

TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK  
INSTITUT FÜR TECHNISCHE INFORMATIK  
PROFESSUR FÜR RECHNERARCHITEKTUR  
PROF. DR. WOLFGANG E. NAGEL

## Bachelor-Arbeit

zur Erlangung des akademischen Grades  
Bachelor of Science

# Dynamische Anpassung Compiler-basierter Instrumentierung unter Nutzung von Laufzeitstatistiken

Philipp Trommler  
(Geboren am 15. Januar 1990 in Merseburg (Saale))

Hochschullehrer:  
Prof. Dr. Wolfgang E. Nagel,  
Prof. Dr.-Ing. Jeronimo Castrillon

Betreuer: Dipl.-Inf. Joseph Schuchart

Dresden, 24. Oktober 2016



## AUFGABENSTELLUNG FÜR DIE BACHELORARBEIT

*Name, Vorname des Studenten:* Trommler, Philipp

*Immatrikulations-Nr.:* **3781127**

*Thema:* Dynamische Anpassung Compiler-basierter Instrumentierung  
unter Nutzung von Laufzeitstatistiken

*Zielstellung:*


- Entwurf eines dynamischen Filter-Konzepts für Compiler-Instrumentierung durch Binär-Code-Anpassung zur Laufzeit
- Prototypische Implementierung als Substrat-Plugin in Score-P
- Messung des Laufzeit-Overheads und der Trace-Datengröße
- Vergleich mit existierenden Lösungen für beispielhafte Anwendungen (C/C++ und Fortran)

*Betreuender Hochschullehrer:* Prof. Dr. Wolfgang E. Nagel

*Betreuer:* Dipl.-Inf. Joseph Schuchart,  
Dipl.-Inf. Robert Schöne

*Beginn am:* 01.08.2016

*Einzureichen am:* 24.10.2016

  
Unterschrift des betreuenden Hochschullehrers

---

# Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Bachelor-Arbeit zum Thema:

*Dynamische Anpassung Compiler-basierter Instrumentierung unter Nutzung von  
Laufzeitstatistiken*

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 24. Oktober 2016

Philipp Trommler

---

## Kurzfassung

Im Rahmen dieser Arbeit werden das Potential und die Grenzen der dynamischen Anpassung Compiler-basierter Instrumentierung unter Nutzung von Laufzeitstatistiken untersucht. Hierfür wird prototypisch ein Plugin für die „Substrates“-Schnittstelle von *Score-P* entwickelt, das unter Nutzung verschiedener Metriken zur Laufzeit Instrumentierungsfunktionen aus dem Maschinencode entfernt. Neben dem derzeitigen Stand der Forschung auf diesem Gebiet werden Details über die Implementierung des Prototyps dargelegt. Dabei wird auf technische Herausforderungen und die dafür gefundenen Lösungen ausführlich eingegangen.

Zur Bewertung der gefundenen Lösung werden die Ausführungszeiten und die Größe sowie der Informationsgehalt der entstehenden Traces für die Benchmarks *bt-mz*, *sp-mz* und *lu-mz* der *NAS Parallel Benchmarks*-Suite sowie für den *LULESH*-Benchmark zwischen der vollständigen Instrumentierung mit *Score-P* und der Nutzung des Plugins verglichen. Um eine Einordnung der Ergebnisse zu ermöglichen, wird die in *Score-P* enthaltene Filterfunktion ebenfalls untersucht. Die Ergebnisse zeigen, dass durch die Nutzung von Laufzeitstatistiken und die Anpassung der Instrumentierung durch Veränderung des Maschinencodes der Instrumentierungs-Overhead und die Tracegröße positiv beeinflusst werden können. Gleichzeitig werden die relevanten Informationen gewahrt, die durch die Instrumentierung gewonnen werden. Es zeigt sich aber auch, dass der gewählte Ansatz einer Implementierung als Plugin für *Score-P* diverse Nachteile mit sich bringt.

Der Quelltext des Plugins, das im Rahmen dieser Arbeit entwickelt worden ist, ist unter [https://github.com/Ferruck/scorep\\_substrates\\_dynamic\\_filtering](https://github.com/Ferruck/scorep_substrates_dynamic_filtering) abrufbar.

## Abstract

The scope of this work is to examine the capabilities and limits of the dynamic adaptation of compiler-based instrumentation with the usage of run-time statistics. Therefor a prototype in form of a plugin for the “substrates” interface of *Score-P* has been developed which uses different metrics to remove instrumentation functions from machine code at run-time. Besides the current state of research, details about the implementation of the prototype are given. In the course of this technical challenges and the found solutions are presented.

To evaluate the found solution, execution times as well as the size and the information content of the resulting trace files are compared between the full instrumentation using *Score-P* and the usage of the plugin for the *bt-mz*-, *sp-mz*- and *lu-mz*-benchmarks of the *NAS Parallel Benchmarks* suite and the *LULESH* benchmark. To put the results in context, *Score-P*’s built-in filtering method is evaluated as well. The results show that the usage of run-time statistics and the adaption of the instrumentation through modifying the machine code can have a positive influence on the overhead and the size of the traces. At the same time all relevant information gathered by the instrumentation is respected. However, results show that choosing the approach of an implementation as a plugin for *Score-P* has various disadvantages, too.

The code of the plugin that has been developed in the scope of this work can be found at [https://github.com/Ferruck/scorep\\_substrates\\_dynamic\\_filtering](https://github.com/Ferruck/scorep_substrates_dynamic_filtering).

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
<b>2</b>	<b>Stand des Wissens</b>	<b>4</b>
<b>3</b>	<b>Technischer Hintergrund &amp; prototypische Implementierung</b>	<b>8</b>
3.1	Erkennen der Instrumentierungsaufrufe . . . . .	8
3.2	Erkennen der irrelevanten Funktionen . . . . .	12
3.3	Entfernen der Instrumentierungsaufrufe . . . . .	14
3.4	Synchronisierung mehrerer Threads . . . . .	17
<b>4</b>	<b>Ergebnisse</b>	<b>19</b>
4.1	Einfluss des Plugins auf die Ausführungsdauer . . . . .	19
4.2	Einfluss des Plugins auf die erzeugten Traces . . . . .	27
<b>5</b>	<b>Diskussion</b>	<b>31</b>
<b>6</b>	<b>Zusammenfassung &amp; Ausblick</b>	<b>35</b>
	<b>Anhang</b>	<b>36</b>
	<b>Literaturverzeichnis</b>	<b>38</b>
	<b>Abbildungsverzeichnis</b>	<b>41</b>
	<b>Tabellenverzeichnis</b>	<b>42</b>
	<b>Algorithmenverzeichnis</b>	<b>43</b>
	<b>Quelltextverzeichnis</b>	<b>44</b>

# 1 Einleitung

Fehler in der Logik eines Programms in der Softwareentwicklung können durch das iterative Abwechseln von Erweiterung des Quelltextes und Prüfen des daraus resultierenden Programms oft schnell entdeckt und behoben werden. Hierbei können verschiedene Programme wie beispielsweise Debugger genutzt werden. Zur Analyse von Problemen bei der Ausführung des Programms (*Programmablauf*) bedarf es ebenfalls spezieller Werkzeuge, da es zunehmend schwerer wird, ohne die Zuhilfenahme dieser Werkzeuge Programme zu entwickeln und deren Ablauf vollumfänglich zu verstehen. Dies gilt insbesondere vor dem Hintergrund der immer weiter zunehmenden Komplexität und Heterogenität der unterliegenden Hardware, vor allem im Bereich des Hochleistungsrechnens (*High Performance Computing, HPC*). Besonders bei der Optimierung der Laufzeit und des Ressourcenverbrauchs eines Programms (*Performance-Optimierung*) sind Werkzeuge, die dem Programmierer einen tiefen Einblick in den Programmablauf erlauben, unersetzlich. Bei der Benutzung von Debuggern ist der von diesen erzeugte Overhead nicht relevant, da eine Fehlersuche zumeist keine performance-kritische Problemstellung darstellt. Bei Werkzeugen, die Messungen zu Laufzeit durchführen, muss der Overhead jedoch möglichst gering gehalten werden, da dieser Einfluss auf die Messergebnisse nimmt.

Mit *Score-P* [KRM<sup>+</sup>12] existiert ein Werkzeug, mit dem es möglich ist, Informationen über den Programmablauf zur Laufzeit zu akquirieren, um sie post-mortem analysieren zu können. Dafür erweitert es das Programm durch Einfügen zusätzlicher Funktionsaufrufe, um so bestimmte Ereignisse wie das Betreten oder Verlassen von Funktionen registrieren zu können (*Instrumentierung*). Die so gewonnenen Daten können entweder aggregiert als Zusammenfassung (*Profile*) oder als vollständige Sammlung aller erhobenen Daten (*Trace*) gespeichert werden, wobei Letzteres für die Analyse von Programmen unverzichtbar ist, da auch Ausreißer oder Auffälligkeiten, die nur an bestimmten Punkten in der Programmausführung auftreten, aufgenommen werden. Durch dieses grundsätzliche Vorgehen von *Score-P* entstehen vor allem zwei Probleme:

1. Das Einfügen der zusätzlichen Funktionsaufrufe führt zu einer Verlängerung der Programmablaufzeit (*Overhead*). Dies wiegt umso schwerer, je kürzer die instrumentierten Funktionen sind, da der Instrumentierungsaufwand pro Funktion konstant ist.
2. Beim Tracing entstehen mitunter sehr große Datenmengen. Diese zur Laufzeit performant zu halten und zu speichern, ist eine der größten Schwierigkeiten bei der Sammlung von Informationen durch Instrumentierung.

Beiden Problemen kann dadurch begegnet werden, dass die Instrumentierung nur auf relevante Funktionen angewendet wird, also auf solche Funktionen, die für das Verständnis des Programmablaufs zwingend notwendig sind. Anders ausgedrückt können die Probleme der Instrumentierung dadurch abgemildert werden, dass irrelevante Funktionen nicht instrumentiert werden.

Oft kann jedoch im Vorfeld nicht automatisiert festgestellt werden, welche Funktionen eines Programms irrelevant sind, weshalb alle Funktionen instrumentiert werden müssen. Zudem zeigen sich erst zur Laufzeit die Charakteristika der Funktionen wie die (durchschnittliche) Laufzeit oder die Anzahl der Aufrufe, die maßgeblich den Einfluss der Instrumentierung auf die Programmlaufzeit sowie die entstehende Datenmenge bestimmen.

*Score-P* enthält die Möglichkeit zur Begrenzung der Instrumentierung auf relevante Funktionen, um die entstehenden Datenmengen und den Overhead zu reduzieren. Aufgrund zuvor genannter Probleme wird hierbei auf vom Benutzer manuell erstellte Filterlisten zurückgegriffen. Die Instrumentierung wird bei diesem Vorgehen allerdings nicht vollständig aus dem Programm entfernt, sodass ein Teil des negativen Einflusses auf die Programmlaufzeit bestehen bleibt.

Im Rahmen dieser Arbeit ist daher eine Methode entwickelt worden, mit der die Instrumentierung zur Laufzeit, wenn die entscheidenden Charakteristika der Funktionen bekannt sind, automatisiert entfernt werden kann. Dies ist in Form eines prototypischen Plugins für *Score-P* geschehen. Dieses Plugin erkennt anhand festgelegter Metriken irrelevante Funktionen und entfernt die Instrumentierung an den entsprechenden Stellen im Programm vollständig.

Die vorliegende Arbeit ist wie folgt gegliedert: Zunächst soll in Kapitel 2 ein Überblick über den Stand des Wissens und der aktuellen Forschung in diesem und angrenzenden Gebieten gegeben werden. Anschließend sollen die Umsetzung der Idee in Kapitel 3 und die daraus resultierenden Ergebnisse in Kapitel 4 dargestellt und in Kapitel 5 diskutiert werden. Den Abschluss der Arbeit bilden das Fazit und der Ausblick auf mögliche zukünftige Arbeiten in diesem Gebiet in Kapitel 6.

## 2 Stand des Wissens

Kurze, oft aufgerufene (*hochfrequente*) und instrumentierte Funktionen, im Kontext der Instrumentierung auch *Regionen* genannt, erzeugen bei der Messung sehr große Datenmengen, die nur sehr schwer handhabbar sind, und weisen einen höheren Overhead auf als langandauernde Funktionen. Im Fall von sehr kurzen Funktionen verändert dieser Overhead zudem in signifikantem Maße die Messergebnisse (*Performance Perturbation*, [MRW92]). Gleichzeitig tragen Informationen über viele von diesen Regionen, zum Beispiel Funktionen zum Lesen und Schreiben einzelner Variablen, aber kaum zum Verständnis über den Programmablauf bei [WDKN14].

Aus diesen Gründen werden diverse Anstrengungen unternommen, um dem Overhead und den entstehenden Datenmengen zu begegnen (Tabelle 2.1). Als Einstiegspunkt im Hinblick auf *Score-P* ist hier dessen eingebaute Filterfunktion zu nennen, die die Aggregation von Daten für Funktionen einstellt, die in vom Nutzer bereitgestellten Filterlisten genannten werden, und so die Menge an entstehenden Daten reduziert. Hierbei wird bei jedem Instrumentierungsauf-ruf der Name der zugehörigen Region mit den in der Liste gegebenen Namen verglichen. Ist die Region zum Filtern markiert, wird die Instrumentierung an dieser Stelle verlassen, sonst normal fortgeführt. Nach Tabelle 2.1 handelt es sich also um ein Vorgehen des Typs A1. So entsteht auch bei gefilterten Funktionen ein gewisser, wenn auch geringer, Overhead durch die Instrumentierung [KRM<sup>+</sup>12].

Für *Scalasca* [GWW<sup>+</sup>10] und *TAU* [SM06] existieren vergleichbare Ansätze des Typs A1. Außerdem bieten diese die Möglichkeit, die Filterlisten durch einen vollinstrumentierten Lauf selbstständig zu erstellen (C1). Für Programme mit kritischen Datenmengen oder Overhead kann letztgenannte Möglichkeit aber nicht genutzt werden, da die Ressourcen für einen solchen Lauf unter Umständen nicht zur Verfügung stehen. Eine ähnliche Funktionalität bietet *Score-P* mit dem Kommandozeilenprogramm *scorep-score*, das Informationen zu einem bereits erfolgten, instrumentierten Lauf aufbereitet, aus denen der Nutzer manuell eine Filterliste erstellen kann (Typ C1). Zusätzlich bietet *TAU* mit *throttling* [MCSM06] eine Funktion, um die Datenakquise für kurze, hochfrequente Funktionen zur Laufzeit auszusetzen. Dies ist analog zu der Vorgehensweise von *Score-P* umgesetzt, sodass auch hier die entstehende Datenmenge und der Overhead reduziert, jedoch nicht gänzlich vermieden werden. Dementsprechend handelt es sich um ein Vorgehen vom Typ D1.

Auch das inzwischen von *Score-P* abgelöste *VampirTrace* [KBD<sup>+</sup>08] bot die Möglichkeit über manuell angelegte Filterlisten Einfluss auf die Instrumentierung zu nehmen. Dabei konnte nicht nur die Datenakquise analog zum Vorgehen von *Score-P* für bestimmte Funktionen übersprungen werden, es konnte auch eine maximale Anzahl von Aufrufen definiert werden, bis zu der die Region zur Akquise herangezogen wurde. Auch diese Implementierung fiel in die Kategorie D1 und zeigte deshalb dieselben Nachteile wie die Umsetzungen von *Score-P*, *Scalasca* und *TAU* [Zen13].



Tabelle 2.1: Übersicht über verschiedene Ansätze zur Reduktion des Instrumentierungs-Overheads, sowohl zeitlich als auch in Form von unbenötigten Daten. Die manuelle Selektion irrelevanter Funktionen (A) erfordert Vorwissen über das instrumentierte Programm und bildet deshalb einen Sonderfall. Während die Reduktion der entstehenden Datenmenge in 1 und 2 gleich ist, kann der Overhead nur bei 2 theoretisch komplett eliminiert werden.

		Methode zur Erkennung irrelevanter Funktionen			
		manuell (A)	statische Analyse (B)	post-mortem Analyse (C)	Laufzeit-analyse (D)
Overheadre- duktion durch	Überspringen (1)	<i>Score-P</i> , <i>Scalasca</i> , <i>TAU</i>		<i>Score-P</i> , <i>Scalasca</i> , <i>TAU</i>	<i>TAU</i> ( <i>throttling</i> ), STOLLE
	Entfernen (2)	<i>GCC</i> , <i>Paradyn</i> , <i>Pin</i> , MUSS- LER et al.	<i>OpenUH</i> , MUSS- LER et al.		

All diesen Ansätzen ist die Tatsache gemein, dass die eigentliche Instrumentierung im Programm verbleibt und somit weiterhin Overhead bei der Programmausführung entsteht, da bei jedem Instrumentierungsaufwurf geprüft werden muss, ob die vorliegende Region für die Datenakquise herangezogen werden soll. Um dem zu begegnen, stehen verschiedene Möglichkeiten bereit, nur ausgewählte Funktionen zu instrumentieren, um so auch den Overhead möglichst gering zu halten.

So bieten die Compiler der *GNU Compiler Collection*<sup>1</sup> (*GCC*) ein eigenes Verfahren, um benutzerdefinierte Funktionen zur Instrumentierung in den Quelltext einzufügen. Dabei können durch Angabe entsprechender Kommandozeilenargumente beim Kompilieren ganze Dateien oder auch einzelne Funktionen von der Instrumentierung ausgenommen werden (Typ A2 Vorgehen). Der *OpenUH*-Compiler, eine von LIAO et al. [LHC<sup>+</sup>07] vorangetriebene Abspaltung des *Open64*<sup>2</sup>, nutzt die während der Kompilierung verfügbaren Informationen über das vorliegende Programm, um selbsttätig Entscheidungen darüber zu treffen, welche Funktionen instrumentiert werden sollen. Durch diese statische Analyse handelt es sich um ein Vorgehen des Typs B2 [HJC07].

MUSSLER et al. [MLW11] beschreiben hingegen eine Schnittstelle, die es erlaubt, mithilfe eigener Programme Einfluss auf die Instrumentierung zu nehmen. Über die Schnittstelle werden diesen Informationen über das Programm übermittelt, die zuvor mittels statischer Analyse gewonnen worden sind. Die Erweiterungen entscheiden auf Grundlage dieser Informationen, welche Funktionen instrumentiert werden sollen, und übermitteln diese Entscheidung dann wieder über die Schnittstelle. Die eigentliche Instrumentierung findet hierbei durch Einschleusen von Binär-Code in das bereits vorhandene Kompilat statt (*binary rewriting*). Die Kombination der statischen Analyse mit der manuellen Auswahl lässt eine eindeutige Einordnung in die Tabelle 2.1 nicht zu, es handelt sich um eine Kombination aus den Typen A2 und B2.

Die zuletzt genannten Lösungsansätze können sowohl die Menge an anfallenden Daten als auch den Overhead der gefilterten Funktionen komplett reduzieren, sie befinden sich alle in der zweiten

<sup>1</sup><https://gcc.gnu.org/>

<sup>2</sup><https://sourceforge.net/projects/open64/>

Zeile der Tabelle 2.1. Dabei entscheiden sie entweder benutzergesteuert oder auf Grundlage der Informationen, die ihnen die statische Analyse des Quelltexts liefert. Aus statischer Analyse allein lässt sich jedoch nur schwer die für ihre Relevanz oftmals entscheidende, zeitliche Länge einer Funktion ermitteln, da diese nicht immer mit der Anzahl an Befehlen in der Funktion, der Wortlänge, korreliert. Synchronisierungen, Kommunikation oder auch Speicherzugriffe beeinflussen die Aufrufdauer einer Funktion maßgeblich und lassen sich nicht mit statischer Quelltextanalyse vorhersagen. Für den Benutzer jedoch bedeutet die Bestimmung irrelevanter Funktionen einen großen zeitlichen Aufwand, der dem Zeitgewinn durch das Filtern gegenübersteht. Um also automatisiert Aussagen darüber treffen zu können, welche Funktionen voraussichtlich für das Verständnis des Programmablaufs irrelevant sind und deswegen nicht instrumentiert werden müssen, bedarf es daher zusätzlicher Informationen, die nur zur Laufzeit des Programms gesammelt werden können und die den zuvor genannten Ansätzen nicht zur Verfügung stehen. Außerdem benötigen diese bei jeder Veränderung der Instrumentierung ein erneutes Kompilieren oder erneutes Anpassen des Kompilators. Dies kann für komplexe Softwareprojekte einen unvermeidbaren Aufwand bedeuten.

Um diese Probleme zu umgehen, wählt *Paradyn* [MCC<sup>+</sup>95] einen Ansatz, bei dem das Verändern des Programms erst zur Laufzeit stattfindet. Ein überwachender Prozess untersucht das gestartete Programm auf Punkte, die für die Instrumentierung relevant sind, und fügt an diesen zusätzlichen Binär-Code ein, wobei vom Benutzer festgelegt werden kann, an welchen dieser Punkte Instrumentierung eingefügt werden soll. Auch *Pin* [LCM<sup>+</sup>05] instrumentiert Programme nach vom Benutzer gegebenen Regeln zur Laufzeit, führt sie dafür aber in einer eigenen virtuellen Maschine aus, um das Verhalten der Programme analysieren zu können. Diesen Ansätzen ist – neben dem zusätzlichen Overhead für die Überwachung beziehungsweise Virtualisierung – gemein, dass der Benutzer entweder Vorwissen über das Programm einbringen muss, um zu entscheiden, welche Regionen instrumentiert werden sollen, oder zunächst alle Funktionen instrumentiert werden müssen, um mithilfe der daraus resultierenden Informationen eine Entscheidung darüber zu treffen, welche dieser Funktionen relevant sind. Sie unterscheiden sich nach der Einteilung in Tabelle 2.1 deswegen nicht von den Ansätzen des *GCC* und von MUSSLER et al.

STOLLE [Sto15] beschreibt mehrere Möglichkeiten, auf der Grundlage der Laufzeitinformationen Funktionen automatisiert zu erkennen, die für das Verständnis des Programmablaufs irrelevant sind, um diese aus der Datenakquise im Rahmen der Instrumentierung eines zunächst vollinstrumentierten Programms zu entfernen (*adaptives/dynamisches Filtern*). Sein Hauptziel ist dabei das Filtern von Funktionen, deren Verhalten und Charakteristika sich zwischen einzelnen Aufrufen nicht ändern, wobei insbesondere kurze und oft aufgerufene Funktionen betrachtet werden. Dies folgt der Beobachtung, dass ebendiese Funktionen kaum zum Verständnis über den Programmablauf beitragen. Die in seiner Arbeit vorgestellten Verfahren entfernen die Instrumentierung aber nicht aus dem Programm, sodass der Overhead nur verringert, nicht jedoch beseitigt wird. Trotz der komplexeren Herangehensweise handelt es sich daher wie das *throttling* um ein Vorgehen vom Typ D1.

Es gibt also eine Vielzahl von vorhandenen Werkzeugen, die die beim Tracing entstehende Datenmenge reduzieren. Aus dem vorangegangenen Überblick über den Stand des Wissens ergibt sich jedoch eine Lücke. Zum einen existieren Werkzeuge, die zudem auch den Overhead für ir-

relevante Funktionen vermeiden, indem sie diese nicht instrumentieren. Bei diesen Werkzeugen kann aber nur auf Grundlage statischer Analyse oder von Benutzereingaben entschieden werden, welche Funktionen irrelevant sind (Typ A2 und B2). Wie zuvor bereits erwähnt, können qualifizierte Aussagen über die Charakteristika einer Funktion und somit über ihre Relevanz jedoch nur aufgrund von Informationen getroffen werden, die erst zur Laufzeit bekannt sind. Dementsprechend gibt es zum anderen die Werkzeuge, die auf der Grundlage ebendieser Informationen (teilweise automatisiert) dynamisch filtern. Bei der Benutzung dieser Werkzeuge bleibt jedoch ein Teil des Overheads der Instrumentierung bestehen. Das ist besonders vor dem Hintergrund, dass sich zwischen den irrelevanten Funktionen, die gefiltert werden, und den kurzen, hochfrequenten Funktionen, die den höchsten Overhead aufweisen, große Überdeckungen finden, kritisch zu sehen.

Nach dem derzeitigen Stand des Wissens existiert bisher keine Lösung, die diese Lücke schließt, die also sowohl automatisiert auf der Grundlage von Laufzeitinformationen irrelevante Funktionen erkennt – also dynamisch filtert – als auch gleichzeitig bei der Entfernung dieser Funktionen den Overhead der Instrumentierung so weit reduziert, dass die Ausführungsdauer der betroffenen Regionen mit denen einer uninstrumentierten Version des Programms vergleichbar ist. Dies kann nur von einem Ansatz des Typs D2 geleistet werden, der in dieser Arbeit vorgestellt wird.

## 3 Technischer Hintergrund & prototypische Implementierung

Beim Erkennen und Entfernen unerwünschter Instrumentierungsaufrufe im Abstrakten und der Implementierung der Vorgänge als Plugin für *Score-P* im Praktischen stellen sich diverse Herausforderungen, die im Folgenden zusammen mit den jeweils gefundenen Lösungen vorgestellt werden sollen. Grundsätzlich ist zu allen folgenden Erläuterung zu erwähnen, dass das im Rahmen dieser Arbeit entwickelte Plugin nur die Compiler-Instrumentierung betrachtet. Andere von *Score-P* genutzte Instrumentierungsarten wie beispielsweise die Bibliotheksinstrumentierung im Falle vom *Message Passing Interface*<sup>1</sup> (MPI) [Mes94] oder die Nutzerinstrumentierung werden aus Gründen nicht behandelt, die in Kapitel 5 näher erläutert werden.

### 3.1 Erkennen der Instrumentierungsaufrufe

Prinzipiell wird die Compiler-Instrumentierung durch Funktionsaufrufe umgesetzt, die beim Übersetzen des Programms eingefügt werden. Sie werden jeweils am Anfang (*Enter-Event*) und am Ende (*Exit-Event*) einer Funktion platziert, um die Zeitpunkte, zu denen die Region betreten und verlassen wird, und daraus abgeleitete Messwerte bestimmen zu können. Das im Rahmen dieser Arbeit entwickelte Plugin wird wiederum aus diesen Instrumentierungsfunktionen heraus aufgerufen (Abbildung 3.1).

Das Erkennen dieser Instrumentierungsaufrufe beschreibt zwei verschiedene Abläufe: Zum einen muss zu Beginn der Programmlaufzeit festgestellt werden, welche Aufrufe im vorliegenden Programm verwendet werden. Zum anderen müssen beim Versuch des Löschens der Instrumentierung diese Aufrufe wieder gefunden werden.

Zwar finden diese beiden Abläufe zu unterschiedlichen Zeiten bei der Ausführung des Programms statt, doch da sie sich technisch ähnlich sind, sollen sie im Folgenden zusammen betrachtet werden.

**Feststellen der verwendeten Instrumentierungsaufrufe** *Score-P* unterstützt mehrere Compiler für die Compiler-Instrumentierung. Die Funktionsaufrufe, die es dabei zum Einsprung in die Instrumentierung verwendet, sind vom verwendeten Compiler abhängig. Da das im Rahmen dieser Arbeit entwickelte Plugin nicht zwingend mit demselben Compiler übersetzt worden sein muss wie das vorliegende Programm, muss der Prototyp zur Laufzeit den für das Programm verwendeten Compiler beziehungsweise die daraus resultierenden Instrumentierungsaufrufe erkennen können.

Grundsätzlich nutzt *Score-P* folgende Funktionen für den Einsprung in die Instrumentierung beim Betreten bzw. Verlassen von Regionen:

---

<sup>1</sup><http://mpi-forum.org/>

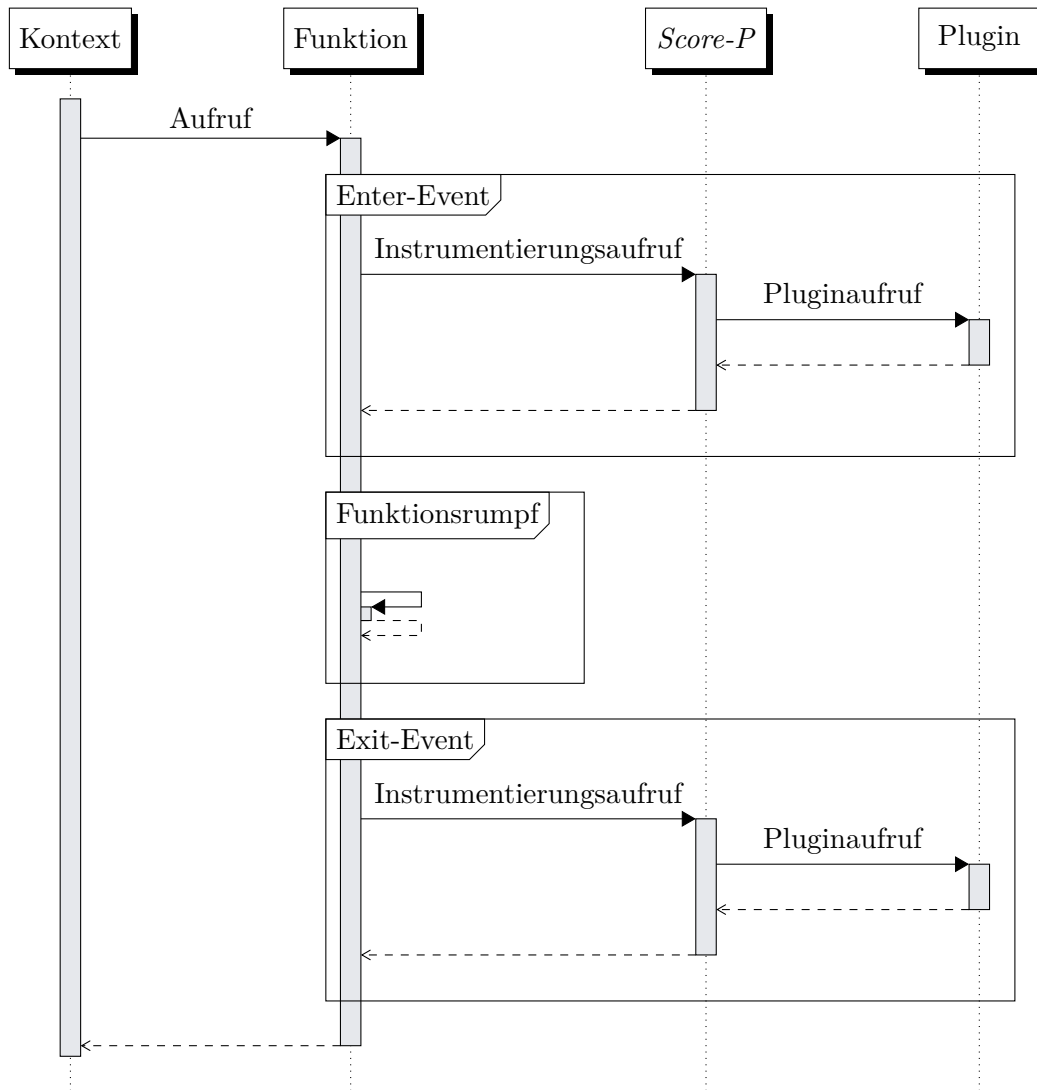


Abbildung 3.1: Schematische Darstellung der Abläufe in einer instrumentierten Funktion. Zu Beginn der aufgerufenen Funktion findet der erste Einsprung in die Instrumentierung (*Enter-Event*), direkt vor dem Rücksprung aus der Funktion der zweite (*Exit-Event*) statt. Aus der Instrumentierung heraus wird jeweils einmal das Plugin aufgerufen.

- `__cyg_profile_func_enter/__cyg_profile_func_exit` für die Compiler der *GCC* vor Version 4.5
- `scorep_plugin_enter_region/scorep_plugin_exit_region` für die Compiler der *GCC* ab Version 4.5 (*GCC*-Plugin, der Nutzer kann dennoch die ältere Variante wählen)
- `__VT_IntelEntry/__VT_IntelExit` für die *Intel*-Compiler
- weitere für im Rahmen dieser Arbeit nicht betrachtete Compiler

Der Algorithmus zum Bestimmen der Instrumentierungsaufrufe ist im Rahmen dieser Arbeit unter Nutzung der Bibliothek *libunwind*<sup>2</sup> implementiert worden. Mit dieser Bibliothek kann der Aufrufkontext betrachtet werden. Möglich ist dies durch den Kellerspei-

<sup>2</sup><http://www.nongnu.org/libunwind/>

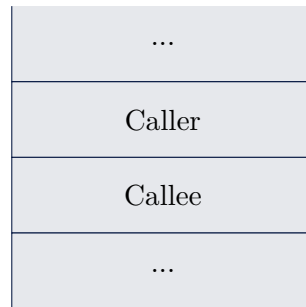


Abbildung 3.2: Schematische Darstellung eines Kellerspeichers. Für jede aufgerufene Funktion (*Callee*) wird ein neuer Speicherbereich (*Stack-Frame*) angelegt. Dieser fügt sich direkt hinter den Stack-Frame der aufrufenden Funktion (*Caller*) im Programmspeicher an. Jeder Stack-Frame enthält die für den Ausführungskontext der jeweiligen Funktion wichtigen Informationen, wie z. B. übergebene Parameter. Die übliche Darstellungsform ist wie hier ein nach unten wachsender Stapel.

cher (*Stack*), einen byteweise adressierten Speicherbereich. Dieser beginnt an der höchsten Adresse des Speichers des Programms und wächst „nach unten“, also hin zu kleineren Adressen. Auf dem Stack wird für jeden Funktionsaufruf ein neuer Speicherrahmen (*Stack-Frame*) angelegt (Abbildung 3.2). Dieser Stack-Frame bleibt erhalten, bis diese wieder verlassen wird, und enthält die lokalen Variablen der Funktion [TB15]. Die Bibliothek *libunwind* untersucht den Stack in einem *bottom-up*-Verfahren, also beim „untersten“ Stack-Frame beginnend und nach oben aufsteigend (*Stack-Unwinding*), und ermittelt für jeden Stack-Frame Informationen wie beispielsweise den Namen der zugehörigen Funktion. Mithilfe dieser Funktionalität von *libunwind* ist es möglich, die Namen aller Funktionen zu ermitteln, deren Aufruf zum Erreichen des Enter-Events geführt haben, und so den Algorithmus 3.1 zu implementieren.

Dieser Algorithmus wird beim ersten Betreten einer Region ausgeführt und durchsucht in dem zuvor beschriebenen *bottom-up*-Verfahren den Aufrufkontext nach einem der oben genannten Funktionsaufrufe für den Einsprung in die Instrumentierung beim Betreten einer Region. Der erste gefundene Aufruf bestimmt den Aufruftyp für das gesamte Programm, da der Typ innerhalb eines Programms konstant ist.

Vorausgesetzt das vorliegende Programm verwendet einen der vom Plugin behandelten Aufruftypen, ist der Algorithmus korrekt und terminiert immer, da er innerhalb der Instrumentierung beim Betreten einer Region aufgerufen wird und somit immer ein Einsprung in die Instrumentierung im Aufrufkontext vorhanden sein muss.

**Finden der Instrumentierungsaufrufe** Im Wesentlichen unterscheidet sich das Finden der Instrumentierungsaufrufe kaum vom Feststellen des Aufruftyps. Soll eine als irrelevant erkannte Funktion (siehe Abschnitt 3.2) entfernt werden, wird beim nächsten Enter-Event der Region der entsprechende Einsprung in die Instrumentierung für das Betreten der Funktion gesucht und analog beim nächsten Exit-Event der Einsprung in die Instrumentierung für das Verlassen dieser Funktion. Dadurch wird vermieden, dass zusätzlicher Overhead durch das Entfernen von Instrumentierungsfunktionen, die in der restlichen Programmlaufzeit nicht erneut aufgerufen werden, entsteht.

**Algorithmus 3.1** Bestimmung der verwendeten Instrumentierungsaufrufe**Voraussetzung:** Aufruf innerhalb der Instrumentierung eines Enter-Events.**Rückgabe:** Der im Programm verwendete Aufruftyp.

---

```

1: function DETECTCALLTYPE
2:    $o \leftarrow \text{"\_cyg\_profile\_func\_enter"}$ 
3:    $p \leftarrow \text{"scorep\_plugin\_enter\_region"}$ 
4:    $i \leftarrow \text{"\_VT\_IntelEntry"}$ 
5:   while stack not empty do
6:      $n \leftarrow$  function name of lowermost stack frame
7:     if  $n = o$  then
8:       return GCC                                     ▷ alter GCC-Typ
9:     end if
10:    if  $n = p$  then
11:      return GCC-Plugin                               ▷ neuer GCC-Typ
12:    end if
13:    if  $n = i$  then
14:      return Intel                                     ▷ Intel-Typ
15:    end if
16:    do pop the lowermost stack frame
17:  end while
18: end function

```

---

Wie auch für das Feststellen des Aufruftyps wird für das Finden der Einsprünge *libunwind* verwendet. Es werden jedoch von der Bibliothek auch Sprünge innerhalb von Funktionen beim Stack-Unwinding gefunden (Abbildung 3.3), sodass ein analoges Vorgehen zu Algorithmus 3.1, also das Erkennen des **ersten** passenden Funktionsaufrufs, in Folge zum Entfernen des falschen Aufrufs führen würde. Diesem Umstand wird in Algorithmus 3.2 Rechnung getragen, indem er den Stack solange emporsteigt, bis er einen Einsprung in die Instrumentierung gefunden hat, der selbst von einer Funktion aufgerufen wird, die keine Einsprungsfunktion ist. Von letzterer wird mithilfe von *libunwind* der *Instruction Pointer* ermittelt. Dieser zeigt auf die Adresse desjenigen Befehls, der nach dem Rücksprung in diese Funktion als nächstes ausgeführt werden soll. Die Bytes, die im Programmspeicher vor dieser Adresse liegen, müssen folglich den Sprung in die aufgerufene Funktion, also den gesuchten Einsprungspunkt in die Instrumentierung, darstellen. Auch dieser Algorithmus terminiert immer, wenn die Voraussetzungen erfüllt sind, also der korrekte Instrumentierungstyp bekannt ist und er innerhalb einer Instrumentierungsfunktion aufgerufen worden ist, da sich in diesem Fall immer ein Einsprung in die Instrumentierung, der vom Plugin erkannt wird, im Aufrufkontext befinden muss.

Beiden Algorithmen ist gemein, dass sie zur Ausführung Stack-Unwinding benötigen. Dies erzeugt jedoch einen großen Overhead. Darum wird insbesondere der Algorithmus 3.2 nur für als irrelevant markierte Funktionen ausgeführt und sein Ergebnis wird bei paralleler Ausführung des Programms an alle Threads verteilt, trotz des dadurch entstehenden Synchronisierungsaufwands (siehe auch Abschnitt 3.4).

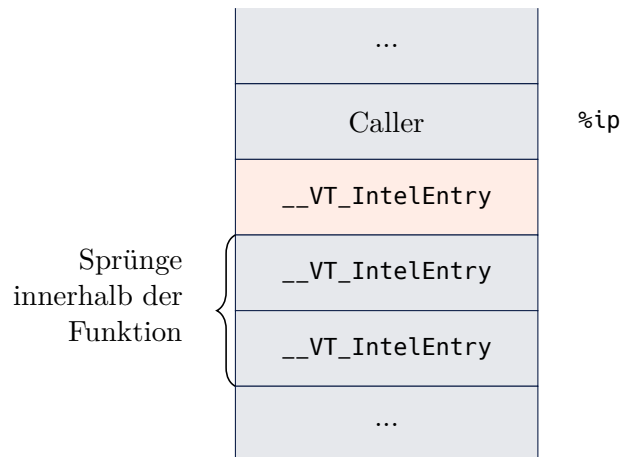


Abbildung 3.3: Beispielhafter Aufbau eines von der Ausgabe von *libunwind* implizierten Stacks. Da die Bibliothek Sprünge innerhalb von Funktionen nicht zuverlässig erkennt, entsteht der Eindruck mehrerer aufeinanderfolgender Aufrufe derselben Funktion, wie sie beispielsweise bei rekursiven Funktionen auftreten. Um aber den benötigten *Instruction Pointer* (%ip) zu erhalten, muss das erste Vorkommen der Instrumentierungsfunktion (hier: `__VT_IntelEntry`) gefunden werden.

## 3.2 Erkennen der irrelevanten Funktionen

Wie bereits in Kapitel 2 dargelegt, sind die Charakteristika „Anzahl der Aufrufe“ und „Laufzeit“ von besonderem Interesse bei der Beurteilung der Relevanz einer Funktion. Diese lassen sich, zusammen mit weiteren Messwerten, in beliebig komplexen Metriken verwenden. STOLLE [Sto15] zeigt in seiner Arbeit einige der Möglichkeiten auf, auf diese Weise irrelevante Funktionen zu erkennen. Da aber mit der Komplexität der Metriken auch der Aufwand und somit der durch die Berechnung entstehende Overhead steigt, werden im Rahmen der prototypischen Implementierung in dieser Arbeit zwei einfachere Metriken verwendet, die im Folgenden erläutert werden.

**Absolute Metrik** Die absolute Metrik entscheidet anhand der durchschnittlichen Laufzeit einer Funktion über ihre Relevanz. Genutzt werden hierfür sowohl die akkumulierte Laufzeit der Region als auch die Anzahl der Aufrufe ebendieser, um die durchschnittliche Laufzeit zu bestimmen. Der Nutzer muss zusätzlich einen Grenzwert angeben. Fällt die durchschnittliche Laufzeit der Funktion unter diesen Grenzwert, wird sie als irrelevant markiert:

$$\text{irrelevant}(\text{region}) = \begin{cases} \text{true} & \text{für } \text{mean\_duration}(\text{region}) < \text{threshold} \\ \text{false} & \text{sonst} \end{cases} \quad (3.1)$$

Diese Metrik erlaubt das Erkennen irrelevanter Funktionen schon nach dem jeweils ersten Aufruf sowie eine einfache Justierbarkeit der Aggressivität des Filterns, also des Anteils der gefilterten Regionen, über den Grenzwert. Andererseits erfordert sie vom Benutzer eine Vorstellung von der Verteilung der Laufzeiten der Funktionen seines Programms, damit dieser einen geeigneten Grenzwert wählen kann.

**Relative Metrik** Die relative Metrik vergleicht die durchschnittlichen Laufzeiten der Funktionen eines Programms untereinander. Wie auch bei der absoluten Metrik werden die Laufzeiten-



**Algorithmus 3.2** Finden eines Instrumentierungsaufrufs

**Voraussetzung:** Aufruf innerhalb eines Enter- oder Exit-Events, Name des Aufrufs der Instrumentierung  $t$  bekannt.

**Rückgabe:** Zeiger auf das erste Byte hinter dem Instrumentierungsaufruf.

---

```

1: procedure FINDCALL( $t$ )
2:    $b \leftarrow \text{false}$ 
3:   while stack not empty do
4:      $n \leftarrow$  function name of lowermost stack frame
5:     if  $n = t$  then
6:        $b \leftarrow \text{true}$  ▷ Stack-Frame gehört zu Instrumentierungsfunktion
7:     else if  $b = \text{true}$  then ▷ Stack-Frame gehört zu Aufrufer der Instrumentierung
8:       return instruction pointer for current stack frame
9:     end if
10:    do pop lowermost stack frame
11:  end while
12: end procedure

```

---

formationen verwendet, um für jede Region die durchschnittliche Laufzeit zu bestimmen, und auch hier muss vom Benutzer ein Grenzwert angegeben werden. Allerdings entscheidet die relative Metrik anhand der Unterschiede zwischen den Laufzeiten. Hat eine Funktion eine kürzere durchschnittliche Laufzeit, als die durchschnittliche Laufzeit über alle Funktionen abzüglich des vom Nutzer gegebenen Grenzwerts, wird sie als irrelevant markiert, also:

$$\begin{aligned}
 \text{condition} &= \frac{\sum_{i=1}^n \text{mean\_duration}(i)}{n} - \text{threshold} \\
 \text{irrelevant}(\text{region}) &= \begin{cases} \text{true} & \text{für } \text{mean\_duration}(\text{region}) < \text{condition} \\ \text{false} & \text{sonst} \end{cases} \quad (3.2)
 \end{aligned}$$

Aus diesem Vorgehen ergibt sich ein dynamisches Verhalten des Filterns. Durch das Entfernen kurzer Funktionen aus der Instrumentierung steigt der Wert von `condition`. Dadurch werden wiederum mehr Regionen als irrelevant markiert und anschließend entfernt. Dies hat zur Folge, dass gewahrt wird, dass Regionen mit einer längeren Laufzeit und somit einer mutmaßlich höheren Relevanz länger in der Instrumentierung verbleiben. Die Dauer der Datenakquise je Funktion ist hierbei also proportional zu ihrer Relevanz. Allerdings ist es bei der Nutzung der relativen Metrik für den Benutzer schwieriger einen geeigneten Grenzwert festzulegen als bei der absoluten Metrik.

Beiden Ansätzen sind die Zeitpunkte gemein, an denen die Metriken berechnet werden. In sequentiellen Phasen der Programmausführung werden die Metriken bei jedem Verlassen einer instrumentierten und noch nicht als irrelevant markierten Funktion berechnet, in parallelen Phasen jeweils beim Verlassen des parallelen Abschnitts (beispielsweise ein *OpenMP parallel for*, siehe auch Abschnitt 3.4). Durch diese Konzentration des metrik-spezifischen Quelltextes lassen sich die Metriken im Fall einer späteren Erweiterung des Plugins leicht austauschen oder durch weitere ergänzen.

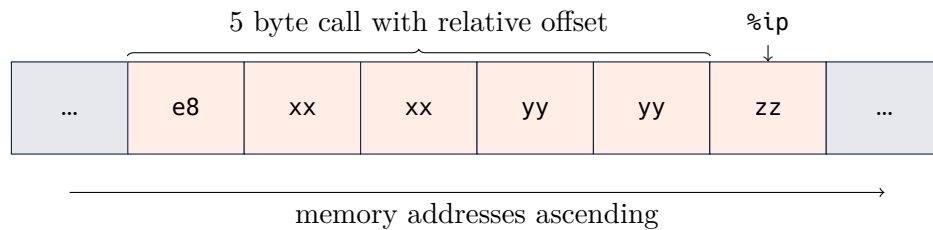


Abbildung 3.4: Ausschnitt aus dem Textsegment eines Programms vor dem Entfernen. Die fünf dem *Instruction Pointer* (%ip) vorangehenden Byte bilden einen Funktionsaufruf (*Call*) mit relativer Adressierung (Adressierung mittels *Offset*). Das Offset kann 16 (0xe8 xx xx 00 00) oder 32 (0xe8 xx xx yy yy) Bit lang sein. Der *x86\_64*-Standard kennt weitere Funktionsaufrufe mit teilweise von fünf Byte abweichenden Längen, die im Rahmen dieser Arbeit nicht behandelt werden. [Adv15, Int16a]

### 3.3 Entfernen der Instrumentierungsaufrufe

Das Entfernen der Instrumentierung aus dem Programm erfolgt durch Überschreiben der Programmteile, die für den Aufruf von *Score-P* verantwortlich sind. Der *x86\_64*-Befehlssatz kennt, wie die meisten Befehlssätze, sogenannte *No Operation*-Befehle (*NOPs*), die den Prozessor veranlassen, nichts zu tun. Moderne Prozessoren verarbeiten bis zu vier dieser *NOPs* pro Taktzyklus, ältere nur drei oder noch weniger [Fog16]. Die Länge des Zeitraums, in dem der Prozessor aufgrund eines *NOPs* nichts tut, ist also abhängig vom Modell und der Taktrate. Genutzt werden *NOPs* üblicherweise, um eine günstige Ausrichtung des Programms im Speicher zu erreichen oder Freiräume für das spätere Einfügen zusätzlicher Befehle zu schaffen. Das Plugin verwendet diese Befehle, um mit ihnen die Instrumentierungsaufrufe für die als irrelevant markierten Funktionen zu überschreiben.

Im *x86\_64*-Befehlssatz sind verschiedene Befehle zum Aufruf von Funktionen definiert, die sich sowohl in ihrer Semantik als auch in ihrer Länge unterscheiden [Adv15, Int16a]. Die Compiler der *GCC* sowie die *Intel*-Compiler verwenden jedoch in den Optimierungsstufen 01 (0) und 02 nur den Maschinenbefehl 0xe8. Dieser ist inklusive seiner Argumente fünf Byte lang und erlaubt den Sprung zu einer Adresse, die relativ zum letzten Byte des Sprungbefehls mittels eines sogenannten *Offsets* angegeben wird. Dieses Offset kann entweder 16 oder 32 Bit lang sein. Im ersten Fall werden die letzten zwei Byte des Sprungbefehls mit Nullen aufgefüllt (Abbildung 3.4). Aus Gründen, die in Kapitel 5 näher erläutert werden, behandelt das im Rahmen dieser Arbeit entwickelte Plugin nur diesen Funktionsaufruf.

Das Überschreiben des Funktionsaufrufs setzt voraus, dass das Programm Schreibrechte für sein Textsegment hat. Da dies aus Sicherheitsgründen ohne weiteres Zutun des Entwicklers nicht der Fall ist, muss zunächst mittels *mprotect* der Schreibschutz entfernt werden. Da *mprotect* die Schreibrechte nur pro Speicherseite (*Page*) ändert, werden zum Überschreiben eines Funktionsaufrufs die Schreibrechte für die Seite, auf der das erste Byte des Aufrufs liegt, sowie für die Seite, auf der das letzte Byte des Aufrufs liegt, angefordert. Da die Seitengrößen in realen Systemen zwischen 512 Byte und einem Gigabyte [TB15] liegen, kann sich der Funktionsaufruf

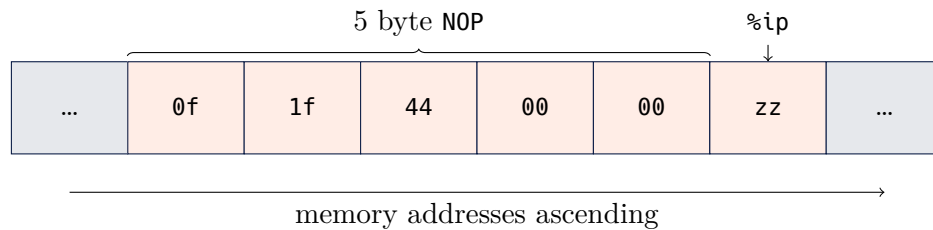


Abbildung 3.5: Ausschnitt aus dem Textsegment eines Programms nach dem Entfernen. Der Funktionsaufruf ist durch ein fünf Byte langes *No Operation* (NOP, `0x0f 1f 44 00 00`) ersetzt worden. Durch den Erhalt der Länge des Aufrufs wird die Verteilung des Programms im Speicher, die Einfluss auf die Ausführungsgeschwindigkeit nimmt, nicht geändert. Zudem werden nur jeweils ein *Instruction Fetch* und ein *Instruction Decode* benötigt. [Adv15, Int16a]

nicht über mehr als zwei Seiten erstrecken. Nach dem Überschreiben des Funktionsaufrufs wird das Schreibrecht wieder entfernt.

Für das Entfernen des Funktionsaufrufs bieten sich prinzipiell drei Möglichkeiten:

**Verkürzen des Speicherbereichs** Prinzipiell ist es möglich, die fünf Byte des Funktionsaufrufs zu entfernen und die darauffolgenden Teile des Textsegments dementsprechend um fünf Byte zu verschieben. Dies ist allerdings sehr aufwändig, da dafür das gesamte Textsegment eingelesen und kopiert werden muss und alle Sprungadressen neu berechnet werden müssen. Insbesondere für relative Adressen ist dies nicht trivial. Zudem wird durch dieses Vorgehen die vom Compiler durchgeführte Optimierung der Verteilung des Programms im Speicher invalidiert. Dieses Vorgehen kann also nicht mit vertretbarem Aufwand umgesetzt werden.

**Nutzung von fünf 1-Byte-NOPs** Um die zuvor genannten Probleme zu umgehen, kann der Funktionsaufruf mit fünf NOPs der Länge 1 Byte überschrieben werden. Zusätzlich zu der Zeit, in der der Prozessor durch das NOP angewiesen wartet, muss jedoch jedes NOP aus dem Speicher gelesen (*Instruction Fetch*) und dekodiert (*Instruction Decode*) werden. Dies kann zwar durch den sogenannten *μOP-Cache* und die *Pipeline* moderner Prozessoren nebenläufig und somit ohne negativen Einfluss auf die Programmlaufzeit geschehen. Dennoch wartet der Prozessor unter Umständen mehrere Takte und dadurch entsteht zusätzlicher Overhead.

**Nutzung eines 5-Byte-NOP** Um den zusätzlichen Overhead zu verringern, sieht der *x86\_64*-Standard auch längere Befehle mit der gleichen Semantik vor. Zusätzlich zu `0x90` gibt es NOPs mit einer Länge von zwei bis neun Byte. Für das Überschreiben des Funktionsaufrufs kann also das 5-Byte-NOP `0x0f 1f 44 00 00` genutzt werden, das nur ein *Instruction Fetch* und ein *Instruction Decode* benötigt und den Prozessor nur einen Takt warten lässt. [Adv15, Int16a]

Offensichtlich ist die Nutzung des 5-Byte-NOPs für diesen Anwendungszweck optimal. Dementsprechend überschreibt das Plugin die fünf Byte vor dem Zeiger, der durch den Algorithmus 3.2 gefunden wird (siehe Abschnitt 3.1), mit dem Wert `0x0f 1f 44 00 00` (Abbildung 3.5). Zusätzlich zu dem eigentlichen Sprung gehört jedoch noch mehr zu einem Funktionsaufruf. So

```

1  00000000004004a6 <foo>:
2    4004a6:    55                push   %rbp
3    4004a7:    48 89 e5          mov    %rsp,%rbp
4    4004aa:    89 7d fc          mov    %edi,-0x4(%rbp)
5    4004ad:    90                nop
6    4004ae:    5d                pop    %rbp
7    4004af:    c3                retq
8
9  00000000004004b0 <main>:
10   4004b0:    55                push   %rbp
11   4004b1:    48 89 e5          mov    %rsp,%rbp
12   4004b4:    bf 01 00 00 00    mov    $0x1,%edi
13   4004b9:    e8 e8 ff ff ff    callq  4004a6 <foo>
14   4004be:    b8 00 00 00 00    mov    $0x0,%eax
15   4004c3:    5d                pop    %rbp
16   4004c4:    c3                retq
17   4004c5:    66 2e 0f 1f 84 00 00 nopw   %cs:0x0(%rax,%rax,1)
18   4004cc:    00 00 00
19   4004cf:    90                nop

```

Quelltext 3.1: Ausschnitt aus einem beispielhaften Programm. Die Funktion *main* ruft die Funktion *foo* auf, die eine Ganzzahl (*int*) als Argument entgegen nimmt und keinen Rückgabewert hat (*void*). Zu sehen sind das Übergeben des Arguments an die Funktion *foo* durch Verschieben in das Register *%edi* in Zeile 12, der Aufruf der Funktion mit dem Sprungbefehl *0xe8* in Zeile 13, der Aufbau (Zeile 1 und 2/10 und 11) und der Abbau (Zeile 6/15) der Stack-Frames sowie die Übergabe von einem Rückgabewert durch Verschieben in das Register *%eax* in Zeile 14. Das Programm ist mit dem C-Compiler der *GCC* in Version 6.2.1 ohne Angabe weiterer Argumente übersetzt worden, die Darstellung ist mit *objdump* erzeugt worden. Die Zahlen in der linken Spalte geben jeweils die hexadezimale Adresse des ersten Bytes des Befehls an.

müssen Argumente übergeben und für jede aufgerufene Funktion ein neuer Stack-Frame angelegt werden. Das Übergeben des Arguments (*0x1*) geschieht im beispielhaften Programmausschnitt in Listing 3.1 in Zeile zwölf, indem es in das Register *%edi* verschoben wird, in dem es von der aufgerufenen Funktion *foo* erwartet wird. Anschließend folgt der eigentliche Sprung in Zeile 13. Das Offset für den Sprungbefehl *0xe8* ist in diesem Fall *0xe8 ff ff ff*, also -24, angegeben im *Little-Endian*-Format (beginnend mit dem niedrigstwertigen Byte). Ausgehend vom letzten Byte des Sprungbefehls (*0x40 04 bd*) ist die angesprungene Adresse *0x40 04 a6* in Zeile zwei, der Beginn der Funktion *foo*. Diese wiederum legt den benötigten Stack-Frame selbst an (Zeile zwei und drei) und löscht ihn vor dem Rücksprung wieder (Zeile 6). Von der Funktion *main* wird anschließend ihr eigener Rückgabewert (*0x0*) in das Register *%eax* (Zeile 14) verschoben und der Stack-Frame gelöscht (Zeile 15), bevor auch sie in die überliegende Funktion zurückspringt (Zeile 16). Die nachfolgenden Zeilen enthalten den schon zuvor angesprochenen Code, der vom Compiler eingefügt wird, um eine optimierte Ausrichtung des Programms im Speicher zu ermöglichen. Dieser wird nicht ausgeführt.

Durch das Übergeben von Argumenten und Rückgabewerten in speziellen Registern wie `%edi` und `%eax`, die nur für diesen Zweck reserviert sind, sowie durch die Tatsache, dass der benötigte Stack-Frame von der aufgerufenen Funktion selbst erzeugt und gelöscht wird, ist es möglich, nur den Funktionsaufruf selbst zu überschreiben und dennoch die sonstige Semantik des Programms zu erhalten. Das durch das Entfernen des Sprungbefehls obsolet gewordene Verschieben der Argumente in die entsprechenden Register stellt jedoch ein durch das Plugin nicht genutztes Optimierungspotential dar [HJM<sup>+</sup>13].

### 3.4 Synchronisierung mehrerer Threads

Parallelität kann auf modernen Systemen auf zweierlei Weise erreicht werden. Zum einen können mehrere Prozesse gestartet werden, die ihren Zustand sowie ihre Ergebnisse untereinander kommunizieren. Dieser Ansatz wird zum Beispiel von *MPI* genutzt. Für das im Rahmen dieser Arbeit entwickelte Plugin stellt die Prozessparallelität keine Schwierigkeit dar, da Prozesse voneinander getrennte Speicherbereiche für ihr Programm nutzen (*distributed memory*). Dementsprechend kann das Plugin in jedem Prozess autark laufen und Änderungen am Speicher vornehmen, ohne dabei Einfluss auf die anderen Prozesse zu nehmen. Hierdurch können zwar Inkonsistenzen bei den Filterentscheidungen entstehen, der korrekte Programmablauf ist jedoch zu jeder Zeit sichergestellt.

Auf der anderen Seite gibt es aber auch die Möglichkeit, Parallelität über Threads zu erreichen, wie es zum Beispiel bei der Nutzung von *OpenMP*<sup>3</sup> [DM98] geschieht. Threads teilen sich im Unterschied zu Prozessen einen Teil ihres Speichers (*shared memory*) und besitzen demnach ein gemeinsames Textsegment, sodass alle durch das Plugin verursachten Änderung sofort in allen Threads verfügbar sind. Hierdurch kann unter bestimmten Umständen der korrekte Programmablauf im Zusammenspiel mit *Score-P* gestört werden.

*Score-P* erwartet für jedes ausgelöste Enter-Event ein dazugehöriges Exit-Event. Bleibt dies aus, beendet es die Messung und das untersuchte Programm. Um dies zu umgehen, darf das Plugin keine Instrumentierungsaufrufe von Funktionen entfernen, in denen gleichzeitig ein weiterer Thread aktiv ist. Dies kann jedoch durch die eingeschränkte Sicht des Plugins auf die *Score-P* bekannten Informationen nicht sichergestellt werden. So ist es möglich, dass das interne Enter-Event in *Score-P* bereits ausgelöst worden ist, das Enter-Event im Plugin jedoch noch nicht. Dadurch würde das Plugin davon ausgehen, dass der Thread nicht in der entsprechenden Region aktiv ist und dementsprechend die Instrumentierungsaufrufe entfernen. *Score-P* würde jedoch auf das zugehörige Exit-Event warten, welches aufgrund der fehlenden Einsprünge in die Instrumentierung nicht ausgelöst werden würde. Löst nun der Thread das Exit-Event der aufrufenden Funktion aus, stellt *Score-P* einen invaliden Zustand fest und beendet die Messung. Da sich dieses Problem nicht umgehen lässt, verzichtet das Plugin darauf, in den parallelen Phasen des untersuchten Programms Veränderungen an diesem vorzunehmen. Stattdessen sammeln alle Threads während paralleler Programmabschnitte lediglich Informationen über die Funktionen des Programms und integrieren diese am Ende der Abschnitte. Veränderungen am

---

<sup>3</sup><http://openmp.org/>

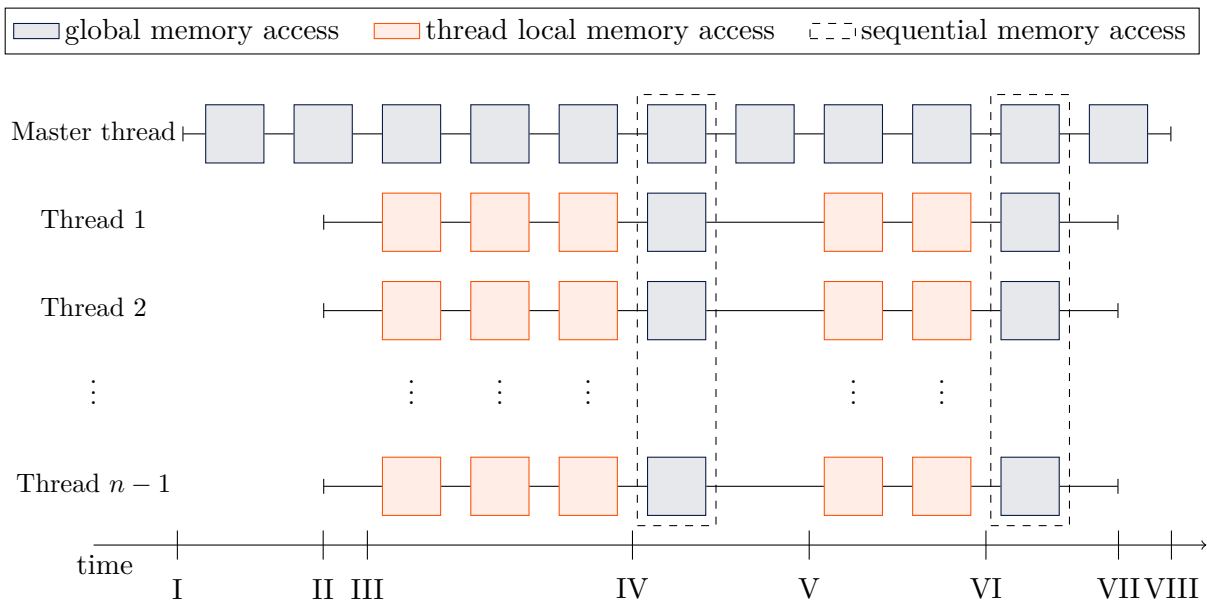


Abbildung 3.6: Schematische Darstellung der Datensynchronisierung innerhalb des Plugins. Nach dem Programmstart (I) werden alle Informationen über die Regionen zunächst in einem globalen Speicher abgelegt, der vom *Master Thread* verwaltet wird. Beim Anlegen der weiteren Threads (II) werden für jede Region im thread-lokalen Speicher (*Thread Local Storage*) Einträge für die Aufrufanzahl und -dauer erstellt. Am Ende jeder parallelen Phase des Programms (IV, VI) werden diese thread-lokal gesammelten Informationen in den globalen Speicher integriert. Beim Zerstören der Threads (VII) werden die thread-lokalen Speicher gelöscht, der globale Speicher bleibt bis zum Ende der Programmlaufzeit (VIII) erhalten.

Programm werden nur während der seriellen Phasen der Ausführung vorgenommen. Die dafür notwendige Synchronisierung der Threads folgt dem in Abbildung 3.6 dargestellten Schema. Nach dem Start des Programms (I) gibt *Score-P* grundlegende Informationen über alle Funktionen des untersuchten Programms an das Plugin weiter, das diese zusammen mit einem Aufrufzähler und einem Aufrufdauerakkumulator in einer globalen Hash-Tabelle ablegt. Der zu diesem Zeitpunkt schon existierende Thread (*Master Thread*) arbeitet ausschließlich auf dieser globalen Tabelle. Zu dem Zeitpunkt, an dem die restlichen Threads erstellt werden (*Location Creation*, II), wird für jeden dieser neuen Threads eine weitere Hash-Tabelle angelegt, in der für jede bis dahin bekannte Region die Aufrufanzahl und -dauer gespeichert werden. Hierdurch wird erreicht, dass in den Phasen der parallelen Ausführung des Programms (zwischen III und IV sowie zwischen V und VI) das Abspeichern der gesammelten Informationen ebenfalls parallel durchgeführt werden kann und somit keine Synchronisierung zwischen den Threads erfolgen muss. Lediglich nach dem Ende der nebenläufigen Abschnitte müssen die Threads ihre Ergebnisse sequenziell in den globalen Speicher integrieren. Dadurch, dass die thread-lokalen Speicher über mehrere parallele Phasen hinweg erhalten bleiben und erst beim Abbauen der Threads (*Location Deletion*, VII) gelöscht werden, wird der durch die Datenhaltung entstehende Overhead zusätzlich reduziert. Der globale Speicher wird erst beim Beenden des Programms (VIII) freigegeben.

## 4 Ergebnisse

Alle Messungen, die im Rahmen dieser Arbeit erfolgt sind, sind auf einem Testsystem mit zwei *Intel Xeon E5-2680 v3*<sup>1</sup> Prozessoren durchgeführt worden (je zwölf Kerne mit Hyperthreading, 2,5 bis 3,3 Gigahertz und 30 Megabyte Cache;  $8 \times 16$  Gigabyte DDR4-2133 RAM). *Score-P* ist aus den Quellen des *SVN*-Branches *TRY\_TUD\_substrates\_plugins* in Revisionsnummer 11245 gebaut worden, dessen Änderungen in Kürze in den Hauptentwicklungszweig einfließen sollen. Als Compiler sind die *GCC* in Version 6.1.0 respektive die *Intel-Compiler* in Version 16.0.2 eingesetzt worden. Als *MPI*-Implementierung ist *Open MPI*<sup>2</sup> [GFB<sup>+</sup>04] in Version 1.10.2 genutzt worden, *libunwind* in Version 1.1. Die Visualisierung der Ergebnisse ist mit *Vampir*<sup>3</sup> [KBD<sup>+</sup>08] in Version 9.1.0 erfolgt. Die verwendete Version des Plugins ist 1.0.2<sup>4</sup>.

Als Tests fungieren im Rahmen dieser Arbeit der „Block Tri-diagonal solver“ (*bt*), der „Scalar Penta-diagonal solver“ (*sp*) und der „Lower-Upper Gauss-Seidel solver“ (*lu*) jeweils in ihren *Multizone*-Varianten (*mz*) aus der „NAS Parallel Benchmarks“-Suite (*NPB*) in Version 3.3.1<sup>5</sup>. Sie können als serielle Variante oder unter Nutzung von *MPI* und *OpenMP* ausgeführt werden und stehen jeweils in unterschiedlichen Problemgrößen zur Verfügung, von denen die Größen A bis C genutzt werden. Als Programmiersprache kommt Fortran zum Einsatz. [BBB<sup>+</sup>91]

Als Beispiel für eine in C++ geschriebene Anwendung wird *LULESH*<sup>6</sup> aus der *CORAL*-Benchmark-Suite verwendet. Es bietet analog zu den *NPB* die Möglichkeit, die Problemgröße zu variieren, und kann ebenfalls unter Nutzung von *MPI* und *OpenMP* oder seriell ausgeführt werden. [KKN13]

Wenn nicht anders angegeben, zeigen die Ergebnisse jeweils die Werte aus zehn Versuchsdurchführungen, wobei die von den Benchmarks angegebene Ausführungszeit genutzt wird. Die vor-eingestellten Compiler-Optionen sind übernommen worden, nur die Optimierungsstufe ist von 03 auf 0 reduziert worden (siehe Abschnitt 3.3). Die Versuche mit den *NPB*-Benchmarks sind sowohl seriell als auch unter Nutzung von *MPI* (zwei Prozesse, jeweils ein Thread) sowie mit *MPI* und *OpenMP* (zwei Prozesse, jeweils zwölf Threads) durchgeführt worden. *LULESH* ist im Rahmen der Messungen seriell und unter Nutzung von *OpenMP* (zwölf Threads) ausgeführt worden.

### 4.1 Einfluss des Plugins auf die Ausführungsdauer

Der Hauptunterschied zwischen den bereits existierenden Filtermethoden (siehe Kapitel 2) und dem im Rahmen dieser Arbeit entwickelten Plugin ist, dass das Plugin die Instrumentierung

<sup>1</sup>[http://ark.intel.com/de/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2\\_50-GHz](http://ark.intel.com/de/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz)

<sup>2</sup><https://www.open-mpi.org/>

<sup>3</sup><https://www.vampir.eu/>

<sup>4</sup>[https://github.com/Ferruck/scorep\\_substrates\\_dynamic\\_filtering/tree/v1.0.2](https://github.com/Ferruck/scorep_substrates_dynamic_filtering/tree/v1.0.2)

<sup>5</sup><http://www.nas.nasa.gov/publications/npb.html>

<sup>6</sup><https://codesign.llnl.gov/lulesh.php>

für gefilterte Regionen nicht nur abkürzt, sondern die Instrumentierung aus dem Binärtext des Programms entfernt. Hierdurch soll der durch die Instrumentierung verursachte Overhead reduziert werden. Folglich soll zunächst der Einfluss des Plugins auf die Programmlaufzeit betrachtet werden.

Die im Folgenden dargestellten Ergebnisse vergleichen jeweils die Ausführungsdauer des uninstrumentierten Programms mit der der vollständig instrumentierten Version sowie unter Nutzung des im Rahmen dieser Arbeit entwickelten Plugins. Hierbei werden sowohl die absolute als auch die relative Metrik betrachtet, jeweils mit einem Grenzwert von 1000 Prozessortakten (*Cycles*). Zum Vergleich ist außerdem die integrierte Filterfunktion von *Score-P* gemessen worden. Hierbei sind diejenigen Funktionen in die Filterliste aufgenommen worden, die unter Nutzung des Plugins bei absoluter Metrik und einem Grenzwert von 1000 Cycles gefiltert worden sind (siehe Anhang). Die Benchmarks wie auch das Plugin und *Score-P* sind für die Messungen mit den Compilern der *GCC* übersetzt worden. Vergleichsmessungen für die *Intel*-Compiler werden im Anschluss betrachtet. Alle im Folgenden angegebenen Prozentwerte beziehen sich jeweils auf den Median der verglichenen Messwerte.

Die Ergebnisse für *NPB bt-mz A* (Abbildung 4.1a) zeigen, dass das Plugin im seriellen den Instrumentierungs-Overhead so weit reduziert, dass die Ausführungsgeschwindigkeit nahezu der uninstrumentierten Version (Median 35,47 s) entspricht. Dies gilt sowohl für die absolute Metrik (Median 36,61 s, Overhead 3,21 %) als auch für die relative (Median 36,57 s, Overhead 3,10 %). Das entspricht einer Reduktion des Instrumentierungs-Overheads um über 97 % gegenüber der vollständigen Instrumentierung mit *Score-P* (Median 74,43 s). Die Nutzung der Filterliste reduziert allerdings den Overhead noch stärker auf 2,85 % (Median 36,48 s).

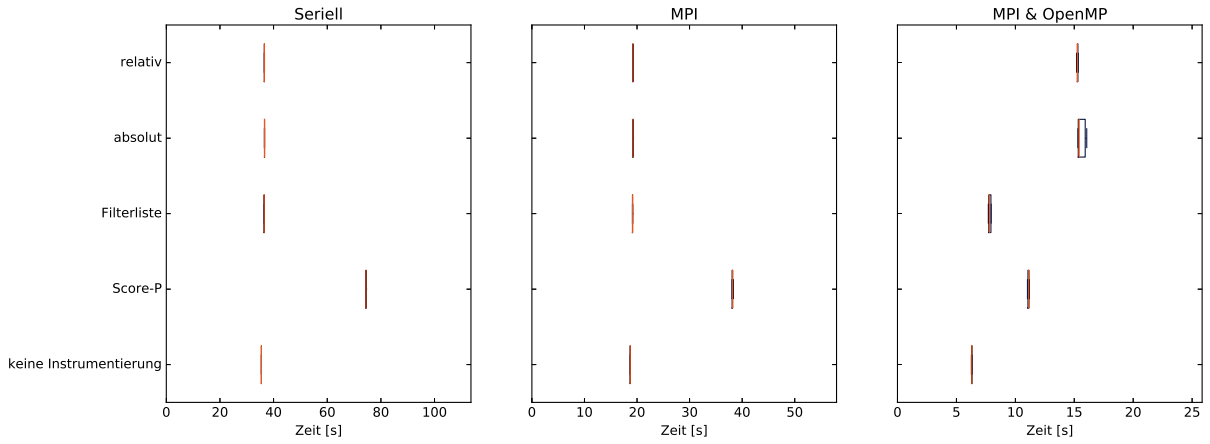
Ein ähnliches Bild zeigen die Ergebnisse bei der Nutzung von *MPI*. Auch hier reduzieren alle drei Filtermethoden den Overhead um circa 97 % gegenüber *Score-P* (Median 38,14 s) und erreichen so nahezu die Laufzeit des uninstrumentierten Programms (Median 18,71 s). Wieder erreicht die Filterliste die stärkste Reduktion auf einen Overhead von 2,35 %, die beiden Varianten des Plugins erreichen 2,73 %.

Anders stellen sich die Ergebnisse unter Hinzunahme der Threadparallelität dar. Hier schafft es das Plugin nicht, den von *Score-P* (Median 11,13 s) induzierten Overhead gegenüber der Version ohne Instrumentierung (Median 6,31 s) zu reduzieren. Im Gegenteil erhöht es ihn sogar um 89 % (absolut) beziehungsweise 86,10 % (relativ). Die Nutzung der Filterliste kann zwar den Overhead um 69 % verringern, bleibt somit aber ebenfalls deutlich hinter der Leistung im seriellen Fall und bei der ausschließlichen Nutzung von *MPI* zurück.

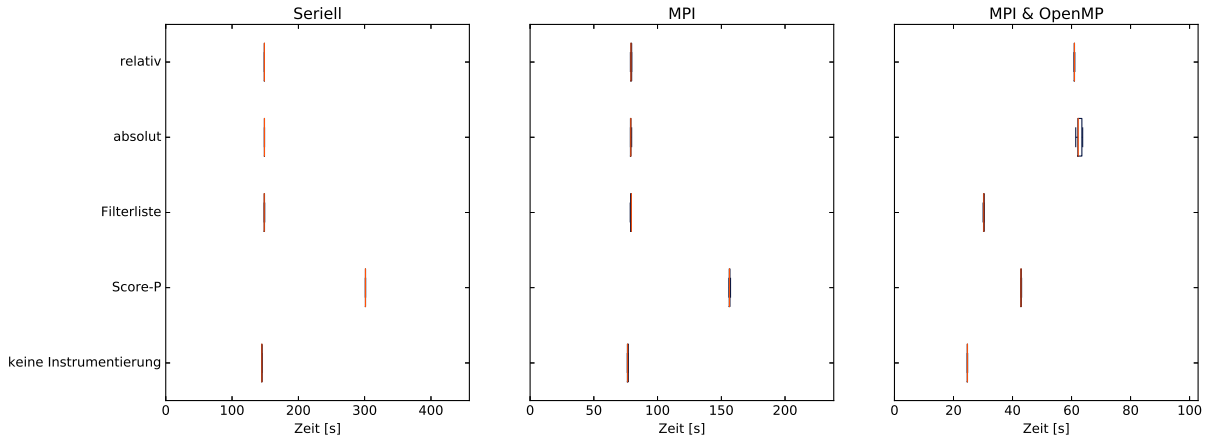
Bei der Betrachtung der Ergebnisse für *NPB bt-mz B* (Abbildung 4.1b) und C (Abbildung 4.1c) zeigt sich ein analoges Verhalten. Auch hier liegt die Overhead-Reduktion aller Filteransätze bei 96 bis 98 % für seriell und *MPI*, wobei jeweils *Score-P*s integrierte Filterfunktion etwas bessere Ergebnisse erzielt als das Plugin. Ebenso schafft es das Plugin für die Größen B und C nicht, den Overhead zu reduzieren, wobei auch die Reduktion durch die Filterliste weiter auf circa 66 % sinkt.

Die Ergebnisse für *sp-mz* (Abbildung 4.2) zeigen hingegen ein gänzlich anderes Bild. Der Overhead bei der seriellen Ausführung liegt für alle Problemgrößen bei unter 1 %, ohne dass darin ein Trend zu erkennen wäre. Dies gilt für die Werte von *Score-P* (A 0,31 %, B 0,1 %, C 0,41 %) wie

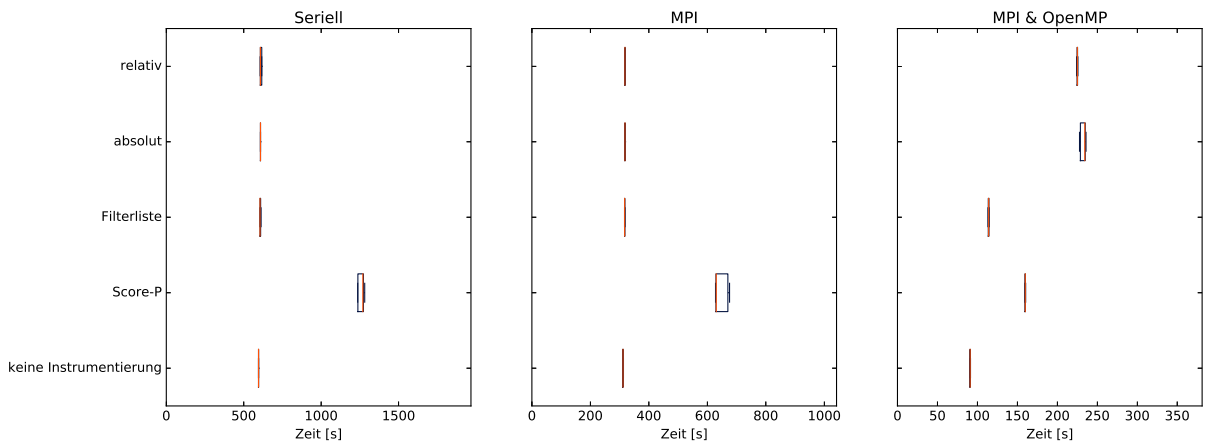




(a) Ergebnisse für die Problemgröße A.

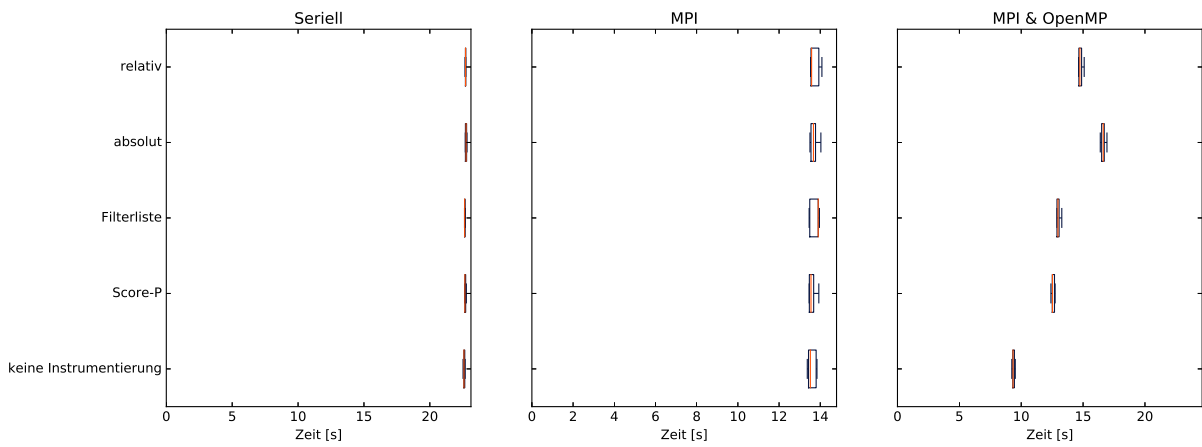


(b) Ergebnisse für die Problemgröße B.

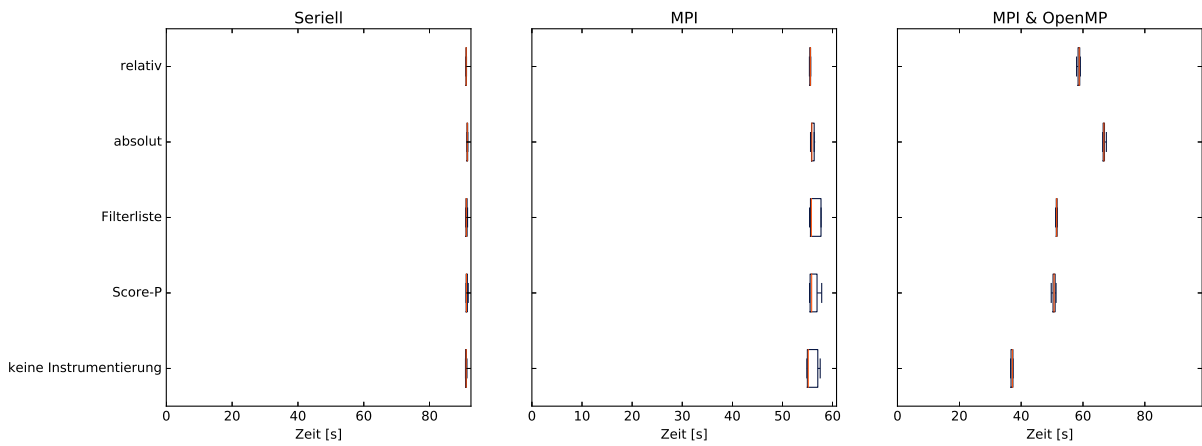


(c) Ergebnisse für die Problemgröße C.

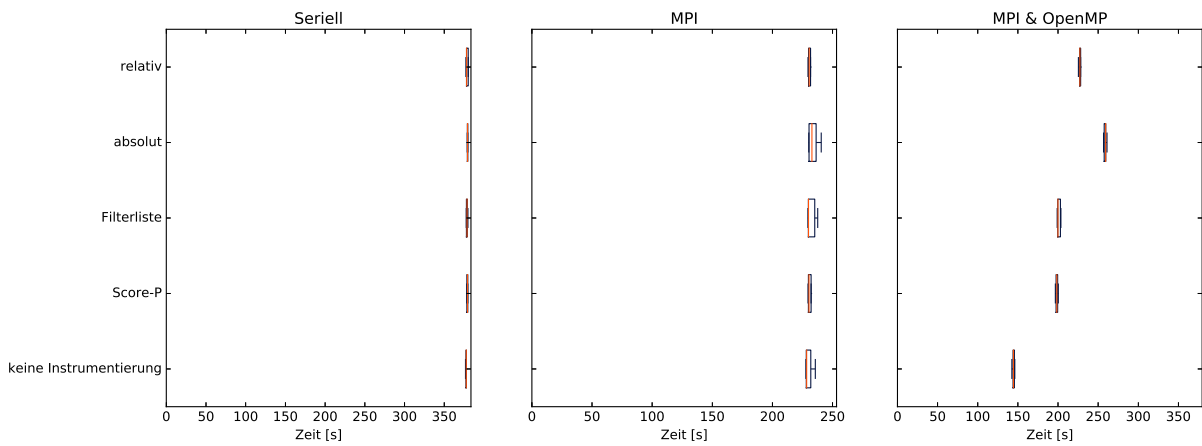
Abbildung 4.1: Ergebnisse für *NPB bt-mz* unter Nutzung des Fortran-Compilers der *GCC*. Die Werte unter *MPI* sind mit zwei Prozessen, die unter *MPI & OpenMP* mit zwei Prozessen und jeweils zwölf Threads erreicht worden.



(a) Ergebnisse für die Problemgröße A.



(b) Ergebnisse für die Problemgröße B.



(c) Ergebnisse für die Problemgröße C.

Abbildung 4.2: Ergebnisse für *NPB sp-mz* unter Nutzung des Fortran-Compilers der *GCC*. Die Werte unter *MPI* sind mit zwei Prozessen, die unter *MPI & OpenMP* mit zwei Prozessen und jeweils zwölf Threads erreicht worden.

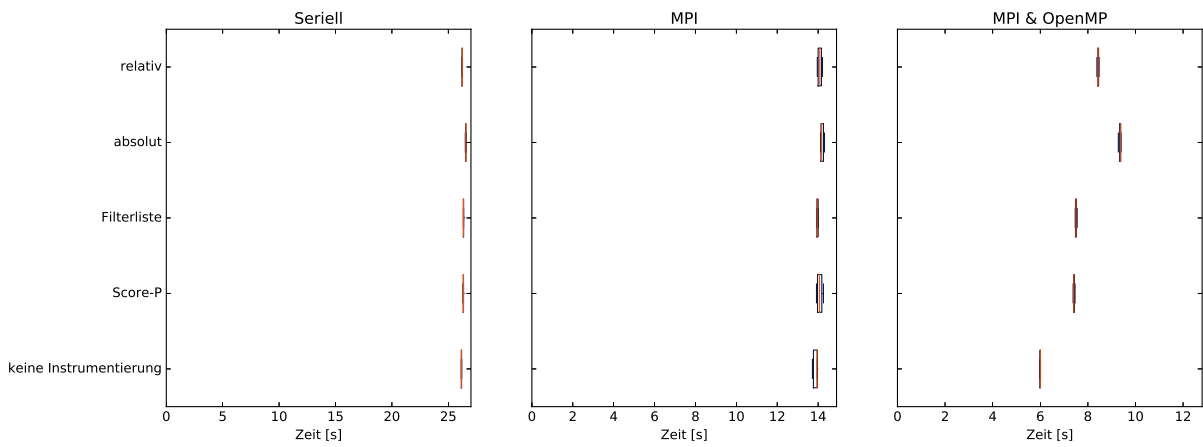
auch für die unter Nutzung der Filterliste (A 0,22 %, B 0,04 %, C 0,25 %) und die des Plugins (absolut: A 0,71 %, B 0,36 %, C 0,41 %; relativ: A 0,53 %, B 0,0 %, C 0,11 %). Ähnliche Beobachtungen können für die Ausführung unter Nutzung von *MPI* gemacht werden. Zwar steigt hier der Overhead auf bis zu 2,81 % (Filterliste, Problemgröße A), doch in Anbetracht der teils sehr kurzen Gesamtlaufzeiten, den zu erwartenden Messungenauigkeiten und des Fehlens eines erkennbaren Trends über die verschiedenen Problemgrößen hinweg kann sowohl aus den seriellen wie auch aus den prozessparallelen Messungen keine Aussage abgeleitet werden.

Auch für *sp-mz* schafft es das Plugin nicht, den Overhead bei der thread-parallelen Ausführung zu verringern. Im Gegenteil vergrößert es ihn um 112 bis 129 % (absolut) beziehungsweise 53 bis 69 % (relativ). Unter Nutzung von *Score-Ps* integrierter Filterfunktion steigt der Overhead im Unterschied zu *bt-mz* allerdings auch um 3 bis 14 %. Insgesamt ist der durch die Instrumentierung induzierte Overhead bei *sp-mz* mit circa 1 % bei serieller und prozessparalleler beziehungsweise 35 bis 38 % bei prozess- und thread-paralleler Ausführung auch ohne Filtern deutlich geringer als bei *bt-mz* (seriell und prozessparallel 100 bis 110 %, prozess- und thread-parallel circa 75 %). Die Ergebnisse für *lu-mz* (Abbildung 4.3) zeigen ein ähnliches Bild. Analog zu den Messwerten für *sp-mz* ist der Overhead in der seriellen Ausführung sehr gering. Er beträgt für die volle Instrumentierung mit *Score-P* 0,69 % (Größe A) beziehungsweise 0,46 % (Größe B & C) und bei der Nutzung der Filterliste 0,69 % (Größe A & C) beziehungsweise 0,55 % (Größe B). Bei der Nutzung des Plugins sind die Werte für die absolute Metrik etwas größer (A 1,53 %, B 0,64 %, C 0,54 %), für die relative Metrik tendenziell eher kleiner (A 0,31 %, B 0,38 %, C 0,60 %). Allerdings kann auch hier im Hinblick auf die Messungenauigkeit keine Aussage aus den Ergebnissen abgeleitet werden.

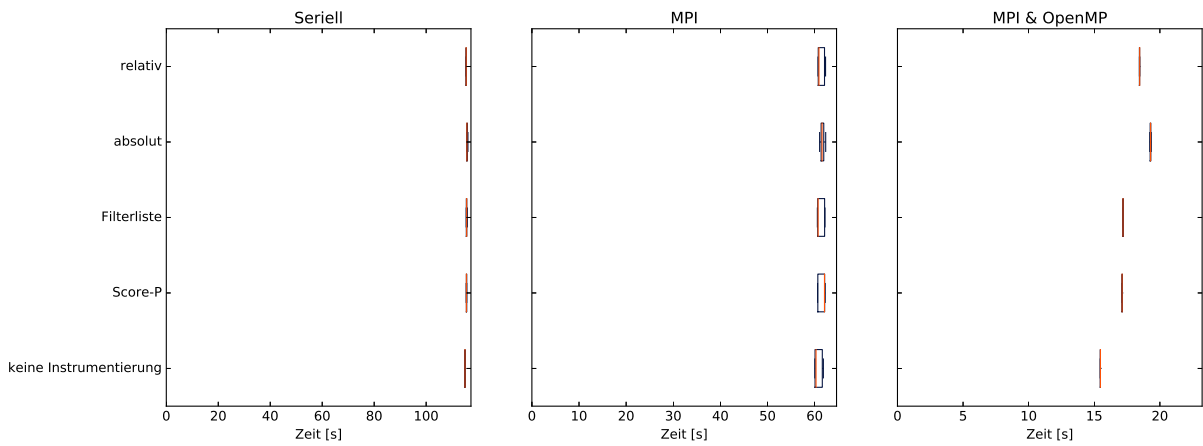
Die Messungen von *lu-mz* mit *MPI* zeigen ebenso ein analoges Verhalten zu *sp-mz*. Der Overhead mit voller Instrumentierung ist auch hier sehr gering (A 0,93 %, B 2,97 %, C 0,28 %) und wie bei *sp-mz* schaffen es alle drei Filtermethoden nicht, diesen signifikant zu verringern. Sowohl für die Filterliste (A 0,07 %, B 0,73 %, C 1,17 %) als auch für das Plugin (absolut: A 1,65 %, B 2,12 %, C 1,93 %; relativ: A 1,00 %, B 1,03 %, C 0,72 %) ist der gemessene Overhead jedoch ebenfalls sehr gering und es kann im Hinblick auf die zu erwartende Messungenauigkeit auch kein Trend zur Erhöhung des Overheads festgestellt werden.

Unter Hinzunahme der Thread-Parallelität erhöht sich der Overhead wie bei *sp-mz* durch jede der drei Filtermethoden. Allerdings zeigt sich eine Abnahme mit zunehmender Problemgröße. Dabei erzeugt die Nutzung der Filterliste einen deutlich geringeren Overhead (A 25,00 %, B 11,12 %, C 4,60 %) als das Plugin (absolut: A 56,00 %, B 24,69 %, C 9,25 %; relativ: A 40,50 %, B 19,33 %, C 7,78 %). Die Abnahme des Overheads über die Problemgröße ist auch für die vollständige Instrumentierung zu beobachten (A 23,50 %, B 10,67 %, C 4,45 %). Bei der Wertung dieser Ergebnisse ist allerdings die geringe Laufzeit der Größen A (Median 6,00 s) und B (Median 15,47 s) der Referenz, also des uninstrumentierten Programms, zu beachten.

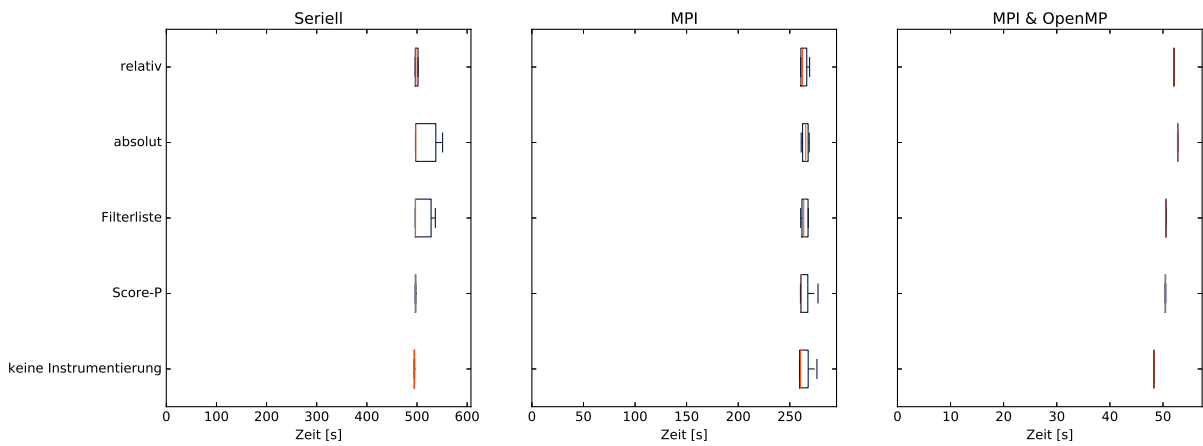
Die Messungen für *LULESH* als ein in C++ geschriebenes Beispielprogramm bestätigen die Ergebnisse der *NPB*-Benchmarks (Abbildung 4.4). In den seriellen Läufen wird durch die Instrumentierung mit *Score-P* ein Overhead von 9,04 % zu der Laufzeit des uninstrumentierten Programms (Median 31,97 s) hinzugefügt. Dieser wird durch die Nutzung der Filterliste um



(a) Ergebnisse für die Problemgröße A.



(b) Ergebnisse für die Problemgröße B.



(c) Ergebnisse für die Problemgröße C.

Abbildung 4.3: Ergebnisse für *NPB lu-mz* unter Nutzung des Fortran-Compilers der *GCC*. Die Werte unter *MPI* sind mit zwei Prozessen, die unter *MPI & OpenMP* mit zwei Prozessen und jeweils zwölf Threads erreicht worden.

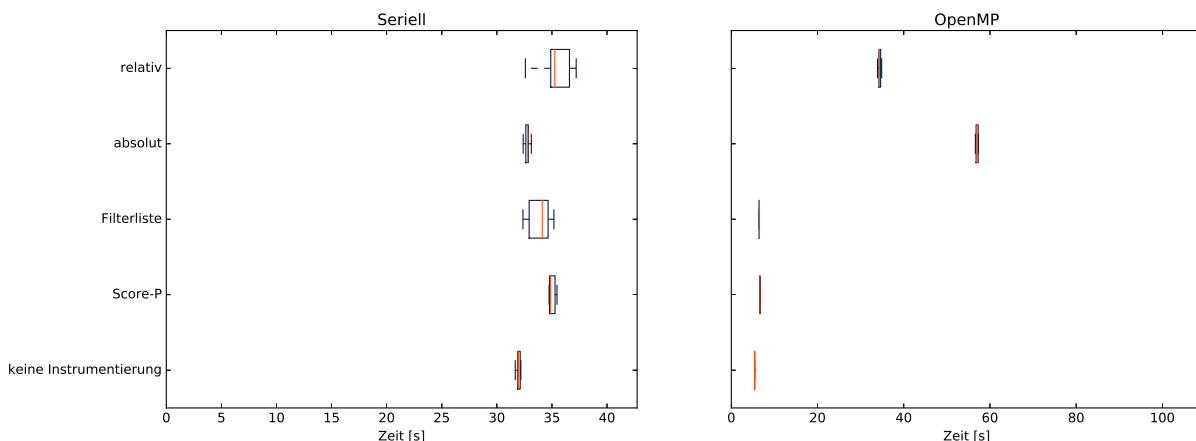


Abbildung 4.4: Ergebnisse für *LULESH*. Die gewählte Problemgröße ist 30. Neben der seriellen Variante ist der Benchmark auch unter Nutzung von *OpenMP* ausgeführt worden (12 Threads).

25,61 %, durch die Nutzung der absoluten Metrik des Plugins um 70,93 % reduziert. Die Verwendung der relativen Metrik erhöht den Overhead jedoch um 13,84 %.

Die Erhöhung des Overheads durch das Plugin bei der thread-parallelen Ausführung zeigt sich bei *LULESH* besonders deutlich. Der durch *Score-P* induzierte Overhead liegt mit 22,96 % im Rahmen der Ergebnisse der *NPB*-Benchmarks und auch die Reduktion durch die Filterliste zeigt mit 10,48 % einen vergleichbaren Wert. Die absolute Metrik des Plugins vergrößert den Overhead jedoch um 4058,87 %, die relative um 2233,87 %. Selbst bei Beachtung der geringen Laufzeit des uninstrumentierten Programms (Median 5,40 s) unterscheidet sich dieses Ergebnis deutlich von denen der *NPB*-Benchmarks.

Die Vergleichsmessungen für die *Intel*-Compiler mit der *NPB*-Suite (Abbildung 4.5) zeigen ein ähnliches Bild wie die Messungen mit den Compilern der *GCC*. Zwar sind die Ausführungszeiten insgesamt etwas geringer, der erzeugte Overhead sowie die Overhead-Reduktion sind jedoch vergleichbar. Im Unterschied zu den zuvor dargestellten Ergebnissen, ist die Filterwirkung bei *bt-mz* (Abbildung 4.5a) bei der seriellen Auswirkung etwas geringer. Zusätzlich zeigt das Plugin hier eine bessere Filterleistung (absolut 96,15 %, relativ 96,66 %) als die Filterliste (90,87 %). Selbiges gilt unter Nutzung von *MPI* (Plugin: absolut 97,93 %, relativ 97,97 %; Filterliste: 93,99 %). Bei Hinzunahme der Thread-Parallelität reduziert wiederum die Filterliste den Overhead am effektivsten (91,69 %), im Unterschied zu den Ergebnissen bei Nutzung der Compiler der *GCC* reduziert das Plugin ebenfalls den Overhead um 41,27 % (absolut), beziehungsweise 59,83 % (relativ). Hierbei muss jedoch wieder die geringe Gesamtlaufzeit der Referenz, also der uninstrumentierten Version, von im Median 2,71 s beachtet werden.

Für *sp-mz* (Abbildung 4.5b) und *lu-mz* (Abbildung 4.5c) können nur geringere Unterschiede zu den vorhergehenden Messungen festgestellt werden, abgesehen von der insgesamt kürzeren Laufzeit. Für diese beiden Benchmarks schafft es das Plugin wie zuvor nicht, den Overhead bei der Kombination von Prozess- und Thread-Parallelität zu verringern.

Vergleichsmessungen für den *LULESH*-Benchmark können mit dem *Intel*-Compiler nicht durchgeführt werden. Dieser erzeugt aus dem C++-Quelltext ein Programm, das teilweise mehrere

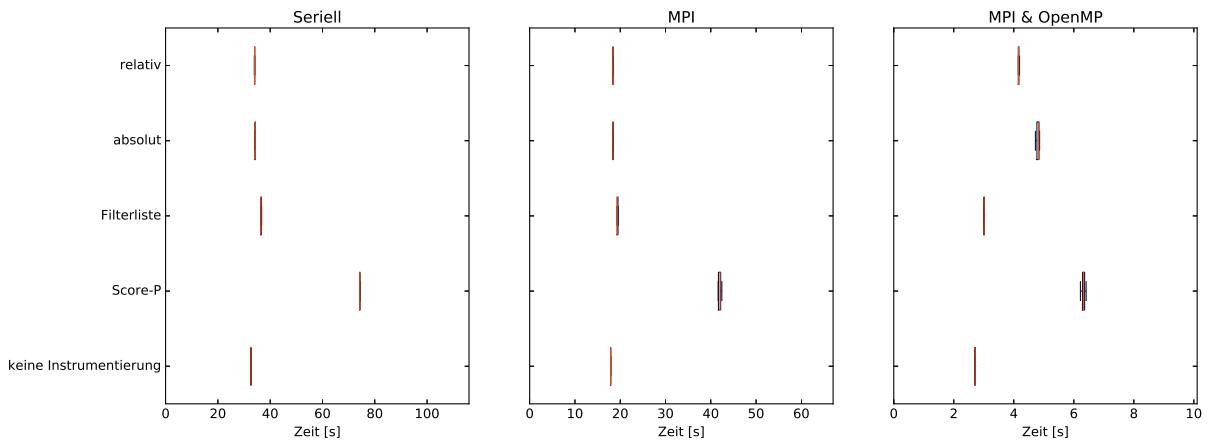
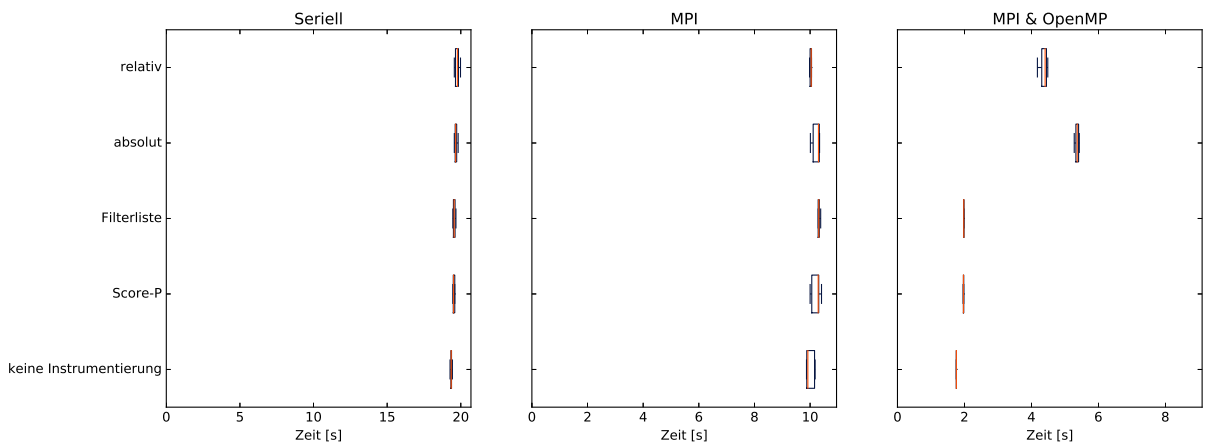
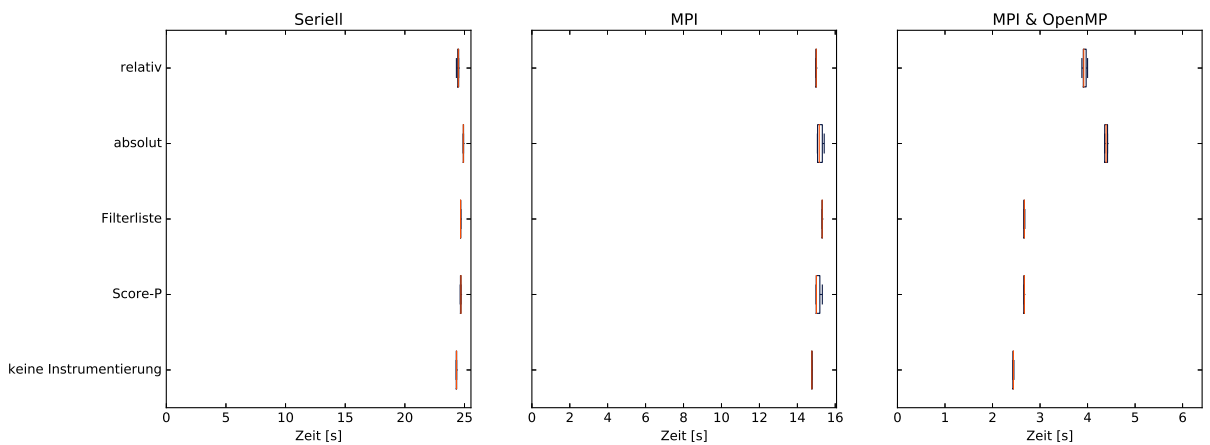
(a) Ergebnisse für *bt-mz*.(b) Ergebnisse für *sp-mz*.(c) Ergebnisse für *lu-mz*.

Abbildung 4.5: Ergebnisse der Vergleichsmessungen mit den *Intel*-Compilern (*NPB*). Dargestellt sind jeweils die Werte für die Problemgröße A sowohl seriell als auch unter Nutzung von *MPI* (zwei Prozesse) sowie *MPI* und *OpenMP* (zwei Prozesse, zwölf Threads).

Rücksprünge aus einer Funktion enthält. Aus Gründen, die in Kapitel 5 dargelegt werden, kann das Plugin nicht für ein solches Programm genutzt werden.

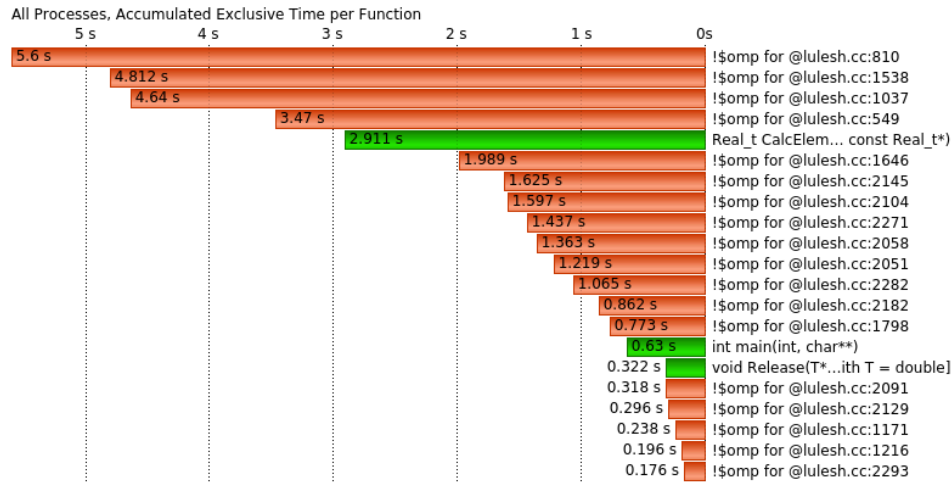
## 4.2 Einfluss des Plugins auf die erzeugten Traces

Das Hauptziel des Filterns von Instrumentierungsaufrufen ist, neben der Reduktion des Laufzeit-Overheads, die Verringerung der durch die Instrumentierung erzeugten Datenmenge. Die aus der Messung resultierende Trace-Datei wächst selbst für die im Rahmen dieser Arbeit genutzten, relativ kleinen Programme schnell auf einige hundert Megabyte bis hin zu über einhundert Gigabyte an. Da aber das Akquirieren von Informationen das Ziel der Instrumentierung ist und sich alle nutzbaren Informationen in der erzeugten Trace-Datei befinden, muss bei der Bewertung der Reduktion der Datenmenge auch immer der damit einhergehende Informationsverlust bewertet werden. Dementsprechend sollen im Folgenden nicht nur die Größen der entstehenden Trace-Dateien verglichen werden sondern auch eine qualitative Bewertung der Ergebnisse stattfinden.

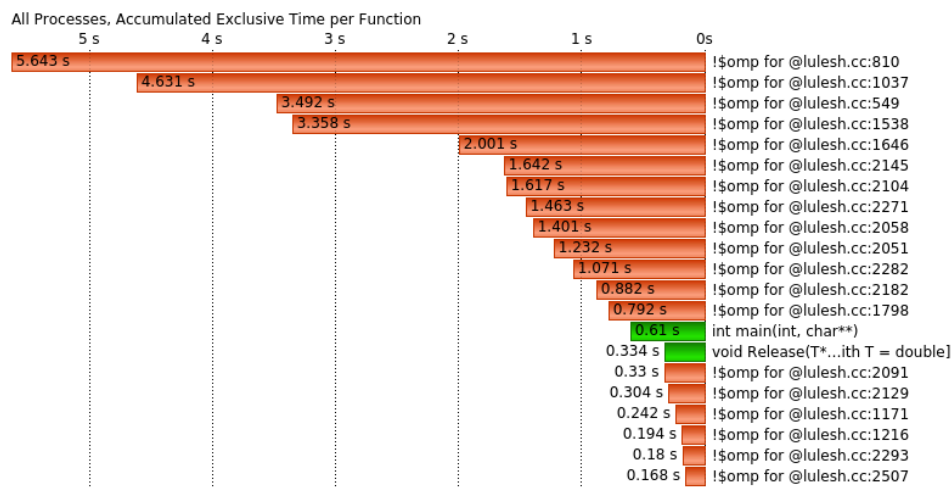
Der Vergleich der im entstehenden Trace enthaltenen Funktionen für den Benchmark *LULESH* in der seriellen Ausführung (Abbildung 4.6) zeigt ein zu erwartendes Ergebnis. Das Plugin filtert bei dem verwendeten Grenzwert von 1000 Cycles und unter Nutzung der absoluten Metrik nur die Funktion *CalcElemVolume* aus der Messung heraus. Dementsprechend erscheint sie nicht in der Darstellung der 21 Funktionen mit der akkumuliert höchsten exklusiven Laufzeit (Abbildung 4.6c), obwohl sie, wie die vollständig instrumentierte Messung (Abbildung 4.6a) zeigt, zu diesen gehört. Unter Nutzung der Filterliste erscheint die Funktion dementsprechend ebenfalls nicht in den Ergebnissen (Abbildung 4.6b), da die genutzte Filterdatei so gewählt worden ist, dass dieselben Funktionen gefiltert werden, die auch das Plugin bei absoluter Metrik und einem Grenzwert von 1000 Cycles filtert.

Das Filtern reduziert, unabhängig von der verwendeten Methode, die Größe der Trace-Datei von ungefähr 646 Megabyte auf circa 67 Megabyte, also um fast 90 %. Der direkte Vergleich zwischen dem ungefilterten und den gefilterten Ergebnissen zeigt, dass dennoch die wesentlichen Informationen im Trace erhalten bleiben: Mittels *OpenMP* parallelisierte *for*-Schleifen haben den größten Anteil an der Gesamtlaufzeit des Programms. Zwar weist *CalcElemVolume* die fünftlängste akkumulierte Gesamtlaufzeit auf, die weitere Untersuchung relativiert diesen Wert jedoch. Es ist mit über 25 Millionen Aufrufen die mit großem Abstand häufigst betretene Funktion des Benchmarks. Daraus resultiert, dass sie ebenfalls die Funktion mit der geringsten durchschnittlichen Laufzeit von circa 115 ns ist. Die durchschnittlichen Laufzeiten der sie in der Liste der akkumuliert längsten Funktionen umgebenden, *OpenMP*-parallelisierten *for*-Schleifen liegt hingegen bei 130  $\mu$ s bis 6 ms, also um mehr als den Faktor 1000 höher. Dementsprechend handelt es sich um eine hochfrequente Funktion (siehe Kapitel 2), die vom Plugin korrekt als solche erkannt und gefiltert worden ist.

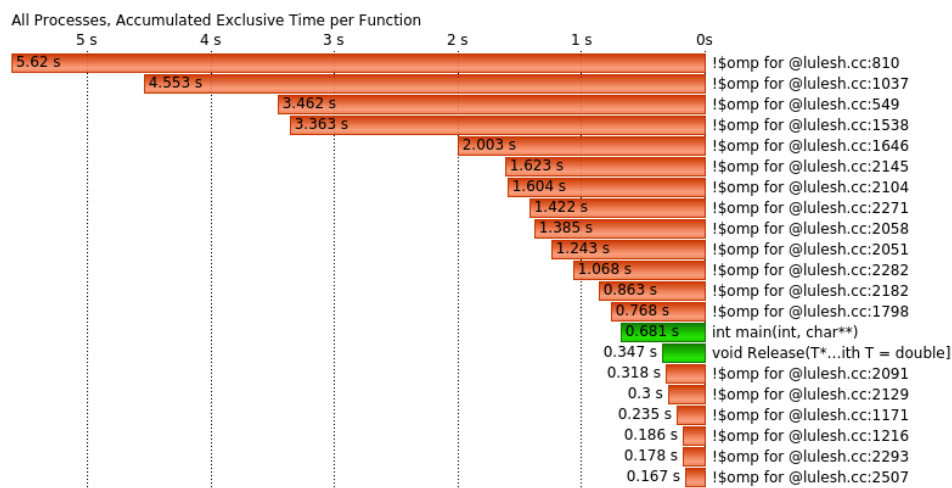
Die Trace-Datei, die unter Nutzung des Plugins gewonnen worden ist, enthält dennoch Informationen zu *CalcElemVolume*. Vor dem Einsetzen des Filterns sind von den eigentlich über 25 Millionen über 9400 Aufrufe gemessen worden. Zwar weicht deren durchschnittliche Laufzeit mit circa 400 ns von der tatsächlichen ab, die gesammelten Informationen genügen aber trotzdem



(a) Übersicht bei vollständiger Instrumentierung.



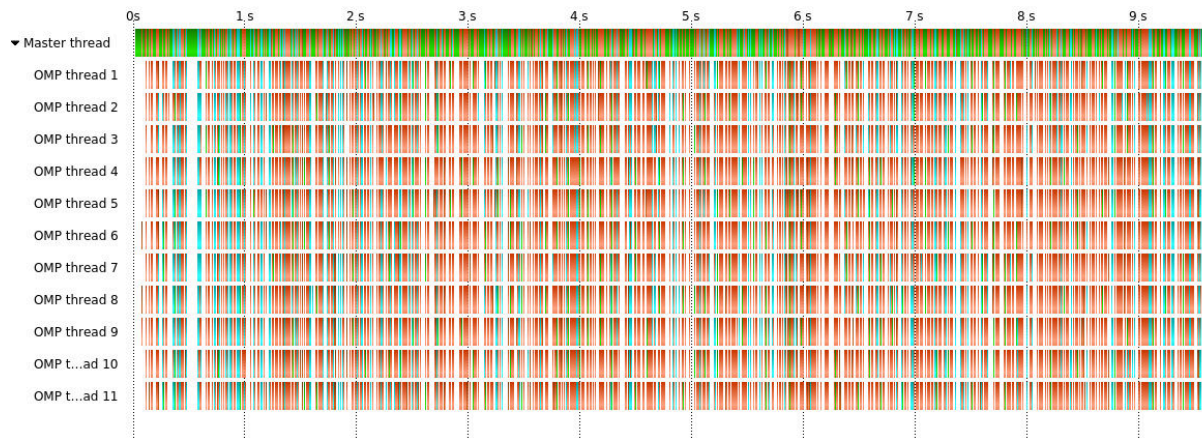
(b) Übersicht unter Nutzung der Filterliste.



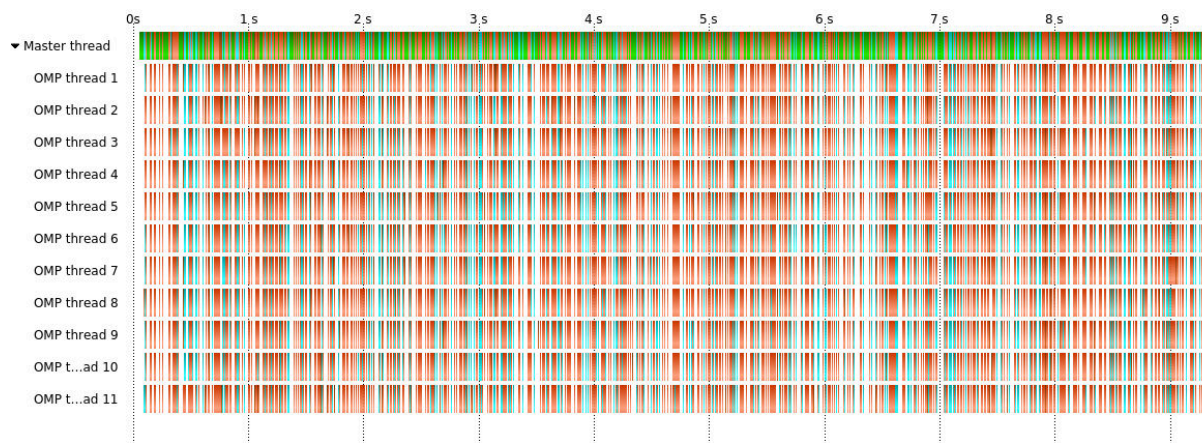
(c) Übersicht unter Nutzung des Plugins (absolute Metrik, Grenzwert 1000 Cycles).

Abbildung 4.6: Vergleich der in *Vampir* dargestellten Funktionen und ihrer Aufrufdauer. Gezeigt werden jeweils die 21 Funktionen mit den längsten akkumulierten Laufzeiten bei vollständiger Instrumentierung, unter Nutzung der Filterliste und unter Nutzung des Plugins bei absoluter Metrik und einem Grenzwert von 1000 Cycles für den *LULESH*-Benchmark (seriell).

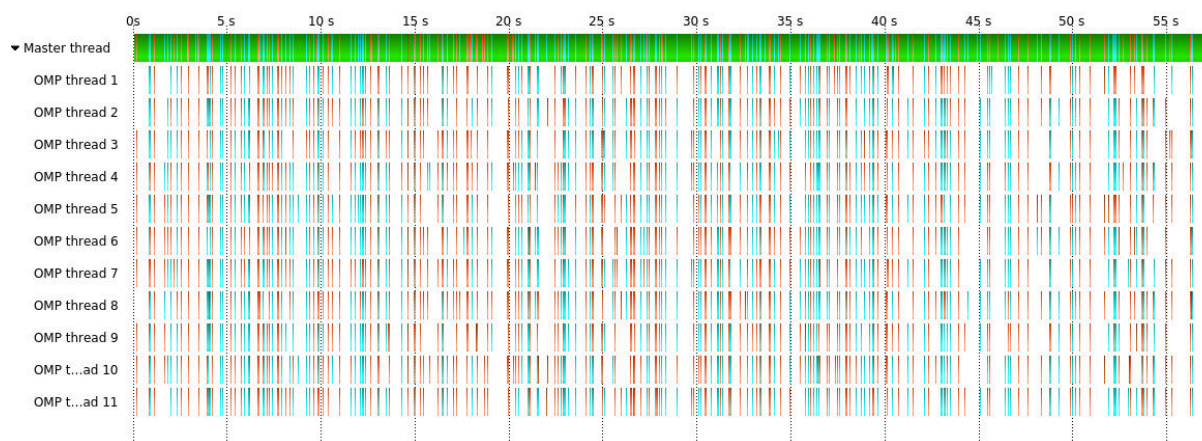




(a) Übersicht bei vollständiger Instrumentierung.



(b) Übersicht unter Nutzung der Filterliste.



(c) Übersicht unter Nutzung des Plugins (absolute Metrik, Grenzwert 1000 Cycles).

Abbildung 4.7: Vergleich der in *Vampir* dargestellten, zeitlichen Abläufe. Gezeigt wird jeweils *Vampirs* sogenannte *Master Timeline* für die vollständige Instrumentierung, unter Nutzung der Filterliste und unter Nutzung des Plugins bei absoluter Metrik und einem Grenzwert von 1000 Cycles für den *LULESH*-Benchmark (*OpenMP* mit zwölf Threads).

für eine grobe Bewertung und relative Einordnung der Funktion. Insbesondere charakteristische Aufrufmuster lassen sich erkennen, da bis zum Einsetzen des Filterns jeder Funktionsaufruf gemessen worden ist. Die Trace-Datei, die unter Nutzung der Filterliste erzeugt worden ist, enthält hingegen keinerlei Informationen über `CalcElemVolume`.

Deutlicher sind die Unterschiede im Vergleich der *Master Timelines* von *Vampir* (Abbildung 4.7). Diese zeigen alle erfassten Events in ihrer zeitlichen Abfolge und getrennt nach den verursachenden Threads. Die dargestellten Messungen für *LULESH* unter Nutzung von *OpenMP* mit zwölf Threads unterscheiden sich vor allem zwischen der vollständigen Instrumentierung mit *Score-P* und der Nutzung der Filterliste auf der einen und der Nutzung des Plugins bei absoluter Metrik und einem Grenzwert von 1000 Cycles auf der anderen Seite.

Das Filtern der Funktion `CalcElemVolume` durch die Filterliste hat nur einen geringen Einfluss auf das Aussehen der *Master Timeline*. Diese ist sowohl bei der alleinigen Nutzung von *Score-P* (Abbildung 4.7a) als auch bei der Nutzung der Filterliste (Abbildung 4.7b) dominiert von *OpenMP*-parallelisierten `for`-Schleifen (dargestellt in rot-braun) und *OpenMP*-Barrieren (dargestellt in blau). Dementsprechend ist es für den Nutzer in beiden Fällen leicht zu erkennen, dass diese einen erheblichen Einfluss auf die Programmlaufzeit haben.

Zwar werden auch unter Nutzung des Plugins sehr ähnliche Werte für die Anzahl der Aufrufe generiert, doch die Synchronisierung der Threads sorgt für eine deutliche Differenz in der Darstellung dieser Werte in der *Master Timeline* (Abbildung 4.7c). Nach jeder parallelen Phase des Programms muss diese Synchronisierung durchgeführt werden und der eigentliche Programmablauf wird erst dann fortgeführt, wenn diese abgeschlossen ist. Hierdurch entstehen Abschnitte, in denen keine Events erzeugt werden. Diese Abschnitte sind in der *Master Timeline* als weiße Lücken zwischen den eigentlichen Events dargestellt. Da aber der Kontext der umliegenden Funktion (grün dargestellt im *Master Thread*) dabei nicht verlassen wird, entsteht der Eindruck, dass in dieser Zeit Berechnungen in dieser Funktion durchgeführt würden. Es kommt also zur *Performance Perturbation*.

Die Größe der entstehenden Trace-Datei wird in diesem Beispiel durch beide Filtermethoden von ungefähr 1,2 Gigabyte auf circa 590 Megabyte verringert, also um etwas über 50 %. Die Reduktion ist hier geringer, da zusätzlich zu den Enter- und Exit-Events der Funktionen auch zu *OpenMP* gehörende Events aufgezeichnet werden. Diese werden aber beim Filtern nicht betrachtet. Der Anteil der Datenmenge, auf die das Filtern Auswirkungen hat, ist dementsprechend geringer, wodurch natürlich die Reduktion bezogen auf die Gesamtdatenmenge sinkt.

## 5 Diskussion

Ziel dieser Arbeit ist es gewesen, mit einem Plugin für *Score-P* die Instrumentierung von Funktionen zu entfernen, die für die Analyse eines Programms irrelevant sind und dabei auf Informationen zurückzugreifen, die zur Laufzeit des Programms ermittelt werden (siehe Kapitel 1). Umgesetzt worden ist dies nur für die Compiler-Instrumentierung, obwohl *Score-P* auch auf andere Instrumentierungsarten setzt, auf die sich der Ansatz analog übertragen lässt. Die Compiler-Instrumentierung verwendet, je nach Compiler, immer dieselben Funktionsaufrufe zum Einsprung in die Instrumentierung (siehe Abschnitt 3.1). Werden auch die anderen Instrumentierungsarten wie die manuelle Benutzerinstrumentierung oder die Bibliotheksinstrumentierung betrachtet, steigt die Komplexität des Erkennens und Auffindens der Instrumentierungsaufrufe stark, da sich dann verschiedene und variierende Instrumentierungsaufrufe im Programm befinden. Gleichzeitig trägt eine Umsetzung des Ansatzes auch für diese Arten der Instrumentierung aber kaum zur Evaluation seiner Möglichkeiten bei. Deshalb ist im Rahmen dieser Arbeit nur auf die Compiler-Instrumentierung eingegangen worden.

Eine weitere Einschränkung gilt für die nutzbaren Optimierungsstufen der Compiler. Das Plugin kann bisher nur mit dem fünf Byte langen `0xe8 e8 ff ff d0` Funktionsaufruf umgehen. In der Optimierungsstufe `03` setzen die Compiler der *GCC* und die *Intel*-Compiler aber auch andere Sprünge ein. Diese weisen eine variable Länge auf, beginnen mit `0xff` und können am Ende mit `0xff` aufgefüllt werden, um eine günstige Verteilung des Programms im Speicher zu erreichen. Anhand eines Beispiels lässt sich zeigen, warum der Umgang mit diesen Funktionsaufrufen mit der hier verwendeten Herangehensweise nicht möglich ist.

Betrachtet wird der fünf Byte lange Sprungbefehl `0xe8 e8 ff ff d0`. Beim Versuch diesen rückwärts, also vom gegebenen Instruction Pointer aus, zu erkennen, findet man zunächst den ebenfalls gültigen Befehl `0xff d0`, der auch ein Sprungbefehl ist (`call %eax`). Nun ist es ein Leichtes, einen passenden vorhergehenden Befehl zu finden, der den Rest des eigentlichen Funktionsaufrufs enthält. So könnte vor dem Sprung an die in Register `%eax` gegebene Adresse das Ganzzahlargument `-1513472` an die aufgerufene Funktion übergeben werden, indem es in das Register `%edi` verschoben wird: `0xbf 00 e8 e8 ff ff d0`. So ergibt sich insgesamt `0xbf 00 e8 e8 ff ff d0`, eine Sequenz, in der sich nicht sicher feststellen lässt, welcher Funktionsaufruf verwendet wird (Abbildung 5.1). Dieses Vorgehen lässt sich beliebig fortführen und kann analog auch verwendet werden, um die Problematik für Funktionsaufrufe zu zeigen, die länger als fünf Byte sind.

Die Probleme sind also der fehlende Kontext beim Einlesen des Binärtextes in umgekehrter Reihenfolge sowie die Mehrdeutigkeit von *CISC*-Befehlssätzen (jede Bitfolge, die einen Befehl darstellt, kann auch als Argument eines Befehls verwendet werden). Eine Möglichkeit, diese Probleme zu umgehen, ist, zu Beginn der Programmausführung den vollständigen Binärtext einmalig vorwärts einzulesen. Durch das Lesen in der korrekten Reihenfolge eliminiert der so gewonnene Kontext die Mehrdeutigkeiten, es können also alle vorkommenden Funktionsaufrufe

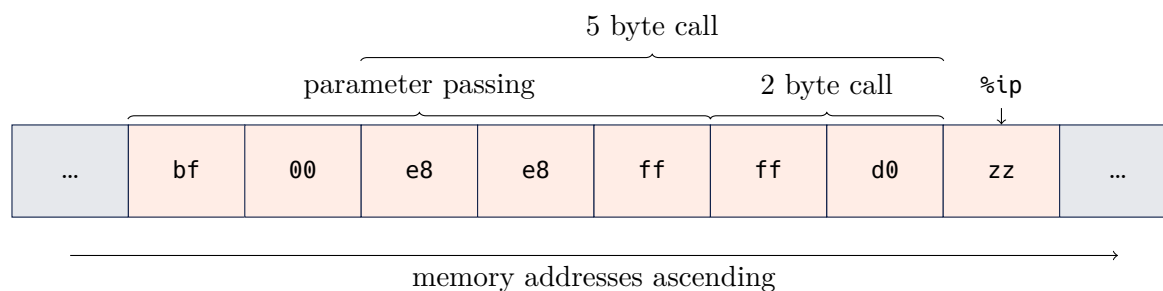


Abbildung 5.1: Verdeutlichung des Problems der variablen Sprungbefehlslänge. Durch das Lesen des Binärtextes in umgekehrter Reihenfolge, ausgehend vom Instruction Pointer, fehlt der Kontext, der zur eindeutigen Identifizierung von Assemblerbefehlen notwendig ist.

erkannt und gespeichert werden. Bei der Suche nach den entsprechenden Funktionsaufrufen zur Programmlaufzeit kann dann der Sprungbefehl gewählt werden, der am nächsten vor dem durch den Algorithmus 3.2 gefundenen Instruction Pointer liegt. Dieses Vorgehen konnte aufgrund des begrenzten zeitlichen Rahmens dieser Arbeit nicht umgesetzt werden, weshalb das Plugin in dieser Version nicht mit anderen Funktionsaufrufen als `0xe8` umgehen kann.

Eine weitere vom Plugin bisher nicht behandelte Eigenschaft von Programmen ist, dass der Compiler mehrere `return`-Anweisungen innerhalb einer Funktion nicht unbedingt immer zu einer einzelnen zusammenfasst (Quelltext 5.1). Da *Score-P* die Instrumentierung über den Compiler einfügt, kann es leicht alle Rücksprünge aus einer Funktion instrumentieren. Dem Plugin steht jedoch nicht dasselbe Wissen wie dem Compiler zur Verfügung, daher findet der Algorithmus 3.2 nur einen Instrumentierungsaufwurf. Es ist derjenige, der zu dem Ausgang der Funktion gehört, der genutzt worden ist, als das Plugin die Entscheidung zum Entfernen getroffen hat. Wird die Region später erneut ausgeführt und dabei ein anderer Rücksprung genutzt, wird deshalb ein Exit-Event generiert, zu dem es kein passendes Enter-Event gibt. Dieses Problem lässt sich analog zum vorher beschriebenen Finden aller Funktionsaufrufe durch ein einmaliges Einlesen des kompletten Binärtextes beim Programmstart lösen.

Die größte Herausforderung, die die derzeitige Implementierung aufzeigt, sind jedoch die Datenhaltung und die aus ihr resultierende, notwendige Synchronisierung der Threads am Ende jeder parallelen Phase des untersuchten Programms (siehe Abschnitt 3.4). Beides führt dazu, dass die Ausführungsdauer teilweise größer ist als bei der Verwendung von *Score-P* ohne das Plugin (siehe Abschnitt 4.1). Dies bestätigt auch die Analyse mit *perf* [Mel10], einem Werkzeug zur Leistungsanalyse von Programmen, das im Rahmen des Linux-Kernels entwickelt wird und mit dem verschiedene Metriken zu einem Programm gewonnen werden können. Eine davon heißt *cache references* [Int16b] und gibt die Anzahl der Zugriffe des Programms auf den sogenannten *last level cache*, also den in der Cache-Hierarchie höchsten und langsamsten Cache [HP12], an. Vergleicht man diese zwischen der instrumentierten Programmausführung ohne beziehungsweise mit Nutzung des Plugins, zeigt sich, dass die Anzahl der *cache references* durch die Nutzung des Plugins um über 100 % steigt (*NPB bt-mz A* mit zwei Prozessen, wobei nur einer mit *perf* untersucht worden ist, und je zwölf Threads: 369 695 438 zu 810 438 510, Steigerung um circa 120 %). Die weitere Untersuchung mit *perf* verdeutlicht, dass die meisten der Anfragen an den

```

1 431530:      e8 cb 3a 00 00      callq 435000 <__VT_IntelExit>
2 431535:      33 c0                xor    %eax,%eax
3 431537:      48 8d bc 24 58 02 00  lea     0x258(%rsp),%rdi
4 43153e:      00
5 43153f:      e8 bc 3a 00 00      callq 435000 <__VT_IntelExit>
6 431544:      33 c0                xor    %eax,%eax
7 431546:      48 8d bc 24 20 02 00  lea     0x220(%rsp),%rdi
8 43154d:      00
9 43154e:      e8 ad 3a 00 00      callq 435000 <__VT_IntelExit>
10 431553:      33 c0                xor    %eax,%eax
11 431555:      48 8d bc 24 f8 01 00  lea     0x1f8(%rsp),%rdi
12 43155c:      00
13 43155d:      e8 9e 3a 00 00      callq 435000 <__VT_IntelExit>
14 431562:      33 c0                xor    %eax,%eax
15 431564:      48 8d bc 24 f0 01 00  lea     0x1f0(%rsp),%rdi
16 43156b:      00
17 43156c:      e8 8f 3a 00 00      callq 435000 <__VT_IntelExit>
18 431571:      33 c0                xor    %eax,%eax
19 431573:      48 8d bc 24 e8 01 00  lea     0x1e8(%rsp),%rdi
20 43157a:      00
21 43157b:      e8 80 3a 00 00      callq 435000 <__VT_IntelExit>
22 431580:      33 c0                xor    %eax,%eax
23 431582:      48 8d bc 24 e0 01 00  lea     0x1e0(%rsp),%rdi
24 431589:      00
25 43158a:      e8 71 3a 00 00      callq 435000 <__VT_IntelExit>
26 43158f:      33 c0                xor    %eax,%eax
27 431591:      48 8d bc 24 d8 01 00  lea     0x1d8(%rsp),%rdi
28 431598:      00
29 431599:      e8 62 3a 00 00      callq 435000 <__VT_IntelExit>

```

Quelltext 5.1: Ausschnitt aus einem vom *Intel*-Compiler übersetzten C++-Programm. Es handelt sich um einen Teil des *LULESH*-Benchmarks. Aufgrund der mehrfachen Aufrufe der zum Exit-Event gehörenden Instrumentierungsfunktion `__VT_IntelExit`, ist der im Rahmen dieser Arbeit verwendete Algorithmus 3.2 nicht ausreichend, um die Instrumentierung aus dem Programm zu entfernen.

*last level cache* durch das Plugin verursacht werden. Die Funktion `on_team_end`, in der die in Abschnitt 3.4 vorgestellte Synchronisierung der Threads durchgeführt wird, ist dabei für den größten Teil der Zugriffe verantwortlich (Tabelle 5.1).

Diese Funktion iteriert in jedem Thread über die gespeicherten Informationen zu allen dem Thread bekannten Funktionen und integriert diese in den lokalen Speicher. Sowohl der thread-lokale als auch der globale Speicher sind als Hash-Tabellen ausgeführt, in denen unter der Annahme, dass keine Kollisionen auftreten, mit  $\mathcal{O}(1)$  auf die Elemente zugegriffen werden kann. Aufgrund der relativ geringen Anzahl an gespeicherten Regionen und den im Plugin getroffenen Vorkehrungen kann von dieser Annahme in den meisten Fällen ausgegangen werden. Dementsprechend werden  $n$  Speicherzugriffe pro Aufruf von `on_team_end` beziehungsweise  $n \cdot m$  pro Synchronisierung benötigt ( $n$ : Anzahl der den Threads bekannten Regionen;  $m$ : Anzahl der Threads). Da Zugriffe auf den *last level cache* gemessen an der benötigten Zeit sehr teuer sind,

Tabelle 5.1: Anteil der Zugriffe auf den *last level cache* für *NPB bt-mz A* (zwei Threads, wobei nur einer mit *perf* untersucht worden ist, jeweils zwölf Threads). Die Werte sind mit *perf* und der Metrik *cache references* ermittelt worden. Es werden nur die Ergebnisse mit den fünf größten Anteilen gezeigt. Deutlich zeigt sich, dass die Funktion *on\_team\_end*, die zur Synchronisierung der Daten zwischen den Threads genutzt wird, die meisten Zugriffe auf diesen Speicher verursacht.

Anteil	Herkunft	Symbol
3,25 %	libdynamic_filtering_plugin.so	on_team_end
2,45 %	bt-mz.A.2	binvcrhs_
2,32 %	bt-mz.A.2	MAIN__
1,60 %	libpthread-2.23.so	pthread_mutex_unlock
1,44 %	bt-mz.A.2	otf2_snap_reader_read
⋮	⋮	⋮

kann hierdurch der Großteil des durch das Plugin erzeugten Overheads bei thread-paralleler Ausführung erklärt werden. Zusätzlich müssen diese Zugriffe in serieller Form ausgeführt werden, um Dateninkonsistenzen zu verhindern. Das dafür notwendige Blockieren und Warten der Threads erzeugt zusätzlichen Overhead.

Nötig sind die Datenhaltung und Synchronisierung aufgrund des gegenüber *Score-P* eingeschränkten Wissens des Plugins über den aktuellen Zustand (siehe Abschnitt 3.4). Da das Plugin aber keinerlei Informationen akquiriert und benötigt, die *Score-P* intern nicht auch benutzt, ist also die logische Konsequenz, die Funktionalität des Plugins direkt in *Score-P* zu integrieren. Hierdurch wird nicht nur die zusätzliche Synchronisierung der Threads obsolet, es müssen auch keine Daten doppelt erhoben werden und zum Finden aller Funktionsaufrufe und Rücksprünge müsste nicht der gesamte Binärtext eingelesen werden. Zudem könnten die von *Score-P* verwendeten Filterlisten so erweitert werden, dass diese ebenfalls die Einsprünge in die Instrumentierung aus dem Programm entfernen. So würde die dem Benutzer bereits bekannte Vorgehensweise deutlich effizienter arbeiten, ohne dass dieser sich umgewöhnen müsste. Das durch das Plugin bereitgestellte Filtern könnte als Alternative angeboten werden.

## 6 Zusammenfassung & Ausblick

Im Rahmen dieser Arbeit ist ein Prototyp erarbeitet worden, der auf der Grundlage von Laufzeitstatistiken die Instrumentierung eines Programms anpasst. Hierfür werden aus der Aufrufdauer und der Anzahl der Aufrufe der Funktionen des Programms Metriken errechnet, anhand derer die Relevanz einer Funktion bewertet wird. Zur Programmlaufzeit können auf der Grundlage einer dieser Metriken irrelevante Funktionen aus der Datenakquise ausgenommen werden. Dies geschieht durch das Überschreiben der entsprechenden Einsprünge in die Instrumentierung mit NOPs.

Die in Kapitel 4 dargestellten Ergebnisse zeigen das Potential des Ansatzes. Durch das Überschreiben der Instrumentierungsaufrufe im Binärtext kann der durch die Instrumentierung erzeugte Overhead teils deutlich reduziert werden. Gleichzeitig kann die Nutzung der Laufzeitinformationen für eine automatisierte Auswahl der zu filternden Funktionen genutzt werden. Bei geeigneter Wahl des Grenzwertes für die vom Plugin verwendeten Metriken werden dennoch die wesentlichen Informationen durch die Messung gewonnen. Das in Kapitel 1 gesetzte Ziel einer Lösung des Typs D2 kann also prinzipiell als erreicht angesehen werden.

Trotzdem bleibt weiteres Verbesserungspotential offen. Es werden die gesteckten Ziele nur unter bestimmten Umständen erreicht und einige Grenzfälle werden vom Plugin bisher nicht betrachtet. Insbesondere die Herausforderungen im Bereich der Messung thread-paralleler Anwendungen müssen angegangen werden, da sowohl Prozess- als auch Thread-Parallelität wichtige Werkzeuge bei der Entwicklung effizienter und performanter Programme im Bereich des Hochleistungsrechnens sind. Auf einige der möglichen Ansätze zur Verbesserung und Fortentwicklung des Plugins ist in Kapitel 5 detailliert eingegangen worden.

Besonders auffällig ist, dass es das Plugin nicht schafft, den Overhead stärker zu reduzieren als die integrierte Filterfunktion von *Score-P*, obwohl der gewählte Ansatz prinzipiell schneller ist. Dies ist der notwendigen doppelten Datenhaltung sowie der Synchronisierung zwischen mehreren Threads geschuldet, wie in Kapitel 5 gezeigt worden ist. Die optimale Implementierung des Ansatzes kann daher nur die Integration der Vorgehensweise in *Score-P* sein. Selbst nach der Lösung der in Kapitel 5 dargelegten Herausforderungen verbleibt durch die Umsetzung in Form eines Plugins vermeidbarer Overhead. Die Erfassung und das Speichern der Informationen, die für die Implementierung des Ansatzes nötig sind, finden sowohl in *Score-P* als auch im Plugin selbst statt. Die Erweiterung muss außerdem zusätzliche Daten akquirieren und Synchronisierungen durchführen, nur um Informationen wiederherzustellen, die in der Schnittstelle zwischen *Score-P* und dem Plugin verloren gehen. Durch eine Integration der Funktionalität kann dieser Overhead vermieden werden und nur so kann eine effiziente Lösung erreicht werden.

Trotz dieser Kritik bleibt festzuhalten, dass das im Rahmen dieser Arbeit entwickelte Plugin die prinzipiellen Möglichkeiten des Ansatzes zeigt und somit die ursprüngliche Aufgabe umsetzt. Es bildet eine gute Grundlage für die weitere Forschung und Entwicklung.



## Anhang

```
1 SCOREP_REGION_NAMES_BEGIN
2 EXCLUDE lhsinit
3     exact_solution
4     binvrhs
5     binvcrhs
6     matmul_sub
7     matvec_sub
8 SCOREP_REGION_NAMES_END
```

Quelltext A.1: *Score-P*-Filterdatei für *NPB bt-mz*, seriell.

```
1 SCOREP_REGION_NAMES_BEGIN
2 EXCLUDE lhsinit
3     exact_solution
4     binvrhs
5     binvcrhs
6     matmul_sub
7     matvec_sub
8 SCOREP_REGION_NAMES_END
```

Quelltext A.2: *Score-P*-Filterdatei für *NPB bt-mz*, *MPI*.

```
1 SCOREP_REGION_NAMES_BEGIN
2 EXCLUDE lhsinit
3     exact_solution
4     binvrhs
5     binvcrhs
6     matmul_sub
7     matvec_sub
8 SCOREP_REGION_NAMES_END
```

Quelltext A.3: *Score-P*-Filterdatei für *NPB bt-mz*, *MPI* & *OpenMP*.



```

1 SCOREP_REGION_NAMES_BEGIN
2 EXCLUDE exact
3 SCOREP_REGION_NAMES_END

```

Quelltext A.4: *Score-P*-Filterdatei für *NPB lu-mz*, seriell.

```

1 SCOREP_REGION_NAMES_BEGIN
2 EXCLUDE exact
3         sync_right
4         sync_left
5 SCOREP_REGION_NAMES_END

```

Quelltext A.5: *Score-P*-Filterdatei für *NPB lu-mz*, *MPI*.

```

1 SCOREP_REGION_NAMES_BEGIN
2 EXCLUDE exact
3 SCOREP_REGION_NAMES_END

```

Quelltext A.6: *Score-P*-Filterdatei für *NPB lu-mz*, *MPI* & *OpenMP*.

```

1 SCOREP_REGION_NAMES_BEGIN
2 EXCLUDE exact_solution
3 SCOREP_REGION_NAMES_END

```

Quelltext A.7: *Score-P*-Filterdatei für *NPB sp-mz*, seriell.

```

1 SCOREP_REGION_NAMES_BEGIN
2 EXCLUDE exact_solution
3 SCOREP_REGION_NAMES_END

```

Quelltext A.8: *Score-P*-Filterdatei für *NPB sp-mz*, *MPI*.

```

1 SCOREP_REGION_NAMES_BEGIN
2 EXCLUDE exact_solution
3 SCOREP_REGION_NAMES_END

```

Quelltext A.9: *Score-P*-Filterdatei für *NPB sp-mz*, *MPI* & *OpenMP*.

```

1 SCOREP_REGION_NAMES_BEGIN
2 EXCLUDE Real_t CalcElemVolume(const Real_t*, const Real_t*, const Real_t*)
3 SCOREP_REGION_NAMES_END

```

Quelltext A.10: *Score-P*-Filterdatei für *LULESH*, seriell & *OpenMP*.

## Literaturverzeichnis

- [Adv15] ADVANCED MICRO DEVICES (AMD). *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. 2015
- [BBB<sup>+</sup>91] BAILEY, D. H. ; BARSZCZ, E. ; BARTON, J. T. ; BROWNING, D. S. ; CARTER, R. L. ; DAGUM, L. ; FATOOHI, R. A. ; FREDERICKSON, P. O. ; LASINSKI, T. A. ; SCHREIBER, R. S. ; SIMON, H. D. ; VENKATAKRISHNAN, V. ; WEERATUNGA, S. K.: The NAS Parallel Benchmarks. In: *The International Journal of Supercomputer Applications* 5 (1991), Nr. 3, S. 63–73
- [DM98] DAGUM, L. ; MENON, R.: OpenMP: an industry standard API for shared-memory programming. In: *IEEE Computational Science and Engineering* 5 (1998), Nr. 1, S. 46–55
- [Fog16] FOG, A.: Instruction Tables – Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs / Technical University of Denmark. 2016. – Forschungsbericht
- [GFB<sup>+</sup>04] GABRIEL, E. ; FAGG, G. E. ; BOSILCA, G. ; ANGSKUN, T. ; DONGARRA, J. J. ; SQUYRES, J. M. ; SAHAY, V. ; KAMBADUR, P. ; BARRETT, B. ; LUMSDAINE, A. ; CASTAIN, R. H. ; DANIEL, D. J. ; GRAHAM, R. L. ; WOODALL, T. S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer, 2004, S. 97–104
- [GWW<sup>+</sup>10] GEIMER, M. ; WOLF, F. ; WYLIE, B. J. N. ; ÁBRAHÁM, E. ; BECKER, D. ; MOHR, B.: The SCALASCA performance toolset architecture. In: *Concurrency and Computation: Practice and Experience* 22 (2010), Nr. 6, S. 702–719
- [HJC07] HERNANDEZ, O. R. ; JIN, H. ; CHAPMAN, B. M.: Compiler Support for Efficient Instrumentation. In: *Parallel Computing: Architectures, Algorithms and Applications*, IOS Press, 2007, S. 661–668
- [HJM<sup>+</sup>13] HUBIČKA, J. ; JAEGER, A. ; MATZ, M. ; MITCHELL, M. ; GIRKAR, M. ; LU, H. ; KREITZER, D. ; ZAKHARIN, V. *System V Application Binary Interface – AMD64 Architecture Processor Supplement*. 2013
- [HP12] HENNESSY, J. L. ; PATTERSON, D. A.: *Computer Architecture: A Quantitative Approach*. 5. Elsevier, 2012
- [Int16a] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 2: Instruction Set Reference, A–Z*. 2016

- [Int16b] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 3B: System Programming Guide, Part 2*. 2016
- [KBD<sup>+</sup>08] KNÜPFER, A. ; BRUNST, H. ; DOLESCHAL, J. ; JURENZ, M. ; LIEBER, M. ; MICKLER, H. ; MÜLLER, M. S. ; NAGEL, W. E.: The Vampir Performance Analysis Tool-Set. In: *Tools for High Performance Computing 2008: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, Springer, 2008, S. 139–155
- [KKN13] KARLIN, I. ; KEASLER, J. ; NEELY, R.: LULESH 2.0 Updates and Changes. 2013. – Forschungsbericht
- [KRM<sup>+</sup>12] KNÜPFER, A. ; RÖSSEL, C. ; AN MEY, D. ; BIERSDORF, S. ; DIETHELM, K. ; ESCHWEILER, D. ; GEIMER, M. ; GERNDT, M. ; LORENZ, D. ; MALONY, A. ; NAGEL, W. E. ; OLEYNIK, Y. ; PHILIPPEN, P. ; SAVIANKOU, P. ; SCHMIDL, D. ; SHENDE, S. ; TSCHÜTER, R. ; WAGNER, M. ; WESARG, B. ; WOLF, F.: Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In: *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing*, Springer, 2012, S. 79–91
- [LCM<sup>+</sup>05] LUK, C.-K. ; COHN, R. ; MUTH, R. ; PATIL, H. ; KLAUSER, A. ; LOWNEY, G. ; WALLACE, S. ; REDDI, V. J. ; HAZELWOOD, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: *SIGPLAN conference on Programming language design and implementation*, ACM, 2005, S. 190–200
- [LHC<sup>+</sup>07] LIAO, C. ; HERNANDEZ, O. ; CHAPMAN, B. ; CHEN, W. ; ZHENG, W.: OpenUH: An Optimizing, Portable OpenMP Compiler. In: *Concurrency and Computation: Practice and Experience* 19 (2007), Nr. 18, S. 2317–2332
- [MCC<sup>+</sup>95] MILLER, B. P. ; CALLAGHAN, M. D. ; CARGILLE, J. M. ; HOLLINGSWORTH, J. K. ; IRVIN, R. B. ; KARAVANIC, K. L. ; KUNCHITHAPADAM, K. ; NEWHALL, T.: The Paradyn Parallel Performance Measurement Tool. In: *Computer* 28 (1995), Nr. 11, S. 37–46
- [MCSM06] MOORE, S. ; CRONK, D. ; SHENDE, S. ; MALONY, A.: Loop-Level Profiling and Analysis of DoD Applications Using TAU. In: *HPCMP User Group Conference*, IEEE, 2006, S. 378–383
- [Mel10] DE MELO, A. C.: The new linux ‘perf’ tools. In: *Slides from Linux Kongress* Bd. 18, 2010
- [Mes94] MESSAGE PASSING INTERFACE FORUM. *MPI: A Message-Passing Interface Standard*. 1994
- [MLW11] MUSSLER, J. ; LORENZ, D. ; WOLF, F.: Reducing the overhead of direct application instrumentation using prior static analysis. In: *Euro-Par 2011 Parallel Processing: 17th International Conference, Part I*, Springer, 2011, S. 65–76

- [MRW92] MALONY, A. ; REED, D. A. ; WIJSHOFF, H. A. G.: Performance Measurement Intrusion and Perturbation Analysis. In: *IEEE Transactions on Parallel and Distributed Systems* 3 (1992), Nr. 4, S. 433–450
- [SM06] SHENDE, S. ; MALONY, A.: The TAU parallel performance system. In: *International Journal of High Performance Computing Applications* 20 (2006), Nr. 2, S. 287–311
- [Sto15] STOLLE, J.: *Adaptive Runtime Filtering supporting an Event-Based Performance Analysis*, TU Dresden, Diplomarbeit, 2015
- [TB15] TANENBAUM, A. S. ; BOS, H.: *Modern Operating Systems*. 4. Pearson, 2015
- [WDKN14] WAGNER, M. ; DOLESCHAL, J. ; KNÜPFER, A. ; NAGEL, W. E.: Selective runtime monitoring: Non-intrusive elimination of high-frequency functions. In: *International Conference on High Performance Computing & Simulation (HPCS)*, IEEE, 2014, S. 295–302
- [Zen13] ZENTRUM FÜR INFORMATIONSDIENSTE UND HOCHLEISTUNGSRECHNEN (ZIH). *VampirTrace 5.14.4 – User Manual*. 2013

## Abbildungsverzeichnis

3.1	Schematische Darstellung der Abläufe in einer instrumentierten Funktion . . . .	9
3.2	Schematische Darstellung eines Kellerspeichers . . . . .	10
3.3	Beispielhafter Aufbau eines von der Ausgabe von <i>libunwind</i> implizierten Stacks .	12
3.4	Ausschnitt aus dem Textsegment eines Programms vor dem Entfernen . . . . .	14
3.5	Ausschnitt aus dem Textsegment eines Programms nach dem Entfernen . . . . .	15
3.6	Schematische Darstellung der Datensynchronisierung innerhalb des Plugins . . .	18
4.1	Ergebnisse für <i>NPB bt-mz</i> . . . . .	21
4.2	Ergebnisse für <i>NPB sp-mz</i> . . . . .	22
4.3	Ergebnisse für <i>NPB lu-mz</i> . . . . .	24
4.4	Ergebnisse für <i>LULESH</i> . . . . .	25
4.5	Ergebnisse der Vergleichsmessungen mit den <i>Intel</i> -Compilern ( <i>NPB</i> ) . . . . .	26
4.6	Vergleich der in <i>Vampir</i> dargestellten Funktionen und ihrer Aufrufdauer . . . . .	28
4.7	Vergleich der in <i>Vampir</i> dargestellten, zeitlichen Abläufe . . . . .	29
5.1	Verdeutlichung des Problems der variablen Sprungbefehlslänge . . . . .	32

## Tabellenverzeichnis

2.1	Übersicht über verschiedene Ansätze zur Reduktion des Instrumentierungs-Over-heads . . . . .	5
5.1	Anteil der Zugriffe auf den <i>last level cache</i> für <i>NPB bt-mz A</i> . . . . .	34

## Algorithmenverzeichnis

3.1	Bestimmung der verwendeten Instrumentierungsaufrufe . . . . .	11
3.2	Finden eines Instrumentierungsaufrufs . . . . .	13

## Quelltextverzeichnis

3.1	Ausschnitt aus einem beispielhaften Programm . . . . .	16
5.1	Ausschnitt aus einem vom <i>Intel</i> -Compiler übersetzten C++-Programm . . . . .	33
A.1	<i>Score-P</i> -Filterdatei für <i>NPB bt-mz</i> , seriell . . . . .	36
A.2	<i>Score-P</i> -Filterdatei für <i>NPB bt-mz</i> , <i>MPI</i> . . . . .	36
A.3	<i>Score-P</i> -Filterdatei für <i>NPB bt-mz</i> , <i>MPI</i> & <i>OpenMP</i> . . . . .	36
A.4	<i>Score-P</i> -Filterdatei für <i>NPB lu-mz</i> , seriell . . . . .	37
A.5	<i>Score-P</i> -Filterdatei für <i>NPB lu-mz</i> , <i>MPI</i> . . . . .	37
A.6	<i>Score-P</i> -Filterdatei für <i>NPB lu-mz</i> , <i>MPI</i> & <i>OpenMP</i> . . . . .	37
A.7	<i>Score-P</i> -Filterdatei für <i>NPB sp-mz</i> , seriell . . . . .	37
A.8	<i>Score-P</i> -Filterdatei für <i>NPB sp-mz</i> , <i>MPI</i> . . . . .	37
A.9	<i>Score-P</i> -Filterdatei für <i>NPB sp-mz</i> , <i>MPI</i> & <i>OpenMP</i> . . . . .	37
A.10	<i>Score-P</i> -Filterdatei für <i>LULESH</i> , seriell & <i>OpenMP</i> . . . . .	37



## Danksagung

Mein Dank gilt Kaja, dafür, dass sie mir in den letzten Wochen den Rücken frei gehalten hat, meinen Betreuern Joseph und Robert, für ihre guten Ratschläge und ihre Geduld, und natürlich Sebastian, Christian, Michael, Daniel und Andreas, die immer wieder hilfreiche Tipps und Tricks insbesondere für den Umgang mit *Score-P* für mich parat hatten.

...und Denis, für seine erfrischend positive Weltsicht.