Gro05]:

$$Speicheradresse_{lokale\ Variable} = logischer\_rbp + offset\_dwarf\_info \tag{3.1}$$

Da aus der Dwarf-Dokumentation ([Gro05]) hervorgeht, dass der logische-Frame-Base-Pointer typischerweise der Stackpointer des Call-Frames zuvor ist, kann man sich das Parsen und Interpretieren des Byte-Codes der Debug-Frame-Sektion sparen und Stackunwinding verwenden. Wie man nun den benötigten „alten Stackpointer“ mittels Stackunwinding erhalten kann, wird im folgenden Kapitel behandelt.

3.2.2 Stackunwinding

Stackunwinding beschreibt den Vorgang, die obersten Stack-Frames abzubauen und die darin gesicherten Register-Inhalte wiederherzustellen, um beispielsweise die Kontrolle an die aufrufende Funktion abzugeben[vgl. tec21]. Für den Optimierer ist Stackunwinding relevant, da sich gezeigt hat, dass der logische Frame-Base-Pointer mit dem „alten Stackpointer“ übereinstimmt. Um dies auszunutzen, wurde im Optimierer Stackunwinding mit Hilfe der Bibliothek Libunwind-ptrace (Dokumentation [DW21b] und [DW21a]) umgesetzt. Deren Funktionen ermöglichen es, die Prozessor-Register-Inhalte eines Prozesses in jedem Stack-Frame zu betrachten[vgl. DW21c; DW21d; DW21a]. Springt man nun vom aktuell obersten Stack-Frame, einen Stack-Frame nach unten und lässt sich dessen Stackpointer ausgeben, so erhält man genau den Stackpointer, der mit dem gesuchten logischen Frame-Base-Pointer übereinstimmt. Hat man diesen erlangt, so kann man mit Hilfe des DWARF-Offsets die gesuchte Speicheradresse des Parameters, oder der lokalen Variable gemäß der Formel 3.1 berechnen[vgl. Gro05].

Die Speicheradressen mit Hilfe von Stackunwinding zu berechnen, hat den Vorteil, das man nicht auf das Frame-Base-Pointer-Register angewiesen ist. Somit können Programme optimiert werden, die mit der GNU-Compile-Option `-fomit-frame-pointer` kompiliert worden sind, da diese Option dafür sorgt, dass Funktionen keinen Frame-Base-Pointer-Register verwenden, wenn er nicht benötigt wird[vgl. FSF21].

4 Ptrace-Library: Hauptbefehle

Dieses Kapitel dient dazu, die Bibliothek vorzustellen, die den Optimierer befähigt, Breakpoints zu setzen und zu löschen, die Konfigurationen in den Variablen und Parametern zu speichern und Threads zu erkennen. Ferner ermöglicht es, Prozesse oder Threads zu überwachen und zu kontrollieren. Überwachung und Kontrolle bedeutet in diesem Fall, dass es durch Ptrace möglich ist, die Register und den Speicher des „überwachten Thread/Prozess“ auszulesen und zu manipulieren[vgl. Webb]. Zur Veranschaulichung wird das Kapitel anhand eines kleinen Code-Beispiels (Abb. 4.2 & Abb. 4.3) in der Programmiersprache C erklärt. In dem Beispiel-Code wird ein Kind-Prozess erzeugt, der von seinem Erzeuger durch Ptrace überwacht wird und ein Dummy-Programm ausführt. Die Absicht des Überwachers (=Tracer) besteht darin, den Wert der globalen Variable des Dummy-Programms, während der Laufzeit, auf den Wert 10 zu ändern. Dieses Beispiel kommt dem Optimierer, der in dieser Arbeit thematisiert wird, sehr nahe, da dieser ebenfalls einen Kind-Prozess erzeugt und dort dann das zu optimierende Programm ausführt. Die Modifikation der Variablen-Werte, die einer zu optimierenden Konfiguration angehören, werden ähnlich bearbeitet.

```
1 long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *
    data);
```

Abb. 4.1: Ptrace-Anweisung [Webb]

Der Aufbau einer ptrace-Anweisung (Abb. 4.1) ist immer gleich, denn es werden stets vier Parameter übergeben: request, pid, address, daten[vgl. Webb]. Der request gibt an, um welche Art von Anweisung es sich handelt [vgl. Webb]. Die pid (=Prozess-ID [Wikh]) gibt den Prozess an, auf den zugegriffen wird[vgl. Webb]. Das addr-Feld gibt die Speicheradresse an, die ausgelesen oder modifizert werden soll[vgl. Webb]. Die Inhalte des Datenfelds (=data) werden meist an die Speicheradresse, die als Adress-Parameter angegeben wurde, gespeichert[vgl. Webb]. Jedoch benötigen nicht alle Operationen alle Parameter, deshalb werden überflüssige Parameterangaben von der betreffenden Anweisung ignoriert[vgl. Webb].

Da nun der Aufbau einer Ptrace-Anweisung geklärt wurde, kann anschließend der Überwachungsvorgang anhand des Beispiel-Codes erklärt werden. Bevor jedoch die oben genannten Aktionen durchgeführt werden können, muss der überwachte

Prozess/Thread mit dem überwachenden Prozess/Thread verbunden (=attached) werden. Dies kann beispielsweise durch eine ptrace-seize-Anweisung (Abb. 4.3, Z. 25) erfolgen, die der Überwacher ausführen muss[vgl. Webb]. Die beiden angegebenen Flags im data-Parameter, sorgen dafür, dass das überwachte Programm anhält und ein SIGTRAP Signal gesendet wird, falls dort eine execve-, oder clone-Anweisung stattfindet[vgl. Webb]. Execve führt das Dummy-Programm aus[vgl. Weba] und die clone-Anweisung wird beispielsweise beim Erzeugen von pthreads verwendet[vgl. Ben18], was eine Thread-Erkennung möglich macht. Falls ein Thread erkannt wird, so wird dieser ebenfalls angehalten und automatisch attached[vgl. Webb]. Ein weiterer Vorteil der ptrace-seize-Anweisung ist, dass der Überwacher zusätzlich den überwachten Prozess/Thread anhalten kann[vgl. Webb], was beim Breakpoint-Handling mit Threads eine Rolle spielt. Nachdem der Kindprozess erzeugt und attached wurde, so führt dieser excecve (Z. 23) aus, was wie oben schon erklärt, dazu führt, dass der überwachte Prozess an der ersten Instruktion des Dummy-Programms angehalten und ein SIGTRAP Signal gesendet wird[vgl. Webb]. So kann der Tracer durch eine wait()-Anweisung (Z. 31) das Signal abfangen[vgl. Webe] und die Modifikation der globalen Variable vom Dummy-Programm beginnen. Bevor die Variable modifiziert wird, kann der Wert einer Speicheradresse durch dessen Angabe mit Hilfe der ptrace-peektext (Z. 34) Anweisung ausgelesen werden[vgl. Webb]. Die Modifikation der Daten einer Speicheradresse funktioniert durch die ptrace-poketext Anweisung (Z. 38), die den Wert, der im Daten-Feld angegeben wurde, an die im Adress-Feld angegebene Speicheradresse, in den Speicherbereich des Ziel-Prozess (=prozess-id-Feld) speichert[vgl. Webb]. Als letztes kann der Kind-Prozess mit einer ptrace-cont-Anweisung (Z. 41) fortgesetzt[vgl. Webb] und auf dessen Beendingung mit einer wait-Instruktion (Z. 44) gewartet werden[vgl. Webe].

Nachdem nun die wichtigsten ptrace-Anweisungen behandelt wurden, wird im nächsten Kapitel erklärt, was Breakpoints sind und wie diese durch ptrace gehandhabt werden können.

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 uint64_t glob_val=1;
5
6 int main()
7 {
8   printf("dummy is executed with value of global glob_val %ld\n",
      glob_val);
9 }
```

Abb. 4.2: Dummy-Programm, dass im überwachten Prozess ausgeführt wird

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdint.h>
#include <sys/ptrace.h>
#include <pthread.h>

int main()
{
    pid_t child;
    int status;

    //Erzeugung eines Kind-Prozesses
    child=fork();
    if(child==0)
    {
        //Ausfuehrung des Dummy-Programms
        execve("./dummy", NULL, NULL);

        return 0;
    }
    else
    {
        //Verbindung von Ueberwacher und Ueberwachtem
        ptrace(PTRACE_SEIZE, child, 0, PTRACE_O_TRACEEXEC |
    PTRACE_O_TRACECLONE);

        //Auf "Dummy's" erste Instruktion warten
        wait(&status);

        //Auslesen des Werts von glob_val vom Dummy-Programm
        uint64_t value_of_dummy=ptrace(PTRACE_PEEKTEXT, child, 0x404030,
    0);

        //Aenderung des Werts von glob_val aus dem Dummy-Programm auf 10
        //Adresse 0x404030 der Variablen aus dwarf-info
        ptrace(PTRACE_POKETEXT, child, 0x404030, 10);

        //Fortfahren des Kind-Prozess
        ptrace(PTRACE_CONT, child, 0, 0);

        //Warten auf das Ende vom Kind-Prozess
        wait(&status);
    }

    return 0;
}
```

Abb. 4.3: Demo-Tracer, der einen überwachten Prozess startet, indem das Dummy-Programm ausgeführt wird und dessen globale Variable (=glob_val) auf den Wert 10 geändert wird

5 Breakpoints

5.1 Hardware Breakpoints

Hardware Breakpoints, folgend als HW-Breakoints bezeichnet, werden über die Hardware gesetzt und sind in ihrer Anzahl begrenzt. In der x86-Architektur ist es nur möglich, maximal vier HW-Breakpoints simultan zu setzen[vgl. nyn06]. Dafür werden vier spezielle Register verwendet, um die Speicheradresse festzulegen, an der der Breakpoint positioniert werden soll. In der Abbildung 5.1 sind diese als DR0-DR3 gekennzeichnet[vgl. Int86, S.196ff.].

Für die Steuerung der einzelnen HW-Breakpoints wird ein weiteres Register, das Debug Control Register (=DR7), verwendet. In diesem werden durch die 2-Bit Felder R/W0-R/W3, die Aktionen spezifiziert, die den jeweiligen Breakpoint auslösen sollen[vgl. Int86, S.196ff.]:

- 00: Der Breakpoint wird nur bei der Ausführung einer Instruktion getriggert[vgl. Int86, S.196ff.].
- 01: Der Breakpoint wird nur getriggert, falls ein schreibender Zugriff auf die entsprechende Adresse stattfindet[vgl. Int86, S.196ff.].
- 10: Undefiniert[vgl. Int86, S.196ff.].
- 11: Der Breakpoint wird getriggert, falls ein lesender oder schreibender Zugriff auf die entsprechende Adresse stattfindet, allerdings wird ein Instruktionsaufruf ignoriert[vgl. Int86, S.196ff.].

Als nächstes muss die Länge des Daten-Objekts angegeben werden, das überwacht werden soll. Dies wird, ähnlich wie bei den Breakpoint-Triggern, über 2-Bit Felder, LEN0-LEN3 im Kontroll-Register CR7 gehandhabt[vgl. Int86, S.196ff.]:

- 00: 1 Byte wird überwacht. Falls RW für den Breakpoint auf 00 (=Intstruktionsausführung), sollte LEN ebenfalls auf 00 gesetzt werden[vgl. Int86, S.196ff.]

- 01: 2 Bytes werden überwacht[vgl. Int86, S.196ff.].
- 10: Undefiniert[vgl. Int86, S.196ff.].
- 11: 4 Bytes werden überwacht[vgl. Int86, S.196ff.].

Der Kontext des HW-Breakpoints kann über die 1-Bit Felder, L0-L3 und G0-G3 des Kontroll-Registers festgelegt werden. Hierbei steht das "L"für lokal, was bedeutet, dass der Breakpoint nach jedem Kontextwechsel (=Prozess-Wechsel [vgl. Wikd]) des Prozessors deaktiviert wird. Ist das entsprechende G-Feld (G für global) aktiviert, so wird der Breakpoint für alle Prozesse getriggert, falls diese auf die zuvor definierte Art und Weise, auf die entsprechnde Speicheradresse zugreifen[vgl Int86, S.196ff.].

Das Debug Status-Register (=DR6) kann vom Debugger verwendet werden, um die Umstände zu bestimmen, die den Breakpoint ausgelöst haben[vgl. Int86]. Auf die Interpretation der einzelnen Felder wird nicht mehr eingegangen und kann bei Bedarf in der Quelle [Int86] nachgelesen werden.

Der Vorteil der HW-Breakpoints liegt darin, dass diese extrem schnell bearbeitet werden können und die Überwachung von Speicherzugriffen („Memory-Breakpoints“) erleichtern[vgl. nyn06]. Für den Optimierer, werden jedoch Software-Breakpoints verwendet, welche im folgendem Kapitel behandelt werden.

5.2 Software Breakpoints

5.2.1 Special Opcode und Signalübertragung

Software Breakpoints, folgend als SW-Breakpoints bezeichnet, bilden die Basis des in der Arbeit thematisierten Optimierers, da diese als Grundlage des Optimierungsprozess dienen. Die SW-Breakpoints simulieren einen Interrupt, der genau dann auftritt, falls die Instruktion ausgeführt wird, in der sich der Breakpoint befindet[vgl. Ben11a]. Erkannt wird der SW-Breakpoint durch einen 1-Bit Opcode (=0xCC), der den Anfang jeder Instruktion ersetzen kann. Dies führt dazu, dass nach der Erkennung des SW-Breakpoints der vordefinierte debug-exception-handler aufgerufen wird[vgl. Int21, S.586]. Wird nun die entsprechende Instruktion im Code ausgeführt, an der ein SW-Breakpoint gesetzt wurde, so wird der Prozess gestoppt und unter Linux ein Signal gesendet[vgl. Ben11a]. Das entsprechende Signal hat die Nummer

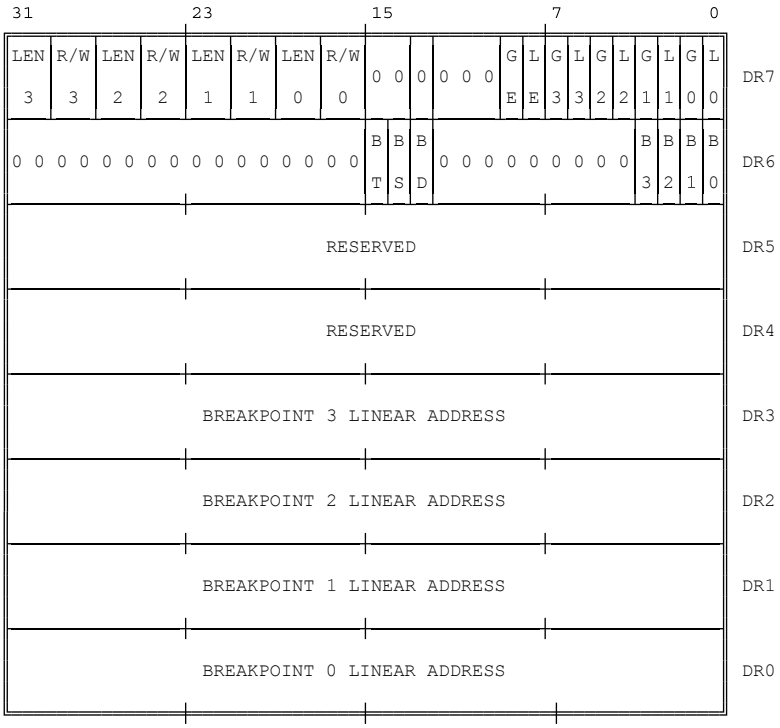

Abb. 5.1: Schematischer Aufbau der Debug-Register [Int86]

5 und ist mit dem Namen SIGTRAP versehen[vgl. Webc].

Der Vorteil gegenüber den Hardware-Breakpoints besteht darin, dass beliebig viele Breakpoints im Code simultan gesetzt werden können. „Memory-Breakpoints“ sind mit SW-Breakpoints nicht realisierbar[nyn06]. Wie SW-Breakpoints nun gesetzt werden können, wird im folgenden Kapitel thematisiert.

5.2.2 Allgemeines Breakpointhandling mit Ptrace

Dieses Kapitel beschreibt das allgemeine Handling von Software-Breakpoints mit Ptrace. Voraussetzung für das Setzen eines Breakpoints über Ptrace ist, dass der zu überwachende Prozess mit Ptrace attached wurde[vgl Webb]. Wie bereits im Kapitel 5.2.1 erwähnt, wird ein SW-Breakpoint über einen speziellen Opcode 0xCC in einer Instruktion gesetzt[vgl. Int21, S.586]. Umgesetzt wird dies mit einem ptrace-poketext Befehl, der als data-Parameter den Opcode 0xCC übergeben bekommt(vgl. Abb. 5.2)[vgl. Ben11a].

```
1 ptrace(PTRACE_POKETEXT, pid, addr, 0xCC);
```

Abb. 5.2: Ptrace-Anweisung, die einen SW-Breakpoint, im Prozess pid und an der Speicheradresse addr setzt [Webb] & [vgl. Ben11a]

Nachdem ein Prozess die Breakpoint-Instruktion ausgeführt hat, wird ein SIGTRAP-Signal gesendet[vgl. Ben11a]. Dies kann beispielsweise von einer WIFSTOP()-Anweisung abgefangen und überprüft werden, ob es sich tatsächlich um einen Breakpoint handelt, indem die Signal-Nummer mit der Nummer 5 übereinstimmt[vgl Webe; Webc].

Um den Breakpoint zu löschen, muss der ursprüngliche Opcode der Instruktion wiederhergestellt werden und der PC (=Program Counter) des Prozesses/Threads um eine Instruktion nach hinten verschoben werden[vgl. Ben11a]. Den ursprünglichen Opcode kann man ebenfalls mit einer ptrace-poketext Instruktion wiederherstellen, die als data-Parameter die Originaldaten übergeben bekommt[vgl. Ben11a; Webb]. Den PC kann man durch die Hintereinanderausführung der Instruktionen zurücksetzen, die in folgender Abb. 5.3 zu sehen sind[vgl. Ben11a; Mic11]:

```
1 struct user_regs_struct regs;
2 //loesche den Opcode des Breakpoints
3 ptrace(PTRACE_POKETEXT, pid, addr, original_data);
4
5 //setze PC=PC-1
6 ptrace(PTRACE_GETREGS, pid, 0, &regs);
7 regs.rip--=1;
8 ptrace(PTRACE_SETREGS, pid, 0, &regs);
```

Abb. 5.3: Folge von C-Anweisungen, die einen Breakpoint im Prozess pid an der Adresse addr löscht[vgl. Ben11a; Mic11]

Die erste ptrace-poketext-Anweisung stellt die ursprüngliche Instruktion an der angegebenen Speicheradresse wieder her und löscht somit die Basis des Breakpoints[vgl. Ben11a]. Um anschließend den PC um eins zurückzusetzen, speichert der folgende ptrace-getregs Befehl die Prozess-Register, in ein Struct vom Typ struct user_regs_struct[vgl. Webb; Ben11a]. Danach wird durch die dritte Anweisung im struct das Äquivalent zu den Prozess-Registern, das den PC enthält, um eins verringert und durch die zweite ptrace()-Anweisung im PC-Register zurückgespeichert[vgl. Webb; Ben11a]. Dies ist erforderlich, da die ursprüngliche Instruktion, die durch den Breakpoint gestoppt wurde, noch nicht ausgeführt wurde[vgl. Ben11a]. Besteht die Absicht den Breakpoint zu reaktivieren, muss etwas Zusatzarbeit investiert werden, da nach der Löschung des Breakpoints und der Ausführung der Instruktion nicht klar ist, welche Instruktion als nächstes ausgeführt wird[vgl. Bra11]. Deshalb wird ein ptrace-singlestep-Befehl (siehe [Webb]) verwendet, der nur die darauffolgende Instruktion des überwachten Prozesses ausführt und danach diesen erneut in einem TRAP-Zustand geraten lässt[vgl. Webb; Bra11]. Nachdem das entsprechende Signal angekommen ist, was bedeutet, dass die Instruktion ausgeführt wurde, kann der Breakpoint erneut durch die oben genannte ptrace()-Anweisung gesetzt werden[vgl. Webb; Bra11]. Veranschaulicht wird der Prozess durch die folgende Abbildung 5.4,

die einen Zustandsautomat zeigt.

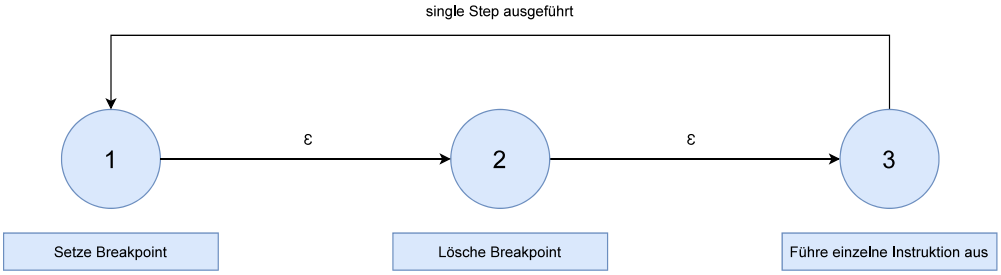

Abb. 5.4: Abbildung eines Zustandsautomaten, der den Ablauf des Breakpointhandlings veranschaulicht

6 Programmarchitektur

Der in dieser Arbeit thematisierte Optimierer hat das Ziel, ein vorhandenes kompiliertes Programm auszuführen und während der Laufzeit ein Parametertuning durchzuführen. Für den Optimierungsprozess sollen Debugger-Werkzeuge verwendet werden, deren benötigten Grundlagen in den Kapiteln zuvor behandelt worden sind. Dieses Kapitel hat die Aufgabe die zugrundeliegende Programmarchitektur des Optimierers vorzustellen. Begonnen wird mit der Benutzerschnittstelle, die die Handhabung des Optimierers erklärt.

6.1 Benutzerschnittstelle

Als Benutzerschnittstelle dient eine YAML-Konfigurationsdatei, deren Aufbau in der folgenden Abbildung veranschaulicht wird.

```
1  program_path: "src/test_programs/c/parallel/parallel" #pfad
2  search_space_exploration: "SIMULATED_ANNEALING"  #such algorithmus (=SA)
3  optimizing_technique: TOGETHER       #optimierungsvariante
4  optimize_main_thread: False          #main thread optimieren?
5  breakpoints:
6     - type: OPTIMIZER                 #Breakpointtyp
7       lifetime: FOREVER               #Lebensdauer
8       location: "breakpoint"          #labelname
9       method: "start_mat_mult"        #Breakpoint-Methode
10      start_temperature: 50000        #Starttemperatur (nur SA)
11      alpha_value: 0.95               #Alpha (nur SA)
12      max_optimization_steps: 40      #max. Optimierungsschritte (nur SA)
13      symbols:                        #Optimierungsvariablen u. Parameter
14         - name: "block"              #Name der Variable
15           type: PARAMETER            #Typ
16           smallest: 1                #kleinst moeglicher Wert
17           biggest: 20000             #groesstmoeglicher Wert
18
19    - type: ZONEBREAKER
20      lifetime: FOREVER
21      line: 198                       #Fuer Zeilennummerangabe muss der
22      location: "parallel.c"          #Dateiname in location angegeben
23      method: "mat_mult_parallel_tiling"  #werden
```

Abb. 6.1: Beispiel einer Konfigurationsdatei für den Optimierer im YAML-Format

Als erstes wird der Programmpfad der zu optimierenden Anwendung angegeben. Deren Programmargumente können dem Optimierer übergeben werden, da diese

an die zu optimierende Anwendung beim Programmstart weitergereicht werden. Danach kann der Suchalgorithmus spezifiziert werden, der den Suchraum erforscht. Momentan sind zwei Suchalgorithmen implementiert. Der Hillclimb-Algorithmus kann durch den String „HILLCLIMB“ ausgewählt werden und analog dazu kann der Simulated-Annealing-Suchalgorithmus durch den String „SIMULATED_ANNEALING“ gewählt werden. Durch den Key optimizing_technique wird die Optimierungsvariante festgelegt. Bisher sind 3 Varianten implementiert, deren genauen Abläufe im Kapitel 6.3 behandelt werden:

- SEQUENTIAL: Legt die sequentielle Optimierung fest, die nur eine Optimierung in sequentiellen Methoden zulässt. Falls ein Breakpoint in einem Thread liegen kann, sollte eine der parallelen Optimierungsvarianten gewählt werden.

- SEPERATE: Parallele Optimierungsvariante, die eine sequentielle Optimierung unterstützt, aber durch das Breakpointhandling potentiell langsamer als diese sein kann. Werden Threads verwendet und liegt ein Breakpoint in einem der Threads, so werden die Zeiten für jeden Thread einzeln gemessen und optimiert. Somit können sich die Parameter-Konfigurationen der einzelnen Threads unterscheiden.

- TOGETHER: Parallele Optimierungsvariante, die ebenfalls sequentielle Programme unterstützt, kann aber wie die separate Variante langsamer sein. Arbeitet das zu optimierende Programm mit Threads und liegt ein Breakpoint innerhalb der Threads, wird die nächste Parameterkonfiguration auf Basis aller Laufzeiten der Threads gewählt. Somit besitzt jeder Thread zu jeder Zeit die gleiche Parameter-Konfiguration.

Des Weiteren muss angegeben werden, ob im „Main-Thread“ (=Prozess, der das Programm gestartet hat) eine Optimierung stattfindet, dies ist vor allem für die gemeinsame Optimierungsvariante relevant, da diese bei falscher Angabe Fehlverhalten produzieren oder in einem Deadlock enden wird.

Nach Angabe der wichtigsten Informationen können die Breakpoints deklariert werden. Bleibt das zu optimierende Programm an einem Breakpoint hängen, so wird an dieser Stelle eine Optimierung, der in dem Breakpoint angegebenen Parameter und Variablen, vorgenommen. Der Optimierer unterstützt die Optimierung beliebig vieler Stellen ($\hat{=}$ Breakpoints) und Paramtern/Variablen in diesen Stellen. Ein Breakpoint wird durch folgende Attribute spezifiziert:

- Typ: Gibt den Typ des Breakpoints an, wobei insgesamt zwei Arten von Breakpoints wählbar sind. Passiert ein Thread einen Optimizer-Breakpoint,

so wird dieser im Optimierungsvorgang mit einbezogen. Hält ein Thread jedoch bei einem „Zone-Breaker“ an, wird dieser markiert und beim Optimierungsvorgang nicht berücksichtigt. Dabei sollte gewährleistet sein, dass „Zone-Breaker-Threads“ niemals einen Part im Programm betreten, der Optimierungs-Breakpoints enthält. Dies könnte zu unvorhergesehen Race-Konditionen führen.

- Die Lebensdauer gibt an, ob der entsprechende Breakpoint reaktiviert (≙ FOREVER), oder nach einmaliger Verwendung (≙ ONETIME) gelöscht werden soll

- Die Zeilennummer ist optional und kann als Positionsangabe für einen Breakpoint verwendet werden

- Der Key „location“ hat zwei Verwendungszwecke. Ist zuvor eine Zeilennummer angegeben worden, muss dort der Dateiname der Datei spezifiziert werden, für die die Zeilennummmer gültig ist. Alternativ kann „location“ für ein Label verwendet werden, das ebenfalls die Position des Breakpoints festlegt.

- Die Breakpoint-Methode legt die Methode fest, in der der Breakpoint aufzufinden ist.

- Danach können 3 optionale Felder (Start-Temperatur, Alpha-Wert und maximale Optimierungsschritte) in der Yaml-Datei angegeben werden, die alle nur für Simulated-Annealing benötigt werden.

- Als letztes werden in Optimierungs-Breakpoints die Variablen oder Parameter angegeben, die an dem jeweiligen Breakpoint optimiert werden sollen. Diese sind ähnlich wie die Breakpoints selbst in ihrer Zahl nicht begrenzt. Spezifiziert werden die Variablen oder Parameter durch den Namen, wie er im Programmcode selbst vorkommt, einen Typ (PARAMETER, VARIABLE oder GLOBAL)[2]und den Intervallgrenzen (≙ smallest und biggest).

Nachdem nun die Benutzereingabe erklärt wurde, wird im folgenden Kapitel auf die Datenstruktur eingegangen, die unter anderem auch die Nutzerdaten verwaltet.

[2]PARAMETER≙ Funktionsparameter, VARIABLE≙lokale Variable und GLOBAL≙globale Variable

6.2 Datenstruktur

Die Datenstruktur des Optimierers besteht aus drei Hauptkomponenten. Einmal die Breakpoints selbst, die zu optimierenden Parameter/Variablen, folgend als Symbole bezeichnet und die Threads, die nur für die parallelen Optimierungsvarianten relevant sind. Jeder dieser Komponenten wird in der Datenstruktur durch ein Struct gespeichert, deren Aufbau und Organisation im Folgenden behandelt wird. Begonnen wird mit der Verwaltung der Breakpoints. Das Struct der Breakpoints (vgl. Abb. 6.2) speichert neben den angegebenen Informationen (Typ, Lebensdauer, Label-Name, Methoden-name, Zeilennummer und Simualted Annealing bezogene Informationen) auch noch die Speicheradresse des Breakpoints, die für das Setzen des Breakpoints über ptrace benötigt wird. Für das Löschen des Breakpoints wird die originale Instruktion ebenfalls gesichert. Zudem müssen Informationen über den Optimierungszustand vorhanden sein, die angeben, wie viele Konfigurationen noch zu testen sind, oder ob die Optimierung für diesen Breakpoint bereits ein Ende gefunden hat. Zudem werden noch der Start- und der Endzeitpunkt für die Laufzeitberechnung der Konfiguration in den Breakpoints gespeichert. Den wichtigsten Teil eines Breakpoints bildet die Menge der zu optimierenden Parameter und Variablen, die in einer Liste von Symbolen gespeichert werden.

Jedes Symbol (vgl. Abb. 6.2) speichert die vom Nutzer angegebenen Informationen. Diese sind der Name, der Typ und die Intervallgrenzen, wobei sich die Variablen typ (für DWARF-TAG) und is_local aus dem angegebenen Typ in der Konfigurationsdatei ableiten. Des Weiteren muss die Speicheradresse eines Symbols gespeichert werden und ob diese bereits berechnet wurde. Die Arrays change_values und times werden verwendet, um die Werte des Symbols und die dazu gemessenen Zeiten zu speichern.

Da der Optimierer das Ziel hat Parameter-Konfigurationen zu Optimieren, dürfen die Werte der Symbole eines Breakpoints nicht einzeln betrachtet werden, sondern als gesamte Konfiguration. Hierfür ein kleines Beispiel: Es existiert ein Breakpoint b und an diesem sollen die drei Variablen $v_1, v_2, v_3 \in V$ optimiert werden. Wird nun der Breakpoint b passiert, so bilden die Werte der Variablen $v_1 = 1, v2 = 1$ und $v_3 = 1$ eine Konfiguration $c_1 = (1, 1, 1)$ aus dem Konfigurationsraum $C \subseteq V^3$. Nun hat der Suchalgorithmus die Menge der Nachbarkonfigurationen N bestimmt, die als nächstes getestet werden sollen. Diese sind $c_2 = (0, 1, 1), c_3 = (2, 1, 1), c_4 = (1, 0, 1), c_5 = (1, 2, 1), c_6 = (1, 1, 0)$ und $c_7 = (1, 1, 2)$ und bilden die Elemente der Menge $N = \{c_2, c_3, c_4, c_5, c_6, c_7\}$. Die Speicherung der Konfigurationen für den Breakpoint b wird durch die folgende Abbildung 6.3 veranschaulicht.

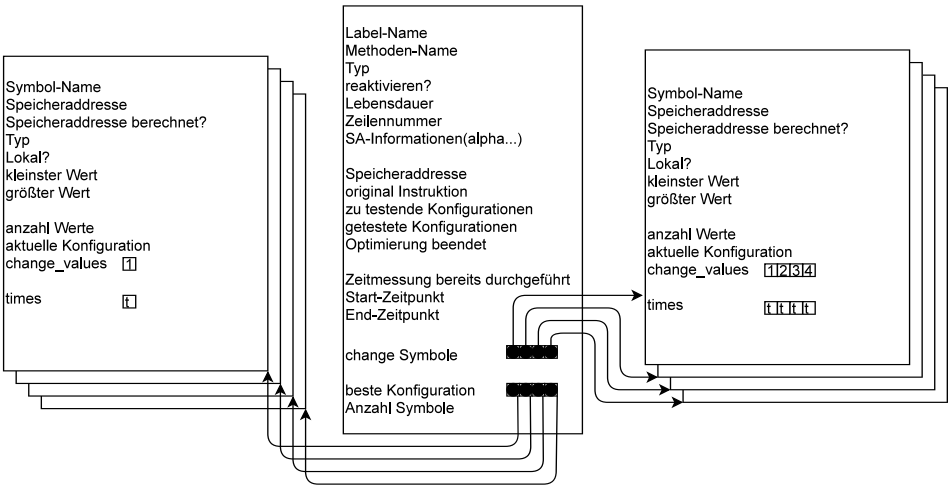

Abb. 6.2: Schematische Darstellung eines Breakpoint-Struct (mittig) und zwei Symbol-Structs

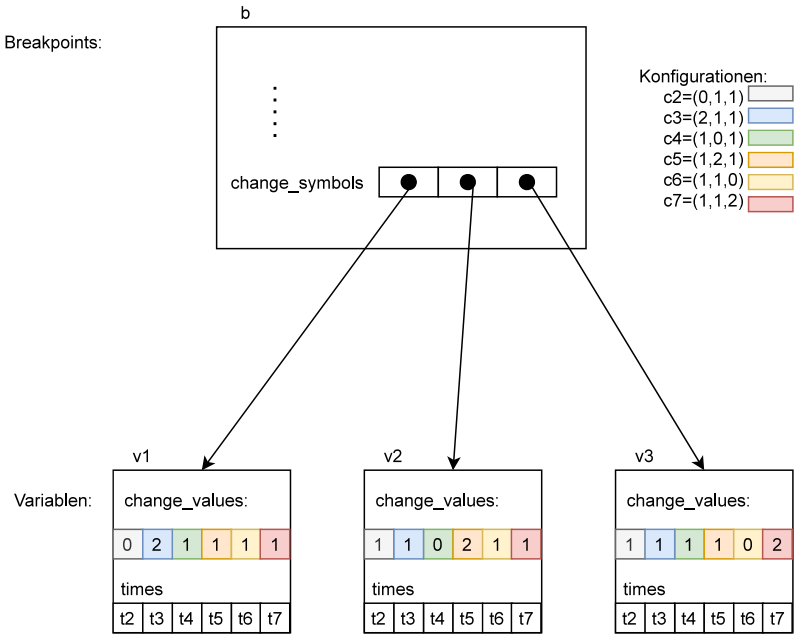

Abb. 6.3: Schematische Darstellung der Speicherung der Konfigurationen c_i, mit den zugehörigen Zeiten t_i, in den times-Feldern der Symbole, in einem Breakpoint-Struct

Werden Threads verwendet, verwaltet jeder Thread-Struct die Thread-Id und eine Liste aller Breakpoints. Des Weiteren wird der Typ des Threads gespeichert, für den Fall, dass es sich um einen Zone-Breaker-Thread handelt und dieser beim Optimierungsvorgang ignoriert werden kann.

6.3 Optimierungsvarianten

6.3.1 Sequentielle-Optimierung

6.3.1.1 Zielsetzung

Das Ziel der sequentiellen Optimierungsvariante besteht darin, die individuellen Parameter-Konfigurationen einer Menge von Breakpoints (=Optimierungsstellen) zu optimieren. Dabei müssen sich alle Optimierungspunkte in sequentiellen Programmteilen befinden. Sollten Threads vom zu optimierenden Programm verwendet werden, so können diese nicht erkannt und berücksichtigt werden.

6.3.1.2 Zustandsautomat

Den Kern dieser Optimierungsvariante bildet ein Zustandsautomat, der im folgenden anhand der Abb. 6.4 und eines fiktiven Beispielablaufs näher beschrieben wird.

Wird nach dem setzen des Breakpoints dieser vom zu optimierenden Programm das erste Mal passiert, so befindet sich der Automat im ersten Zustand. Dort wird als erstes eine Zeitmessung vorgenommen, um den Endzeitpunkt zu erhalten. Danach wird geprüft, ob es sich bei der Stopp-Speicheradresse um einen bekannten Breakpoint handelt. Dies kann ermittelt werden, indem man den aktuellen PC-1 des Prozesses, mit allen Breakpoint-Adressen in der Liste abgleicht. Da es sich hierbei um den initialen Aufruf eines bekannten Breakpoints handelt ($\rightarrow$ Optimierung ist nicht beendet), liegen weder Informationen über die Initial-Konfiguration der Symbole, noch über den Startzeitpunkt der Zeitmessung vor. Aus diesem Grund wird auch noch keine Zeit für die aktuelle Konfiguration in jedem times-Feld der Symbole eingetragen. Anschließend wird der zuvor gemessene Endzeitpunkt als Startzeitpunkt gespeichert. Aufgrund der fehlenden Informationen über die aktuelle Konfiguration der Symbole, werden nun die Speicheradressen der Variablen aus dem DWARF-Offset berechnet und deren aktuellen Werte als Initial-Konfiguration in den change_values gespeichert.

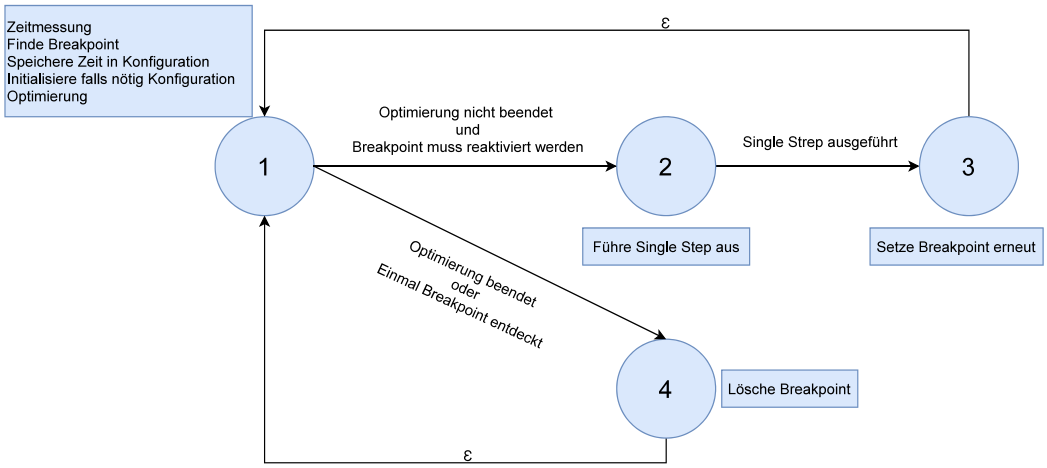

Abb. 6.4: Zustandsautomat, der die Arbeitsweise der sequentiellen Optimierung verbildlicht

Danach wird der bedingungslose Übergang in den zweiten Zustand gewählt. Dort wird der Breakpoint entfernt, die einzelne Breakpoint-Instruktion (ptrace-singlestep) ausgeführt und auf die Erfüllung der Übergangsbedingung für den dritten Zustand gewartet. Wird das SIGTRAP-Signal von ptrace-singlestep empfangen, so wird der Breakpoint erneut gesetzt, das Programm fortgeführt und in den ersten Zustand gewechselt (ohne dessen Aktionen auszuführen). Tritt nun erneut ein SIGTRAP-Signal auf, wird wieder der Endzeitpunkt gemessen und die resultierende Zeit in die entsprechenden times-Felder der aktuellen Konfiguration eingetragen. Folgend wird wieder der Startzeitpunkt auf den gemessen Endzeitpunkt gesetzt und es kann mit der Optimierung der Konfiguration gestartet werden, die im folgenden Kapitel 6.3.1.3 behandelt wird. Ist nach dem Aufruf des Suchalgorithmus der Optimierungsvorgang beendet, so wird in den vierten Zustand gewechselt und der Breakpoint dauerhaft gelöscht.

6.3.1.3 Optimierung

Die Optimierung der Konfigurationen übernimmt ein Suchalgorithmus, der alle Konfigurationen und Informationen über den Optimierungsfortschritt übergeben bekommt. Dieser wertet aus, ob alle angegebenen Konfigurationen getestet wurden, ob die Optimierung bereits beendet ist, oder ob die neuen Test-Konfigurationen abzuarbeiten sind. Werden neue Test-Konfigurationen erzeugt, so teilt der Suchal-

gorithmus über ein „write_back-Flag“ mit, ob die neuen Konfigurationen die alten Konfigurationen in den change_value-Feldern der Symbole überschrieben werden sollen. Ist der Suchalgorithmus zu einem Ergebnis gekommen, wird die aktuell markierte Konfiguration in den enstprechenden Variablen/Parametern des Programms geschrieben. Gibt der Suchalgorithmus an, dass die Optimierung beendet ist, wird die beste zuvor gefundene Konfiguration in die entsprechenden Variablen des Programms gespeichert.

Aktuell sind zwei Suchalgorithmen implementiert: Hillclimb und Simulated Annealing. Im Fall von Hillclimb werden alle Nachbarkonfigurationen getestet und dann die Beste der getesteten Konfigurationen gewählt. Dies wird solange ausgeführt, bis alle Nachbarkonfigurationen schlechter sind, als die aktuell gewählte Konfiguration[vgl. Wikc]. Bei Simulated Annealing wird eine zufällig ausgewählte Nachbarkonfiguration getestet und für den Fall, dass diese besser ist als die aktuelle Konfiguration, wird diese als beste Konfiguration vermerkt. Ist die zufällig gewählte Konfiguration schlechter, so wird diese mit einer gewissen Wahrscheinlichkeit, die abhängig von der aktuellen Temperatur ist (kleine Temperatur $\rightarrow$ geringere Wahrscheinlichkeit & höhere Temperatur $\rightarrow$ höhere Wahrscheinlichkeit), als neue beste Konfiguration gewählt. Diese Schritte werden solange ausgeführt, bis die maximale Optimierungsschrittanzahl erreicht ist[vgl. MAR].

6.3.2 Paralle-Optimierung

6.3.2.1 Zielsetzung

Das Ziel dieser Variante ist es, eine Optimierung in parallelen Programmabschnitten zu ermöglichen. Dabei wird jeder Optimizer-Thread ($\widehat{=}$ kein Zone-Breaker-Thread) einzeln betrachtet und optimiert, was letzlich dazu führen kann, dass sich die Parameterkonfigurationen der Threads unterscheiden, obwohl diese am gleichen Punkt optimiert werden. Des Weiteren ist es möglich Threads aus dem Optimierungsverfahren auszuschließen. Dies wird mit „Zone-Breaker-Breakpoints“ umgesetzt. Das heißt, passiert ein Thread ein „Zone-Breaker-Breakpoint“, so wird dieser als „Zone-Breaker-Thread“ markiert und beim Optimierungsvorgang nicht berücksichtigt. Dies hat vor allem Auswirkungen auf das Breakpoint-Handling (vgl. 6.3.2.4), was dazu führt, dass Zone-Breaker-Threads unter keinen Umständen die Breakpoint-Instruktionen ausführen dürfen. Da nun die Zielsetzung der Variante geklärt wurde, kann im folgenden der Optimierungsablauf anhand eines Zustandsautomaten ausgeführt werden.

6.3.2.2 Zustandsautomat

Auch wie in den Kapiteln zuvor wird dieser Zustandsautomat ebenfalls anhand einer Graphik (vgl. Abb 6.5) erklärt.

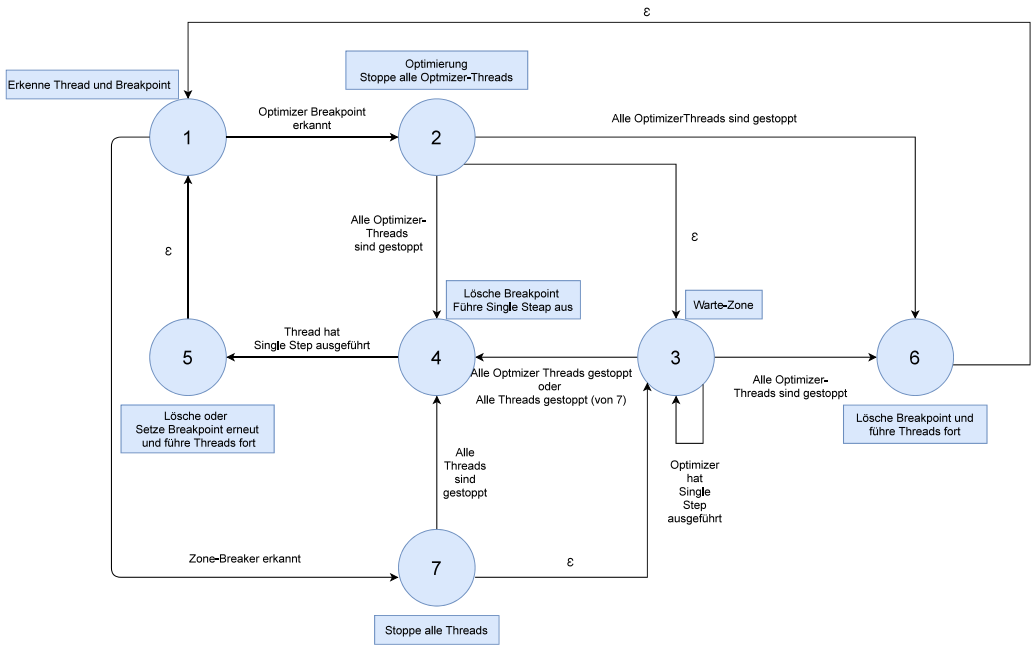

Abb. 6.5: Abbildung eines Zustandsautomaten, der den Ablauf der parallelen Optimierung veranschaulicht

Zu Beginn befindet sich der Automat im ersten Zustand und es tritt ein Breakpointereignis ein. Da der Programm-Prozess beliebig viele Threads erzeugen kann, muss an dieser Stelle geprüft werden, welche Threads angehalten worden sind. Wenn sie angehalten sind, muss eine Zeitmessung für den Endzeitpunkt durchgeführt werden. Alle angehaltenen Threads werden in einer Prioritätswarteschlange eingereiht und nach ihrem Anhaltezeitpunkt sortiert. Somit wird der Thread als erstes abgearbeitet, der als erstes angehalten wurde. Für diesen Thread muss dann festgestellt werden, ob dieser an einem Optimizer-Breakpoint, an einem Zone-Breaker-Breakpoint angehalten ist oder ob ein neuer Thread erzeugt wurde. Die Breakpoint-Erkennung funktioniert analog wie bei der sequentiellen Variante, durch einen Vergleich des PC-1 und den Speicheradressen aller bekannten Breakpoints in der Liste. All diese Aufgaben werden von einer Methode übernommen, die entsprechende Codes zurückgibt, die angeben, welches der zuvor genannten Ereignisse eingetreten ist.

Der erste Fall wäre, dass ein neuer Thread erkannt wird. Dieser wird nach der Erkennung (siehe 6.3.2.4) in die Thread-Liste eingefügt und durch eine ptrace-cont

Anweisung fortgeführt[vgl. Webb].

Der zweite Fall wäre, dass ein Optimizer-Thread erkannt wird (Erkennungsprozess analog zur sequentieller Variante). Dafür wechselt der Zustandsautomat in den zweiten Zustand und speichert dort dann die Laufzeit der Konfiguration in die entsprechenden times-felder der Symbole des Threads. Für den Fall, dass es sich um das erste Passieren des Breakpoints handelt, wird auf die Laufzeitspeicherung der Konfiguration verzichtet, außerdem werden analog zur sequentiellen Variante, die Speicheradressen der Variablen berechnet und mit Hilfe dieser, die initiale Konfiguration ausgelesen. Sind beim Betreten des Zustands Zeitmessungen vorhanden, so wird die Optimierung gemäß Kapitel 6.3.2.4 ausgeführt und die daraus resultierende Konfiguration in den Variablen/Parametern des Programms gespeichert. Um nun die Löschung oder die Reaktivierung des Breakpoints vorzubereiten, werden alle Optimizer-Threads und der Main-Thread gestoppt. Zone-Breaker-Threads werden hierbei nicht gestoppt, da diese existieren, um den Programmablauf durch das Breakpoint-Handling (bspw. stoppen relevanter Threads anstatt allen) nicht unnötig zu verlangsamen. Dafür wechselt der Automat in den dritten Zustand, der solange aktiv bleibt, bis alle betreffenden Threads gestoppt worden sind. Ist die Lebensdauer des Breakpoints begrenzt, wechselt der Zustandsautomat in den sechsten Zustand, entfernt den Breakpoint und führt alle Threads fort, die durch die ptrace-interrupt Anweisung gestoppt worden sind[vgl. Webb]. Nach der erfolgreichen Löschung kann wieder in den ersten Zustand gewechselt und auf ein Breakpointereignis (SIGTRAP-Signal) gewartet werden. Soll der Breakpoint jedoch reaktiviert werden, so wird, nachdem alle betreffenden Threads gestoppt sind, in den vierten Zustand gewechselt. Dort wird der Breakpoint-Opcode, über den Main-Thread, aus der Breakpoint-Instruktion gelöscht und diese dann durch eine ptrace-singlestep-Anweisung ausgeführt[vgl. Webb]. Wurde die Instruktion ausgeführt, wechselt der Automat in den fünften Zustand. Hierbei wird als erstes überprüft, ob die Optimierung des Breakpoints abgeschlossen wurde. Dies lässt sich an den entsprechenden Breakpoint-Structs aller Threads erkennen. Ist die Optimierung des Breakpoints für alle Threads beendet, so wird der Breakpoint gelöscht, alle betreffenden Threads werden fortgeführt und der Automat geht abschließend in den ersten Zustand über. Ist die Optimierung nicht beendet (mind. ein Thread hat die Optimierung nicht beendet), muss der Breakpoint wieder aktiviert werden, indem er vom Main-Thread gesetzt wird. Ist das erledigt, werden alle betreffenden Threads fortgeführt und es kann wieder in den ersten Zustand gewechselt werden.

Der letzte Fall wäre, dass die Methode im ersten Zustand einen Zone-Breaker-Breakpoint erkannt hat. Dafür wechselt der Automat in den siebten Zustand, dort werden dann alle Threads durch eine ptrace-interrupt-Anweisung angehalten[vgl.

Webb]. Danach wird analog zu den Optimizer-Breakpoints im dritten Zustand gewartet, bis alle Threads gestoppt sind. Nachdem alle Threads gestoppt sind, kann der Breakpoint gefahrlos gelöscht und dessen eigentliche Instruktion ausgeführt werden (ptrace-singlestep [vgl. Webb]). Ist dies erledigt, werden Zone-Breaker-Breakpoints automatisch im fünften Zustand erneut gesetzt und alle vorher nicht gestoppten Threads werden fortgesetzt. Als letztes wird abschließend erneut in den ersten Zustand gewechselt.

6.3.2.3 Optimierung

Wie bereits aus dem Kapitel 6.2 hervorgeht, besitzt jeder Thread seine eigene Breakpoint-Liste. Das heißt, jeder einzelne Thread hat Kenntnis über jeden existierenden Breakpoint, aber die Parameter-Konfigurationen können somit für jeden Thread individuell gewählt werden. Die Optimierung durch den Suchalgorithmus unterscheidet sich nur in der Hinsicht von der sequentiellen Optimierungsvariante, dass nicht nur ein Thread/Prozess optimiert wird, sondern potentiell beliebig viele. Des Weiteren werden Zone-Breaker-Threads nicht optimiert. Diese haben den Sinn, Threads nicht unnötig zu unterbrechen, die eventuell stark rechenintensive Programmteile ausführen. Mehr dazu findet sich im nächsten Kapitel.

6.3.2.4 Threaderkennung und Breakpointhandling

Das Kapitel dient zur Erklärung, wie neue Threads erkannt werden können und weshalb bei der Modifikation der Breakpoints in parallelen Programmteilen Vorsicht geboten ist.

Wie bereits im Kapitel 4 erwähnt, verwendet beispielsweise die Pthread-Library für die Thread-Erzeugung einen clone-Befehl, welche durch ptrace erkannt werden kann[vgl. Ben18; Webb]. Wird ein clone-Aufruf erkannt, so wird der Thread-Erzeuger angehalten, der erzeugte Thread automatisch attached und ebenfalls bei der ersten Instruktion angehalten[vgl. Webb]. Da der erzeugende Thread, oder Prozess, beim Erzeugen eines neuen Threads gestoppt wird, ist die Speicheradresse, an der angehalten wird, nicht bekannt. Das heißt, das beim Vergleich dieser Speicheradresse, mit allen Speicheradressen der bekannten Breakpoints, keine Übereinstimmung gefunden werden kann. Nachdem nun festgestellt wurde, dass ein neuer Thread gefunden wurde, muss dessen Thread-Id in Erfahrung gebracht werden. Dies ist für die Steuerung des Threads über ptrace essentiell. Die Thread-Id des neu gefundenen Threads kann über eine ptrace-getmessage-Anweisung in Erfahrung gebracht werden[vgl. Webb]. Um den Optimierer resistenter gegen potentielle Fehlinterpre-

tationen zu machen, werden dort neue Threads und dessen Thread-Ids durch das Parsen des Proc-Ordners gefunden. Dies wird aber nur in zwei Fällen durchgeführt. Der erste Fall wäre, dass die Methode im ersten Zustand den Code zurückliefert, der angibt, dass ein neuer Thread erkannt wurde. Ebenso führt die Beendigung eines Threads dazu, dass der Ordner geparst wird, da dort lediglich überprüft wird, ob sich ein Thread oder Prozess, nach dem Absenden eines Signals beendet hat und folglich dessen Thread-Id unbekannt ist. Im gleichen Zug wird die Liste der aktiven Threads/Prozesse des Programms abgeglichen, ob ein Thread übersehen wurde und falls dies zutrifft, wird dieser der Thread-Liste hinzugefügt.

Für das Handling der Breakpoints (setzen oder löschen), müssen einige Eigenschaften erfüllt sein, um einen korrekten Ablauf des Programms zu gewährleisten. Wird beispielsweise ein Breakpoint gesetzt oder gelöscht, während ein anderer Thread exakt zu diesem Zeitpunkt dessen Instruktion ausführen möchte, so kann es passieren, dass dieser Thread nicht an dem Breakpoint anhält und das Ergebnis der Optimierung verfälscht. Das bedeutet, um derartige Race-Konditionen zu vermeiden, muss ein wechselseitiger Ausschluss der Threads, beim Setzen oder Löschen, der Breakpoints gewährleistet sein. Dies wird im Zustandsautomat durch das Stoppen aller Threads erreicht, die möglicherweise die Breakpointinstruktion ausführen könnten. Zone-Breaker bilden hierbei eine Ausnahme, da die Annahme gilt, dass ein Zone-Breaker-Thread kein Programmteil „betreten“ kann, der einen Optimizer-Breakpoint enthält. Aus diesem Grund kann dieser gefahrlos weiter ausgeführt werden, falls ein Optimizer-Breakpoint modifiziert wird, da dessen Instruktion unter keinen Umständen von dem Zone-Breaker-Thread ausgeführt werden kann und somit das Anhalten des Zone-Breaker-Threads unnötig wäre. Aus diesem Grund werden auch bei der Modifikation von Zone-Breaker-Breakpoints alle Threads angehalten, da die Annahme nicht für Zone-Breaker-Breakpoints gilt. Für den Fall, dass rechenintensive Programmteile existieren, die keine Optimizer-Breakpoints enthalten und diese niemals erreichen würden, könnten Zone-Breaker-Breakpoints verwendet werden, um alle Threads nicht unnötig zu stoppen, die diesen Zone-Breaker-Breakpoint passieren.

6.3.3 Parallel-Gemeinsame-Optimierung

6.3.3.1 Zielsetzung

Das Ziel dieser Optimierungsvariante ist es, ebenfalls Parameter-Konfigurationen in parallelen Programmen zu optimieren. Nur dass diese Variante, im Gegensatz zur parallelen-Optimierung, alle Threads nach dem Optimierungsvorgang eine gemeinsame Parameter-Konfiguration besitzen. Um diese Optimierungsvariante anwenden

zu können, muss sichergestellt sein, dass alle Optimizer-Threads die Breakpoints in der gleichen Reihenfolge betreten. Des Weiteren muss angegeben werden, ob der Main-Thread (=Prozess, der das Programm selbst gestartet hat) ebenfalls die Breakpoints betritt, oder nicht. Dies hat den Grund, dass die Breakpoints in dieser Variante, wie eine Art Thread-Barrier wirken. Das heißt, es wird solange an einem Breakpoint gewartet, bis alle Optimizer-Threads an diesem angehalten sind. Für den Fall, dass Threads nicht an den Barriers halten werden, so sind diese durch einen, oder mehrere Zone-Breaker-Breakpoints, als Zone-Breaker zu markieren. Werden in dieser Hinsicht Fehler begangen, so kann der Optimierer in einem Deadlock enden.

6.3.3.2 Zustandsautomat

Der Zustandsautomat (vgl. Abb 6.6) zeigt die Arbeitsweise dieser Optimierungsvariante und wird im Folgenden beschrieben.

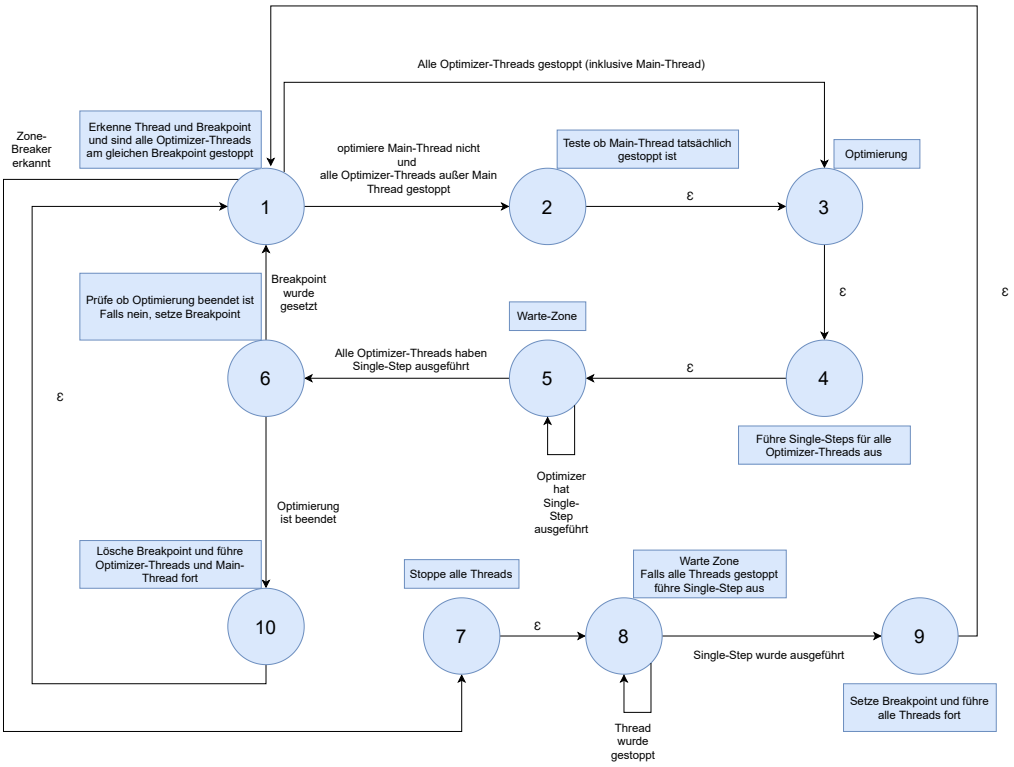

Abb. 6.6: Abbildung eines Zustandsautomaten, der den Ablauf der parallel-gemeinsamen Optimierung veranschaulicht

Nachdem alle Breakpoints gesetzt worden sind, befindet sich der Zustandsautomat im ersten Zustand und es kann auf ein Breakpoint-Ereignis (=SIGTRAP-Signal) gewartet werden. Trifft nun das SIGTRAP-Signal ein, so wird als erstes eine Methode aufgerufen, die feststellt, welche Threads neu angehalten sind. Für die Threads wird anschließend eine Zeitmessung durchgeführt, an welchem Optimizer-Breakpoint

diese gestoppt wurden, ob es sich um einen Zone-Breaker-Thread handelt, oder ob ein neuer Thread erzeugt wurde. Die Methode gibt einen Code zurück, der das eingetretene Ereignis repräsentiert. Der einzige Fall, der nicht eintreten darf und zu einem Abbruch des Programms führt, ist, dass zwei Threads zeitgleich an unterschiedlichen Optimizer-Breakpoints angehalten haben. Mehr dazu findet sich im Kapitel 6.3.3.4. Die Abläufe aller anderen legitimen Fälle werden im folgenden anhand des Automaten erklärt.

Erkennt die Methode im ersten Zustand, dass ein neuer Thread erkannt wird, so hat dieses Ereignis Vorrang und es wird sofort der entsprechende Code zurückgegeben. Nachdem der Automat den Code erhalten hat, wird der Thread-Erzeuger, sowie der Erzeugte-Thread fortgesetzt und der neu erzeugte Thread in die Thread-Liste eingefügt.

Hat nun ein Thread einen Optimizer-Breakpoint erreicht und an diesem gestoppt, wird geprüft, ob weitere Threads an diesem gehalten haben, ist dem so, wird der „Warte-Counter“ um eins inkrementiert. Hat noch kein Thread zuvor an einem Breakpoint gestoppt (z.B.: erster Breakpoint oder nach Optimierungsvorgang), wird dieser Breakpoint als aktuell vermerkt und der „Warte-Counter“ ebenfalls um eins inkrementiert. Nachdem alle gestoppten Threads, wie eben beschrieben, abgearbeitet wurden, wird überprüft, ob alle Optimizer-Threads am aktuellen Breakpoint angehalten haben. Trifft das nicht zu, verbleibt der Automat im ersten Zustand und es wird auf das nächste Breakpoint-Ereignis gewartet. Haben alle Optimizer-Threads am aktuellen Breakpoint angehalten, muss überprüft werden, ob der Main-Thread ebenfalls darunter ist. Falls nicht wird dieser durch eine ptrace-interrupt-Anweisung gestoppt[vgl. Webb] und es wird im zweiten Zustand gewartet bis dieser angehalten wurde. Ist der Main-Thread gestoppt, kann vom ersten, oder vom zweiten Zustand aus, in den dritten Zustand übergegangen werden. Dort wird als erstes überprüft, ob es sich um das erste Betreten dieses Breakpoints handelt. Falls ja, werden die Speicheradressen der Symbole berechnet und die Initial-Konfiguration ausgelesen. Sind bereits Messdaten vorhanden, kann mit der Optimierung gemäß Kapitel 6.3.3.3 begonnen werden. Nachdem diese abgeschlossen wurde, muss der Breakpoint entfernt werden. Dafür müssen keine Threads mehr gestoppt werden, da alle relevanten Threads am Breakpoint angehalten haben. Deshalb wird der Opcode des Breakpoints durch den Main-Thread entfernt und alle Optimizer-Threads führen diese durch eine ptrace-singlestep-Anweisung (vgl. [Webb]) aus. Um auf alle SIGTRAP-Signale der Optimizer-Threads zu warten, die angeben, dass die Instruktion ausgeführt wurde, wird in den fünften Zustand gewechselt. Haben alle betreffenden Threads die Instruktion ausgeführt, kann anhand des Optimierungsfortschritts des Breakpoints entschieden werden, ob dieser gelöscht, oder neu gesetzt werden muss. Hierfür

wechselt der Automat in den sechsten Zustand. Falls die Optimierung beendet ist, geht der Automat in den zehnten Zustand über, wo dann der Breakpoint gelöscht wird, alle Optimizer-Threads fortgesetzt werden und abschließend in den ersten Zustand gewechselt wird. Ist die Optimierung des Breakpoints nicht abgeschlossen, wird dieser im sechsten Zustand erneut gesetzt, der Startzeitpunkt erfasst und alle Optimizer-Threads fortgeführt. Abschließend wird der aktuelle Breakpoint demarkiert und es wird in den ersten Zustand gewechselt.

Handelt es sich bei einem angehaltenen Thread um einen Zone-Breaker, gibt die Methode im ersten Zustand den entsprechenden Code zurück. Dann wechselt der Zustandsautomat in den siebten Zustand, der alle nicht gestoppten Threads anhält. Anschließend wird im achten Zustand solange verblieben, bis alle Threads angehalten sind. Ist dies nun der Fall, wird dort der Breakpoint-Opcode, vom Main-Thread, entfernt und der Thread kann dessen Instruktion ausführen. Wurde die Instruktion ausgeführt, kann der Automat im neunten Zustand den Zone-Breaker-Breakpoint, über den Main-Thread, erneut setzen, alle zuvor angehaltenen Threads fortsetzen und abschließend wieder in den ersten Zustand übergehen.

Da jetzt geklärt wurde wie der Zustandsautomat funktioniert, kann anschließend der Optimierungsprozess näher erläutert werden.

6.3.3.3 Optimierung

Die Optimierung wird analog zu den anderen Varianten, durch einen Suchalgorithmus durchgeführt, der entscheidet, ob die Optimierung beendet ist, neue Konfigurationen gestestet werden müssen, oder noch Zeitmessungen für die vorhandenen Konfigurationen ausstehen. Da es in dieser Variante darum geht, die beste Konfiguration für alle Threads zu finden, wird vor der Ausführung des Suchalgorithmus eine „Meta-Konfiguration“ gebildet. Diese besteht aus allen Konfigurationen der Symbole, nur dass die Zeiten aller Optimizer-Threads aufaddiert werden. So erhält man die gesuchte Information, welche Parameter-Konfiguration auf allen Optimizer-Threads die geringste Ausführungszeit benötigt hat. Falls der Suchalgorithmus neue Konfigurationen zum Testen zurückgibt, wird durch das „write_back-Flag“ signalisiert, dass diese in die Symbole zurückgespeichert werden müssen. Die Ausführungszeiten werden in diesem Fall auf das Maximum gesetzt, da die Messergebnisse für die einzelnen Threads noch ausstehen. Der Optimierungsfortschritt wird ausschließlich über den Main-Thread verwaltet.

6.3.3.4 Threaderkennung und Breakpointhandling

Die Thread-Erkennung funktioniert analog zur parallelen Optimierungsvariante und wird durch Threads erkannt, die an unbekannten Adressen angehalten sind.

Das Breakpointhandling unterscheidet sich etwas von den Varianten zuvor. Da in dieser Variante die Breakpoints wie eine Art Thread-Barrier funktionieren, wird solange auf die Optimizer-Threads an den Breakpoints gewartet, bis alle an diesem angehalten sind. Für den Main-Thread muss explizit angegeben werden, ob dieser an den Optimizer-Breakpoints hält, da dieser für die Modifikation der Breakpoints verwendet wird und sonst separat angehalten werden müsste. Sollte es Programmteile geben, die niemals die Optimizer-Breakpoints erreichen, würde ein Deadlock entstehen. Aus diesem Grund wurden die Zone-Breaker eingeführt, die es erlauben, Threads und somit auch Programmteile zu markieren, die niemals an Thread-Barrieren (=Optimizer-Breakpoints) anhalten würden. Diese Zone-Breaker-Breakpoints bilden einen Kompromiss, der es erlaubt die Optimierungsvariante anzuwenden, selbst wenn nicht alle Threads optimierungsrelevant sind. Auch hier gilt, dass Zone-Breaker-Threads niemals Programmteile ausführen dürfen, die Optimizer-Breakpoints enthalten.

7 Vergleich mit anderen Autotunern

Dieses Kapitel vergleicht, den in dieser Arbeit thematisierten Ansatz, mit zwei bereits vorhandenen Autotunern. Dabei werden auf die bereits implementierten Funktionen, auf mögliche Erweiterungen und auf die Grenzen des Optimierers eingegangen.

Der erste der beiden Vergleichs-Autotuner besitzt den Name „Active Harmony“. Alle Informationen über diesen Autotuner stammen aus der Quelle [AT11]. Active Harmony verwendet Codegenerierung, um neue Codevarianten zu testen. Nach der Generierung werden die Codevarianten als shared-library kompiliert und in das Programm eingebunden. Dabei können zum Beispiel neu erzeugte Funktionsvarianten, durch ein simples Tauschen der Funktionspointer, getestet werden. Dadurch ist es beispielsweise möglich, Schleifen in Funktionen, durch loop-unrolling zu optimieren. Active Harmony unterstützt die Optimierung von MPI-, sowie OpenMP-Anwendungen und libraries. Des Weiteren können Abhängigkeiten zwischen den zu optimierenden Parametern berücksichtigt werden.

Der zweite Auto-Tuner in diesem Vergleich hat den Namen „ATF“ (=Auto-Tuning Framework) und alle Informationen über diesen Auto-Tuner sind aus der Quelle [RA18] entnommen. Dieser unterstützt die Optimierung von Parameter, die Abhängigkeiten untereinander aufweisen. Des Weiteren können Programme unabhängig von ihrer Programmiersprache optimiert werden, dafür stellt der Auto-Tuner drei Suchalgorithmen zur Verfügung, die eine einheitliche Schnittstelle implementieren. Dies führt dazu, dass die Suchalgorithmen erweitert werden können. Eine weitere Besonderheit des Autotuners bildet die Möglichkeit, Anwendungen hinsichtlich der Energieeffizienz zu optimieren. Angewendet wird der Autotuner durch einfache Direktiven im Quellcode des zu optimierenden Programms.

Dieser Ansatz verwendet Breakpoints für den Optimierungsprozess, der während der Laufzeit des Programms stattfindet. Um eine Laufzeitmessung zu ermöglichen, muss der Optimierungs-Breakpoint mehrmals passiert werden. Dies führt zwangsläufig dazu, dass der zu optimierende Programmteil in einer Schleife ausgeführt werden muss, was eine Beschränkung der optimierbaren Anwendungen darstellt.

Loop-Unrolling, wie es Active Harmony unterstützt[vgl. AT11], kann in dem Optimierer der Arbeit nur durch einen Codegenerierer umgesetzt werden und ist mit Debugger-Werkzeugen nicht realisierbar. Was jedoch möglich wäre, sind bereits implementierte und kompilierte Funktionsvarianten des zu optimierenden Programms, durch Ändern des Funktionspointers, im Optimierer zu implementieren. Für die Implementierung dieser Optimierungsmöglichkeit, müssten die Funktionsvarianten allerdings die gleiche Parameter-Schnittstelle besitzen und dem Optimierer durch die Funktionsnamen, beispielsweise in der YAML-Konfigurationsdatei, mitgeteilt werden. Die Suchalgorithmen können erweitert werden, wenn diese die Parameter-Schnittstelle implementieren und Änderungen der Konfigurationen durch das „write_back-Flag“ angeben. Abhängigkeiten zwischen den Variablen werden in der aktuellen Version des Optimierers nicht unterstützt, könnten aber implementiert werden. MPI-Anwendungen können nicht optimiert werden, da diese bei der Ausführung unabhängige Prozesse erzeugen[vgl. Pro20]. Dies führt dazu, dass der Optimierer momentan nicht in der Lage ist, die von MPI erzeugten Prozesse zu erkennen und mit ptrace zu verbinden. Dafür müssten dem Optimierer die PID's der erzeugten Prozesse in irgendeiner Form mitgeteilt und die Optimierung der Prozesse synchronisiert werden. Den größten Vorteil dieses Ansatzes bildet die Nutzerschnittstelle. Für die Verwendung des Optimierers sind zwar einige Kenntnisse über das zu optimierende Programm notwendig (z.B. Variablen-Namen, Variablentyp, Zeilennummer des Breakpoints, ...), aber es müssen kaum Änderungen am Programmiercode der Anwendung vorgenommen werden. Es sei denn, es werden Labels zur Positionsbestimmung der Breakpoints verwendet.

8 Laufzeitmessungen

Dieses Kapitel dient dazu, die Funktionalität des Optimierers zu zeigen. Dabei werden zwei Anwendungen getestet, die jeweils 100 Mal eine sequentielle Matrixmultiplikation ausführen.

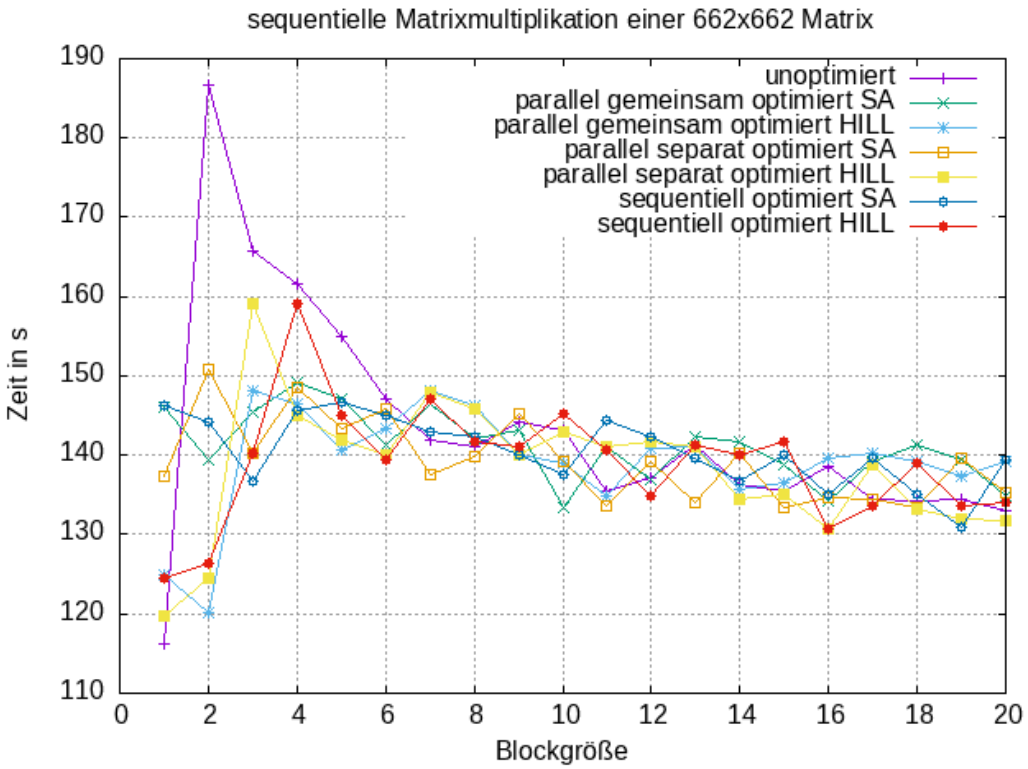

Abb. 8.1: Laufzeitmessung einer sequentiellen Matrixmultiplikation, die 100 mal ausgeführt wird

Die erste Anwendung (Abb 8.1) führt 100 mal eine sequentielle Matrixmultiplikation aus, dabei befindet sich der zu optimierende Teil ebenfalls in einem sequentiellen Programmteil. Dieser Testfall demonstriert, dass Anwendungen getunt werden können, deren zu optimierende Teil sich im sequentiellen befindet. Dazu wird zum Vergleich das unoptimierte Programm mit den Blockgrößen von 1 (ohne Tiling) bis 20 ausgeführt. Dabei hat sich gezeigt, dass eine Blockgröße von 2 die schlechteste Laufzeit besitzt und dass das Optimum bei einer Blockgröße von 1 (ohne Tiling) liegt. Des Weiteren lässt sich erkennen, dass mit steigender Blockgröße (ab

Blockgröße 2) die Laufzeit relativ stark sinkt. Ab einer Blockgröße von 7 sinkt die Laufzeit langsamer und beginnt zudem noch zu schwanken. Für die optimierten Vergleichsmessungen wurde das Programm ebenfalls, mit der für den jeweiligen Messungspunkt angegebenen Blockgröße gestartet. Dann wurde durch den Optimierer, ausgehend von der Initialkonfiguration (=Blockgröße auf x-Achse für den Messungspunkt), durch einen der beiden Suchalgorithmen, Simulated Annealing (=SA) oder Hillclimb (=HILL), die zu testenden Nachbarkonfigurationen gewählt. Für eine Erklärung der Arbeitsweise der beiden Algorithmen kann das Kapitel 6.3.1.3 oder die beiden Quellen ([Wikc; MAR]) zu Rate gezogen werden. Dabei hat sich gezeigt, dass bei einer eher schlecht gewählten Blockgröße (Blockgröße $\in$ [2;7]]), bei allen Optimierungsvarianten und den darin verwendeten Suchalgorithmen eine Reduktion der Laufzeit erreicht wurde, was die Funktionalität des Ansatzes unter Beweis stellt. Besonders fällt hierbei auf, dass die durch Hillclimb optimierten Durchläufe, an den Blockgrößen 1 und 2, ziemlich nah am Optimum liegen. Dies ist darin begründet, dass der Hillclimbalgorithmus abbricht, sobald alle Nachbarkonfigurationen der aktuellen Konfiguration schlechter sind[vgl. Wikc], was bei der optimalen Konfiguration der Fall ist. Die Simulated Annealing Testdurchläufe haben an dieser Stelle (Blockgröße 1-2) etwas mehr Zeit benötigt, da dort das Optimum recht schnell gefunden wurde, aber durch die Anzahl der Optimierungsschritte ständig schlechtere Konfigurationen getestet werden mussten und sich somit die Laufzeit erhöht hat. Für den Rest der Durchläufe [3-20] wurde die optimale Konfiguration von den beiden Suchalgorithmen nicht gefunden. Für die Hillclimb-Durchläufe liegt das daran, dass der Hillclimb-Algorithmus bei lokalen Minima die Optimierung einstellt und ein Schritt in eine „schlechtere Richtung“ nicht möglich ist. Somit kann die optimale Konfiguration ab einer Blockgröße von 3 nicht mehr gefunden werden, da dazu ein Rückschritt auf die schlechtere Blockgröße 2 nötig wäre. Der Simulated Annealing Algorithmus bietet die Möglichkeit, das Optimierungsgeschehen durch eine Menge von wählbaren Parametern (Alpha, Max Optimierungsschritte und Starttemperatur) zu beeinflussen[vgl. MAR]. Werden diese nicht optimal gewählt, so ist der Algorithmus nicht in der Lage das Optimum zu finden[vgl. MAR], was sich ebenfalls in den Messdaten widerspiegelt.

Die zweite Anwendung (Abb 8.2) führt ebenfalls eine sequentielle Matrixmultiplikation durch, nur wird diese im Gegensatz zum ersten Testlauf, in fünf Threads ausgeführt, sodass jeder Thread 100 mal die sequentielle Matrixmultiplikation durchführt. Somit wird der zu optimierende Programmteil parallel ausgeführt und der Test hat folglich die Aufgabe zu demonstrieren, dass dieser Ansatz in der Lage ist paralle Programmteile zu optimieren. Dafür wurde analog zum ersten Test, der unoptimierte Durchlauf, als Vergleich mit den Blockgrößen 1 bis 20 ausgeführt. Dabei hat sich gezeigt, dass ebenfalls die Blockgröße 1 das Optimum und die Blockgröße 2 das globale Maximum darstellt. Im weiteren Verlauf sinkt die Laufzeit mit steigender

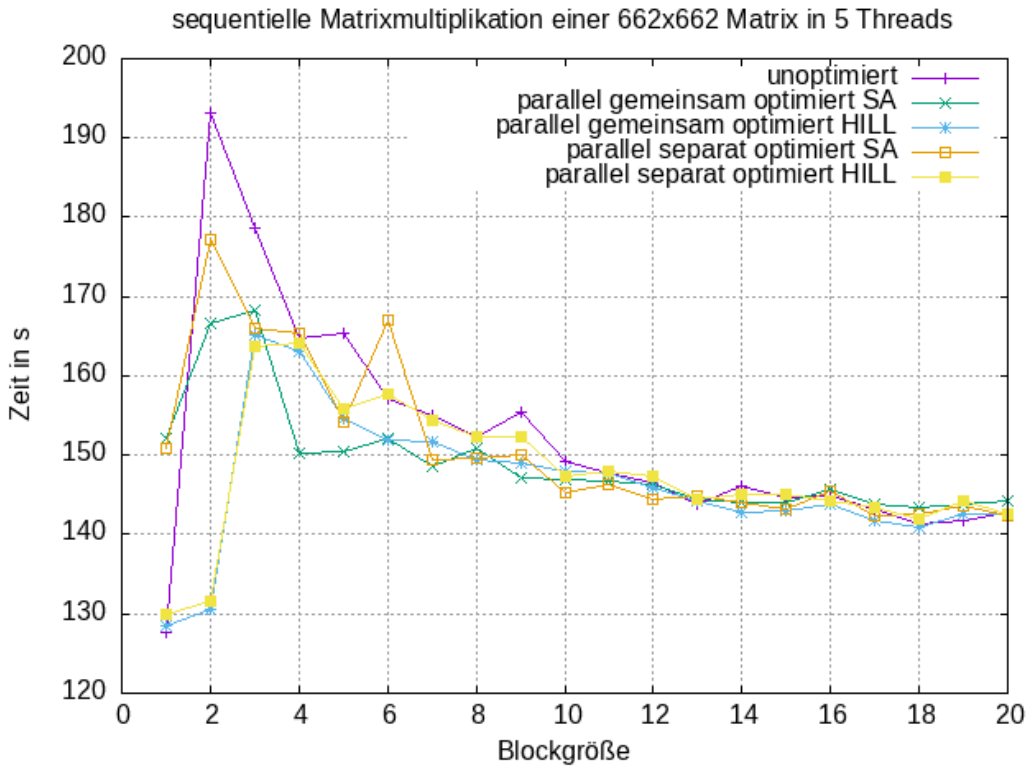

Abb. 8.2: Laufzeitmessung einer sequentiellen Matrixmultiplikation, die 100 mal in jeweils 5 Threads ausgeführt wird

Blockgröße bis einschließlich 13. Wobei in diesem Intervall (Blockgrößen $\in$ [2;13]) zwei Ausreißer (Blockgrößen: 5 & 8)) existieren, die lokale Minima, bei den Blockgrößen zuvor (Blockgrößen: 4 & 8), nach sich ziehen. Am Ende (Blockgröße $\in$ [14;20]) schwankt die Laufzeit des unoptimierten Programms um den Wert 142 Sekunden. Am Anfang (Blockgröße 1) lässt sich das gleiche Phänomen, wie bei der Messung der ersten Anwendung beobachten. Dort (Blockgröße 2) befinden sich die optimierten Durchläufe in der Nähe des Optimums, was analog wie bei der ersten Messung auf die jeweiligen Eigenschaften, sowie der Wahl der Parameter (nur SA), der Suchalgorithmen zurückzuführen sind. Für den oben genannten Bereich, in dem die Laufzeit des unoptimierten Programms, bis auf die zwei Ausnahmen, kontinuierlich sinkt, befinden sich auch die Messergebnisse der optimierten Durchläufe weitestgehend unter dem Graphen des unoptimierten Durchlaufs. Dies trifft vor allem auf die Messungen der parallel-gemeinsamen Optimierung zu, da sich diese ausnahmslos, in dem Blockgrößenintervall von 2 bis 13, unter dem Graphen der unoptimierten Variante befinden. Der Graph der parallel-separaten Optimierungsvariante, die als Suchalgorithmus Hillclimb verwendet, verläuft in diesem Bereich (Blockgröße $\in$ [2;13]) ziemlich dicht am Graphen der unoptimierten Variante und stellt sich deshalb für diese Anwendung als eher ungeeignet heraus. Die Messreihe der parallel-separaten Optimierungsvariante, die den Simulated Annealing Suchalgorithmus verwendet, liegt in diesem Bereich größtenteils unter dem Graphen des

unoptimierten Durchlaufs, aber besitzt zwei Ausreißer an den Blockgrößen 4 und 6. Die Laufzeit des Durchgangs dieser Variante mit der Startblockgröße 4 liegt nur knapp über der Laufzeit des unoptimierten Durchlaufs. Das Messergebnis des Durchlaufs mit der Startblockgröße 6 liegt weit über dem der unoptimierten Variante. Am Ende (Blockgröße $\in$ [14;20]) schwanken die Laufzeiten aller optimierten Durchläufe ebenfalls um den Wert 142 Sekunden und liegen somit sehr dicht an denen des unoptimierten Vergleichsdurchlaufs.

Zusammenfassend lässt sich sagen, dass der Optimierer durchaus funktionsfähig ist, aber nicht in jeder Situation die beste Konfiguration finden kann. Das Ergebnis könnte durch die Implementierung besserer Suchalgorithmen, oder durch eine bessere Anpassung der Parameter im Simulated Annealing Suchalgorithmus verbessert werden.

9 Fazit und Ausblick

In den Laufzeitmessungen hat sich gezeigt, dass der Optimierer in der Lage ist, bei ungünstig gewählten Konfigurationen eine Laufzeitverbesserung zu erreichen. Dies spricht dafür, dass der Ansatz einen Online-Parameterautotuner mit Hilfe von Debuggerwerkzeugen zu realisieren, durchaus funktionsfähig ist. Jedoch können die Optimierungsergebnisse durch die Implementierung besserer Suchalgorithmen verbessert werden. Eine weitere Möglichkeit die Laufzeit einer Anwendung zu minimieren wäre, bereits implementierte Funktionsvarianten bezüglich ihrer Laufzeit gegeineinander zu testen und gegebenenfalls während der Ausführung zu tauschen. Dies kann beispielsweise realisiert werden, indem man die Funktionsvarianten in der Konfigurationsdatei angibt und der Optimierer deren Funktionspointer während der Laufzeit austauscht. Falls eine der neu getesteten Funktionsvarianten eine geringere Ausführungszeit besitzt, kann auf diesem Weg eine Laufzeitverbesserung erreicht werden.

Literaturverzeichnis

Literatur

[AT11] Jeffrey K. Hollingsworth Ananta Tiwari. *Online Adaptive Code Generation and Tuning*. IEEE, 2011 (siehe S. 1, 39, 40).

[Gro02] UNIX International Programming Languages Special Interest Group. *A Consumer Library Interface to DWARF*. 2002 (siehe S. 9, 10).

[Gro05] Free Standards Group. *DWARF Debugging Information Format*. 2005 (siehe S. 4–7, 9, 10, 12).

[Int21] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2021 (siehe S. 18, 19).

[Int86] Intel. *INTEL 80386PROGRAMMER'S REFERENCE MANUAL*. 1986 (siehe S. 17–19).

[NK14] Thomas Rauber Natalia Kalinnik Matthias Korch. *Online auto-tuning for the time-step-based parallel solution of ODEs on shared-memory systems*. 2014 (siehe S. v, 1).

[RA18] Gorlatch S. Rasch A. *ATF: A generic directive-based auto-tuning framework*. 2018 (siehe S. 1, 39).

[Ran11] Randal E. Bryant and David R. O'Hallaron. *Computer Systems A Programmer's Perspective*. Pearson, 2011 (siehe S. 10, 11).

[TIS95] TIS Committee. *Tool Interface Standard (TIS)Executable and Linking Format (ELF) Specification*. 1995 (siehe S. 3, 4).

Webseiten

[Ben11a] Eli Bendersky. *How debuggers work: Part 2 - Breakpoints*. 2011. URL: `https://eli.thegreenplace.net/2011/01/27/how-debuggers-work-part-2-breakpoints` (besucht am 12. Apr. 2021) (siehe S. 18–20).

[Ben11b] Eli Bendersky. *Stack frame layout on x86-64*. 2011. URL: `https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64` (besucht am 10. Apr. 2021) (siehe S. 11).

[Ben18] Eli Bendersky. *Launching Linux threads and processes with clone*. 2018. URL: https://eli.thegreenplace.net/2018/launching-linux-threads-and-processes-with-clone/ (besucht am 10. Apr. 2021) (siehe S. 14, 33).

[Bra11] Simon Brand. *CppCon 2018: Simon Brand "How C++ Debuggers Work"*. 2011. URL: https://www.youtube.com/watch?v=0DDrseUomfU (besucht am 15. Apr. 2021) (siehe S. 20).

[DW21a] David Mosberger-Tang Dave Watson Arun Sharma. *libunwind-ptrace(3)*. 2021. URL: https://www.nongnu.org/libunwind/man/libunwind-ptrace(3).html (besucht am 16. Apr. 2021) (siehe S. 12).

[DW21b] David Mosberger-Tang Dave Watson Arun Sharma. *The libunwind project*. 2021. URL: https://www.nongnu.org/libunwind/docs.html (besucht am 16. Apr. 2021) (siehe S. 12).

[DW21c] David Mosberger-Tang Dave Watson Arun Sharma. *unw_get_reg(3)*. 2021. URL: https://www.nongnu.org/libunwind/man/unw_get_reg(3).html (besucht am 16. Apr. 2021) (siehe S. 12).

[DW21d] David Mosberger-Tang Dave Watson Arun Sharma. *unw_step(3)*. 2021. URL: https://www.nongnu.org/libunwind/man/unw_step(3).html (besucht am 16. Apr. 2021) (siehe S. 12).

[FSF21] Inc. Free Software Foundation. *3.11 Options That Control Optimization*. 2021. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options (besucht am 16. Apr. 2021) (siehe S. 12).

[MAR] MARTIN PFEIFFER. *SIMULATED ANNEALING*. URL: https://isgwww.cs.uni-magdeburg.de/sim/vilab/2003/presentations/martin.pdf (besucht am 25. Mai 2021) (siehe S. 30, 42).

[Mic11] Microsoft. *x64 Architecture*. 2011. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture (besucht am 15. Apr. 2021) (siehe S. 20).

[Mic20] Microsoft. *Aufrufkonvention bei x64-Systemen*. 2020. URL: https://docs.microsoft.com/de-de/cpp/build/x64-calling-convention?view=msvc-160#callercallee-saved-registers (besucht am 18. Apr. 2021) (siehe S. 10).

[nyn06] nynaeve. *Debugger flow control: Hardware breakpoints vs software breakpoints*. 2006. URL: http://www.nynaeve.net/?p=80 (besucht am 11. Apr. 2021) (siehe S. 17–19).

[Pro] Prof. Dr. Reinhold Kröger. *Übersicht zur Programmentwicklung unter UNIX*. URL: https://wwwvs.cs.hs-rm.de/lehre/material/extern/os08ws/dokumente/C_dev_kz.pdf (besucht am 2. Apr. 2021) (siehe S. 3, 5).

[Pro20] The Open MPI Project. *FAQ: Debugging applications in parallel*. 2020. URL: https://www.open-mpi.org/faq/?category=debugging (besucht am 26. Mai 2021) (siehe S. 40).

[SASL] Srini Devadas Michael Ernst Max Goldman John Guttag Daniel Jackson Rob Miller Martin Rinard Saman Amarasinghe Adam Chlipala und Armando Solar-Lezama. *Reading 23: Locks and Synchronization*. URL: https://web.mit.edu/6.005/www/fa15/classes/23-locks/#reading_23_locks_and_synchronization (besucht am 30. Mai 2021) (siehe S. 1).

[tec21] techopedia. *Stack Unwinding*. 2021. URL: https://www.techopedia.com/definition/22761/stack-unwinding (besucht am 16. Apr. 2021) (siehe S. 12).

[Tob] Tobias Stamm. *Call-Stack*. URL: https://manderc.com/concepts/callstack/index.php (besucht am 9. Apr. 2021) (siehe S. 10).

[Weba] *execve(2) — Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man2/execve.2.html (besucht am 16. Apr. 2021) (siehe S. 14).

[Webb] *ptrace(2) — Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html (besucht am 14. Apr. 2021) (siehe S. 13, 14, 19, 20, 32, 33, 36).

[Webc] *signal(7) — Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man7/signal.7.html (besucht am 12. Apr. 2021) (siehe S. 19, 20).

[Webd] *Understanding the Stack*. URL: https://web.archive.org/web/20130225162302/http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html (besucht am 9. Apr. 2021) (siehe S. 10, 11).

[Webe] *wait(2) — Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man2/wait.2.html (besucht am 12. Apr. 2021) (siehe S. 14, 20).

[Wika] Wikipedia. *Executable and Linking Format*. URL: https://de.wikipedia.org/wiki/Executable_and_Linking_Format (besucht am 2. Apr. 2021) (siehe S. 3).

[Wikb] Wikipedia. *Haltepunkt (Programmierung)*. URL: https://de.wikipedia.org/wiki/Haltepunkt_(Programmierung) (besucht am 30. Mai 2021) (siehe S. 1).

[Wikc] Wikipedia. *Hill climbing*. URL: https://en.wikipedia.org/wiki/Hill_climbing (besucht am 25. Mai 2021) (siehe S. 30, 42).

[Wikd] Wikipedia. *Kontextwechsel*. URL: https://de.wikipedia.org/wiki/Kontextwechsel (besucht am 11. Apr. 2021) (siehe S. 18).

[Wike] Wikipedia. *Loop nest optimization*. URL: https://en.wikipedia.org/wiki/Loop_nest_optimization (besucht am 30. Mai 2021) (siehe S. 1).

[Wikf] Wikipedia. *Object file*. URL: https://en.wikipedia.org/wiki/Object_file (besucht am 2. Apr. 2021) (siehe S. 3).

[Wikg] Wikipedia. *Parallele Programmierung*. URL: https://de.wikipedia.org/wiki/Parallele_Programmierung (besucht am 30. Mai 2021) (siehe S. 1).

[Wikh] Wikipedia. *Process identifier*. URL: https://de.wikipedia.org/wiki/Process_identifier (besucht am 14. Apr. 2021) (siehe S. 13).

[Zey17] Zeyuan Hu. *Understanding how function call works*. 2017. URL: https://zhu45.org/posts/2017/Jul/30/understanding-how-function-call-works/ (besucht am 9. Apr. 2021) (siehe S. 10, 11).

Abbildungsverzeichnis

Selbstständigkeitserklärung

Hiermit versichere ich, Fabian Mikula, dass ich die von mir vorgelegte Arbeit *Parameterautotuning für kompilierte Anwendungen durch Verwendung von Debugger-Werkzeugen* selbstständig verfasst und keine anderen als die angegebenen Quelle und Hilfsmittel verwendet habe.

Bayreuth, Juni 10, 2021

Fabian Mikula