Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Besser Benchmarken

Bachelorarbeit von

Johannes Bechberger

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr.-Ing. Jörg Henkel
Betreuende Mitarbeiter:	Andreas Zwinkau

Bearbeitungszeit: 3. Februar 2016 – April 8, 2016

Zusammenfassung

Richtiges Benchmarking ist aufwendig und fehleranfällig. Diese Arbeit widmet sich deshalb der Entwicklung eines Werkzeugs namens *temci*, welches es ermöglicht, mit weniger Fehlern und Aufwand zu benchmarken. Das Werkzeug hilft außerdem dabei die Ergebnisse korrekt darzustellen und mehrere Programme miteinander zu vergleichen. Durch seine modulare Struktur ist es möglich andere Benchmarkingwerkzeuge in wenigen Zeilen Code einzubinden und weitere Funktionalitäten hinzuzufügen.

Einige Dinge die *temci* auszeichnen:

- Programm-Randomisierung zur Verminderung von Cacheeffekten
- Reduzierung der Messwertestreuung durch Separation
- Vergleichen verschiedener Programmrevisionen
- Unterstützung vieler Zeitmesswerkzeuge, wie zum Beispiel `perf stat`
- Aussagekräftige Ergebnisaufbereitung
- Erzeugung von Benchmarking-Paketen zur besseren Reproduzierbarkeit

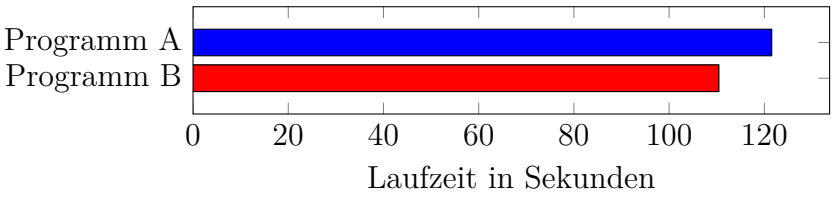
Inhaltsverzeichnis

1. Einleitung

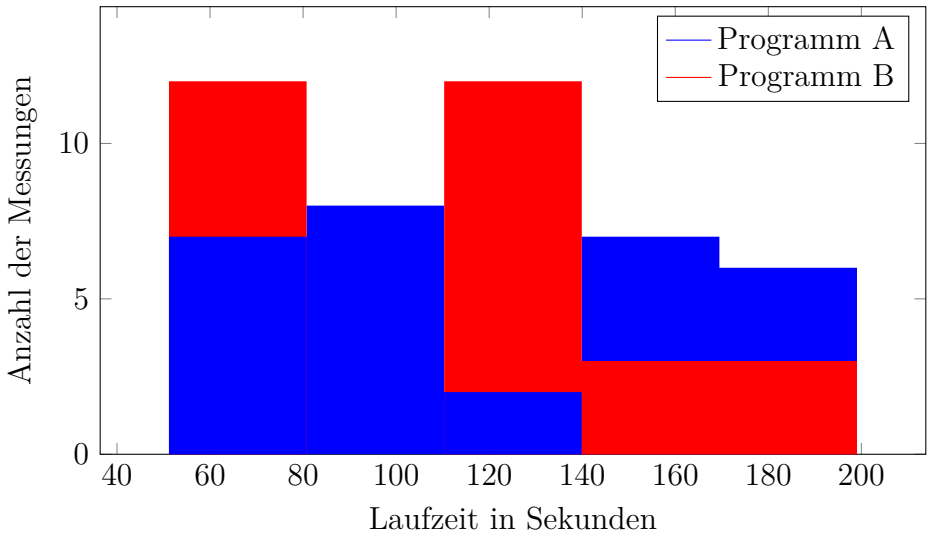
1.1. Motivation

Die korrekte Messung der Laufzeit von Programmen, Benchmarking genannt, ist schwer. Jeder Teil des Benchmarkens (das Aufsetzen des Systems, das Vornehmen von Messungen, die Interpretierung und Darstellung der Ergebnisse) hat seine eigenen Tücken. Begreifbar sind die vielen Probleme bei der korrekten Darstellung und Interpretierung der Ergebnisse. Im Folgenden ein zwar synthetisches aber doch realitätsnahes Beispiel:

Der betrachtete Benchmark ist ein Benchmark zweier Programme A und B. Beide Programme wurden 30-mal ausgeführt und dabei die Ausführungszeiten gemessen. Die Ausführungszeit für das Programm A war im Mittel 122 Sekunden und für das Programm B 110 Sekunden. Die Laufzeiten können, und werden oft als Diagramm dargestellt:



Die mittlere Laufzeit von B ist rund 10% kleiner als jene von A. Es sieht so aus, als wäre A deutlich langsamer als B. In einem normalen Text zum Benchmark würde jetzt oft eine Aufzählung von Gründen kommen, warum B so gut im Vergleich zu A ist. Stattdessen folgt nun die Verteilung der Laufzeiten beider Programme in einem Histogramm:



Man sieht, dass die Situation nicht so klar ist, wie es die eigentliche Beschreibung des Benchmarks glauben lassen wollte. Die Laufzeiten von A und B sind über einen großen Bereich verteilt. Konkret sind die minimalen und maximalen Laufzeitwerte für beide Programme

Programm	Minimum	Maximum	Mittelwert	Median	Standardabweichung
A	54	200	122	110	37
B	51	195	110	115	39

Neben dem Laufzeitenbereich ist auch der Median interessant, der den Mittelwert der 15. und 16. größten Messung (bei 30 Messungen insgesamt) angibt. Angenommen, man würde nur den jeweiligen Median der Programmlaufzeiten betrachten, dann würde man zum Schluss kommen, das A 5% schneller als B ist. Der Grund hierfür ist die hohe Streuung der Laufzeiten (siehe letzte Tabellenspalte), die eine Aussage vom Stile „X ist schneller als Y“ unmöglich macht.

Nun liegt es nahe zu überprüfen, ob beide Programme die gleiche Laufzeitenverteilung haben könnten. Wie hoch also die Wahrscheinlichkeit ist, dass der Mittelwertunterschied nur daher kommt, dass zu wenige Messungen vorgenommen wurden. Dafür kann man einen sogenannten statistischen Test verwenden. Einer der gebräuchlichsten Tests ist der t-Test, dieser gibt die Wahrscheinlichkeit, dass die Laufzeiten von A und B nicht verschiedenen sind mit 32% an. Damit ist die Aussage von weiter oben, dass B deutlich schneller als A ist nicht mehr haltbar, da die Wahrscheinlichkeit über der üblichen Signifikanzgrenze von 5% liegt (vgl. Abschnitt 2.2).

Selbst wenn das Nichtpublizieren von Bestandteilen des Benchmarkergebnisses gut

begründet sein mag, ist eine Angabe sinnvoll, wenn pro Benchmark mehrere Messungen vorgenommen wurden. Neben der Messwertestreuung gehört die Anzahl der Einzelmessungen zu den wichtigen Bestandteilen. Mit beiden Bestandteilen zusammen ist es möglich die Güte des Benchmarks einzuschätzen. Die wichtigen statistischen Grundlagen werden samt Einordnung in den Benchmarkingkontext im Kapitel 2 erläutert.

Leider wird die gerade beschriebene Unachtsamkeit, dass nur Mittelwerte[1] betrachtet und publiziert werden, recht häufig begangen. Im Kapitel B wurden 144 Publikationen von 4 Lehrstühlen der Fakultät für Informatik am KIT mit Laufzeitangaben untersucht, 90% dieser Publikationen begehen die beschriebene Unachtsamkeit. Aber wichtig: Es heißt nicht, dass diese Veröffentlichungen inhaltlich falsch sind, denn oft sind die zu erwarteten Messwertestreuungen kleiner als die in der jeweiligen Veröffentlichungen erläuterten Effekte. Die im Folgenden gezeigten Beispiele sollen die Unachtsamkeit exemplarisch zeigen.

Das folgende Beispiel 1.1 stammt aus einem Blogartikel von Drew Crawford mit dem Titel „Why mobile apps are slow“ Crawford (2013).

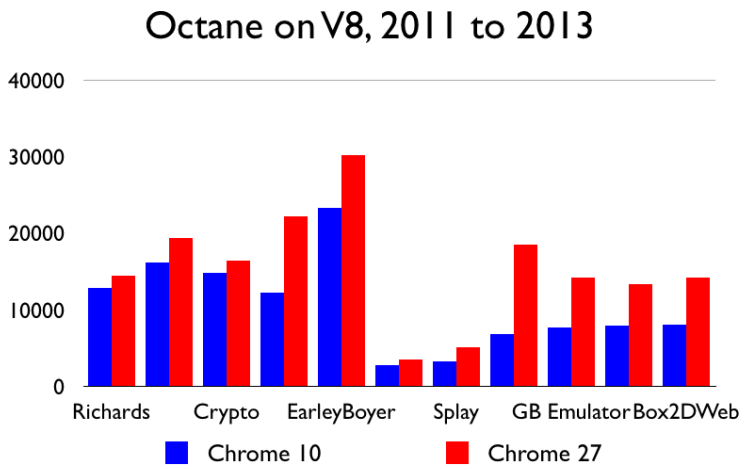


Abbildung 1.1.: Beispieldiagramm aus Crawford (2013)

> In my opinion, this magnitude of performance gain over this period is much too small to support the claim that JS will close the gap in any reasonable amount of time. However, I think it's fair to say that I overstated the case a bit – something is happening in CPU-bound JavaScript.

[1] Oder andere vereinfachende statistische Kennzahlen, wie das Maximum oder Minimum.

Ein weiteres Beispiel ist aus Buchwald u. a. (2011) entnommen: Die Tabelle 1.1 vergleicht zwei im CParser implementierte Registerallokationsalgorithmen anhand des SPEC CPU2000 Integer Benchmarks.

Benchmark	Recoloring	PBQP	Ratio
164.gzip	345	350	101.4%
175.vpr	446	444	99.7%
176.gcc	179	179	99.8%
181.mcf	336	335	99.6%
186.crafty	233	231	99.4%
197.parser	468	467	99.7%
253.perlbmk	355	354	99.8%
254.gap	252	253	100.4%
255.vortex	417	418	100.1%
256.bzip2	374	371	99.4%
300.twolf	684	680	99.4%
Average			99.9%

Tabelle 1.1.: Comparison of execution time in seconds with recoloring and PBQP. Buchwald u. a. (2011)

In dieser Tabelle (und auch im dazugehörigen Text) wurden keinerlei Angaben zur Streuung der Messwerte gemacht. Einer der Autoren ist sich selbst bewusst, dass diese Angaben fehlen. Seine Begründungen und eine ausführliche Reevaluierung finden sich im Abschnitt 6.4.

Aber selbst wenn man diese Unachtsamkeiten vermeidet, hat man immer noch eine wichtige Fehlerquelle übersehen: Verzerrungen der Messungen die durch die Systemumgebung herrühren, in welcher die Benchmarks gemacht werden, zum Beispiel durch verschieden große Umgebungsvariablen Diwan u. a. (2009).

Fehler wie die hier besprochenen können jedem unterlaufen (und auch jedem der eine Publikation nur kurz quer liest). Die gewählten Beispiele sind zwei von vielen. Im Rahmen dieser Bachelorarbeit wurde nun das Benchmarkingwerkzeug *temci* entwickelt, dass dieser Unachtsamkeiten und Fehler zu vermeiden hilft und, zusammen mit dieser Arbeit, auf diese aufmerksam machen soll.

Um wieder zurück zum synthetischen Einführungsbeispiel zu kommen: In der von *temci* erzeugten Ergebnisdarstellung in Abbildung 1.2 wird auf den ersten Blick deutlich, dass die Daten wohl eher unbrauchbar sind und keine Schlussfolgerung vom Stile „X ist schneller als Y“ möglich sein werden:

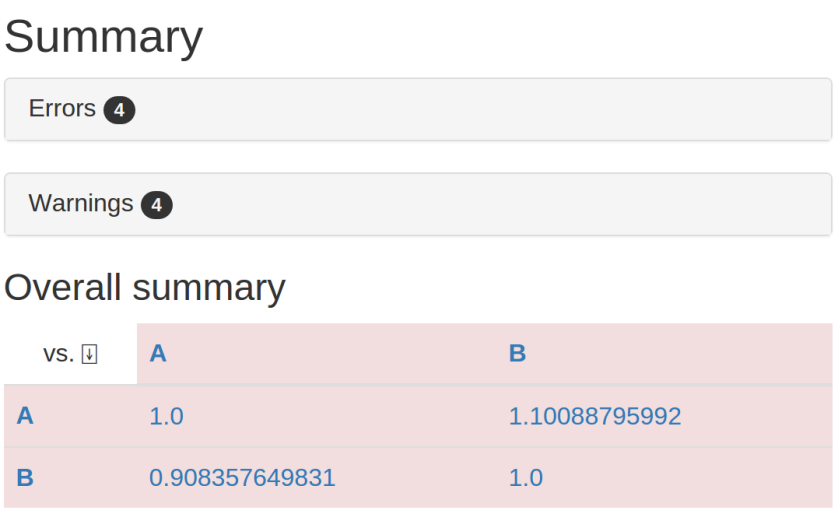


Abbildung 1.2.: Screenshot des von temci 0.5.5, für das synthetische Einführungsbeispiel, generierten Reports. Die Tabelleneinträge geben die relative Performanz des Programms in der Zeile gegenüber dem Programm in der Spalte an. Kleinere Werte sind besser und je weiter sie von der 1 entfernt sind, desto größer ist der jeweilige Performanzunterschied. Die Standardabweichung wird beim Klicken auf den jeweiligen Tabelleneintrag angezeigt.

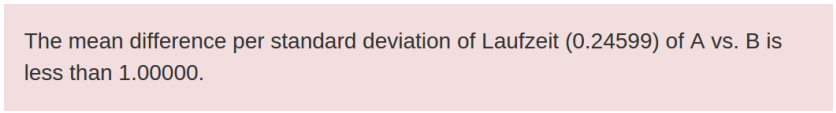


Abbildung 1.3.: Screenshot eines Errors aus dem in Abbildung 1.2 gezeigten Report

Zu jedem der von *temci* erkannten möglichen Fehler (unter „Errors“ und „Warnings“) wird neben der Klarstellung des Fehlers auch ein Hinweis zur Behebung angegeben. Ein Beispiel ist in Abbildung 1.3 gegeben.

Neben dem Aufzeigen von möglichen Fehlern bietet *temci* auch die Möglichkeit gute Benchmarks mit relativ geringem Aufwand anzufertigen und dabei auch während dem Aufsetzen des Systems und dem eigentlichen Messen Fehler zu vermeiden. Die hierfür implementierten Mechanismen werden in den Kapiteln 4 und 5 erläutert. *temci* selbst ist ein Werkzeug, welches auch als Basis für weitere Anwendungen dienen kann. Zum Beispiel kann *temci* benutzt werden, um das bekannte *Computer Language Benchmarks Game* Fulgham u. Gouy zu reimplementieren (siehe Abschnitt 6.2).

1.2. Fokus von *temci*

Der Fokus liegt auf dem **Benchmarken von ganzen Kommandozeilenbefehlen**. Im speziellen geht es darum Messungen anzufertigen, welche reproduzierbar sind. Hinsichtlich von Cacheeffekten ist Letzteres nicht trivial. Mechanismen hierfür werden im Kapitel **Normalisierung durch Randomisierung** erläutert. Gute Benchmarks zeichnen sich des Weiteren durch eine geringe Streuung aus. Mechanismen hierfür werden im Abschnitt **Normalisierung durch Separation** ausgeführt.

Neben dem Benchmarking ging es bei der Entwicklung von *temci* auch darum, dem Anwender ein benutzerfreundliches Werkzeug an die Hand zugeben. *temci* zeichnet sich in diesem Bereich durch Folgendes aus:

Einfache Installierbarkeit Man kann *temci* via `pip3 install temci` auf Linux mit dem Python Paketmanager installieren, sofern man die Pythonpakete *scipy*, *numpy* und *matplotlib* installiert hat.

Vervollständigung für Shells Es gibt Unterstützung für die Vervollständigungen von Bash und ZSH, bei Letzterer mit umfassenden Beschreibungen.

Generierung eines ausführlichen Reports Es ist über das Teilwerkzeug *temci report* möglich, einen umfassenden Report für die gemessenen Daten zu generieren. Dieser gibt viele statistische Kennwerte zu den Daten an und visualisiert sie in verschiedenen Plots. Außerdem wird darin auf mögliche Fehler hingewiesen und wie sie zu vermeiden sind (ein Beispiel ist in Abbildung 1.2 gegeben). Dabei werden alle auftretenden mathematischen Fachbegriffe erläutert. Der Report ermöglicht es die Benchmarks besser zu verstehen und verschiedene Benchmarks miteinander zu vergleichen, zum Beispiel um statistisch korrekte Aussagen über Performanzverbesserungen zu treffen.

1.3. Benutzung von *temci*

Im Folgenden wird die Benutzung von *temci* kurz erklärt, indem ein einfaches Programm (`ls`) gebenchmarkt wird. Es wird davon ausgegangen das *temci* über den Pythonpaketmanager (wie weiter oben) erläutert bereits installiert wurde.

1.3.1. Benchmarken

Um `ls` zu benchmarken verwendeten man am einfachsten *temci short exec* wie folgt:

```
temci short exec -wd "ls"
```

Erläuterungen

`short` Die Kategorie von Subprogrammen, welche es ermöglichen Programme auch ohne Verwendung von Konfigurationsdateien zu benchmarken.

`exec` Der verwendete Benchmarkingtreiber, hier derjenige, der für Benchmarks verwendet wird, bei welchen ein Kommando auf der Kommandozeile ausgeführt wird. Er ist für das eigentliche Benchmarken zuständig.

`-wd` Kurz für `-without_description` gibt an, dass das Programm `ls` als Beschreibung im später erzeugten Report nur den eigenen Namen hat.

Oft werden zusätzlich noch die folgenden Parameter übergeben:

`--runs 100` Die Anzahl der Male, die `ls` ausgeführt und gebenchmarkt werden soll (ohne diese Angabe gilt hier `--runs 20`)

`--out out.yaml` Der Dateiname unter dem die Benchmarkergebnisse als YAML gespeichert werden (ohne diese Angabe gilt `-out run_output.yaml`).

Um zwei Programme miteinander zu vergleichen, wird ein weiteres `-wd PROGRAMM` angehängt. Um `ls` konkret mit `ls ..` zu vergleichen, verwendet man *temci short exec* wie folgt:

```
temci short exec -wd "ls" -wd "ls .."
```

1.3.2. Reporterzeugung

Um aus der Benchmarkergebnisdatei einen ausführlichen HTML Report zu generieren, verwendet man *temci report* folgendermaßen:

```
temci report run_output.yaml
```

Erläuterungen

`out.yaml` Die Datei in welcher die Benchmarkergebnisse gespeichert werde.

Implizit wurden hierbei unter anderem die folgenden Parameter angegeben.

`--reporter html2` Gibt an, dass der HTML2 Reporter[2] verwendet werden soll, welcher einen ausführlichen HTML Report generiert. Dieser Reporter wird auch im Standardfall verwendet. Daneben gibt es noch einen Reporter für die Ausgabe auf der Kommandozeile.

`--html2_out report` Gibt an, in welches Verzeichnis der Report gespeichert werden soll. Im Standardfall wird der Report im Verzeichnis `report` generiert. Den Report kann man sich nun ansehen, indem man die im Verzeichnis liegende Datei `report.html` betrachtet.

1.3.3. Paketierung

Um den ausgeführten Benchmark besser reproduzierbar zu machen, kann mithilfe von *temci* ein Paket erzeugt werden, welches alles notwendige enthält, um den Benchmark zu reproduzieren. Dies mag hier keinen Sinn haben, doch bei größeren Benchmarks mit vielen Abhängigkeiten erleichtert dies die Reproduzierung ungemein.

Um ein solches *temci*-Paket zu erzeugen, wird der folgende Python Code ausgeführt:

```
from temci.package.dsl import actions, store, IncludeTree, ExecuteCmd

actions << IncludeTree(".") \
        << ExecuteCmd("temci short exec -wd 'ls'") \
        << ExecuteCmd("temci report run_output.yaml")
store("package.temci")
```

Erläuterungen

`import ...` Damit werden die benötigten Bestandteile von *temci* eingebunden.

`<< IncludeTree(".")` Fügt den aktuellen Verzeichnisbaum zum Paket hinzu, denn auf diesen Verzeichnisbaum wird von `ls` beim Benchmarken zugegriffen.

[2] Der HTML2 Reporter ist die Weiterentwicklung des eher rudimentären HTML Reporters.

`<< ExecuteCmd("temci short exec -wd 'ls'")` Führt das Kommando `temci short exec -wd 'ls'` zum Benchmarken aus, nach dem das System eingerichtet wurde (der aktuelle Dateibaum plaziert und `ls` installiert ist).

`<< ExecuteCmd("temci report run_output.yaml")` Führt danach das Kommando `temci report run_output.yaml` zur Generierung eines Reports aus.

Das erzeugte *temci*-Paket `package.temci` kann nun den Benchmarkergebnissen beigelegt werden. Es kann mithilfe des folgenden Kommandos ausgeführt werden:

```
temci run_package package.temci
```

Es handelt sich bei den temci Paketen um eine noch spärlich getestete Funktionalität von temci, die aber in Zukunft einer der Kernbestandteile von temci werden soll. Im weiteren Verlauf dieser Arbeit wird nicht weiter auf temci-Pakete eingegangen.

1.4. Literatur

Diese Arbeit baut auf mehreren Arbeiten auf und versucht diese in einem Werkzeug benutzbar zusammen zu fassen.

Ein für diese Arbeit sehr wichtiges Dokument, ist die Liste von „System Benchmarking Crimes“ von Gernot Heiser Heiser. In ihr, wie auch in der zugehörigen Vorlesung zum selben Thema, erläutert er viele Fehler und Unachtsamkeiten, die man beim Benchmarken machen kann und gibt Verbesserungsvorschläge. Diese Liste bildet neben eigenen statistischen Überlegungen die Grundlage des nächsten Kapitels Statistische Grundlagen und der von *temci* erzeugten Reports.

In Diwan u. a. (2009) wird das Konzept der bereits erwähnten Messverzerrungen erläutert, die zum einen von verschieden großen Umgebungsvariablen und zum anderen von der Ausrichtung von Daten im Speicher herrühren. Im Kapitel 4 werden hierfür Gegenmaßnahmen erläutert. Sie lehnen sich an die in Curtsinger u. Berger (2013) beschriebenen Maßnahmen an. Es wurde im Vergleich zu dieser Publikation jedoch ein einfacherer und für den Benutzer transparenter Weg gewählt.

1.5. Der Name *temci*

Zum Ende dieses Kapitels sei noch erwähnt, warum *temci* “temci“ heißt. Bei der Namensfindung für ein Projekt im Bereich Benchmarking steht man vor dem Problem, dass die meisten kurzen und mit der Thematik des Benchmarkens verknüpften Namen bereits vergeben sind. Einen bereits verwendeten Namen nochmals zu verwenden kam aus Gründen der Auffindbarkeit und Installierbarkeit mit einem Paketmanager nicht in Frage. Um dennoch einen passenden und möglichst kurzen Namen zu finden, wurde vom Englischen in Lojban gewechselt. Lojban ist eine künstliche Sprache. Der gewählte Name „temci“ ist in Lojban ein Verb und bedeutet laut dem offiziellen Wörterbuch LLG:

> x1 ist der Zeitraum/das Intervall/die Periode/[verstrichene Zeit] zwischen Zeitpunkten/Ereignissen x2 und x3

2. Statistische Grundlagen

In diesem Kapitel wird näher auf die statistischen Grundlagen eingegangen, mit deren Kenntnis viele Fehler (siehe Problematisierung) vermieden werden können. Außerdem wird mithilfe von Heiser zu jeder der genannten statistischen Kennwerte eine Erläuterung ihrer Wichtigkeit im Bereich des Benchmarkens gegeben. Viele dieser Erläuterungen und Erklärungen finden sich auch in von *temci* erzeugten Reports.

Im Folgenden Text wird davon ausgegangen, dass zwei Programme X und Y jeweils n_x- bzw. n_y-mal gebenchmarkt wurden und nun bzgl. ihrer Performanz verglichen werden sollen. Die konkreten Messungen seien $x = (x_0, \ldots, x_{n_x})$ und $y = (y_0, \ldots, y_{n_y})$.

2.1. Statistische Kennwerte

2.1.1. Mittel

Es gibt verschiedene Mittel, deren konkrete Werte Mittelwerte genannt werden. Das gebräuchlichste Mittel ist das *arithmetische Mittel*, es wird für x wie folgt definiert (Henze, 2013, S. 28)

$$\bar{x} = \frac{1}{n_x} \sum_{i=1}^{n_x} x_i.$$

Dieses Mittel eignet sich gut um die Werte einzelner Messungen des selben Programms zu mitteln.

Dagegen eignet es sich nicht, um normalisierte Benchmarks zusammenzufassen. Das heißt, angenommen man würde weitere Benchmarks von X und Y machen ($\mathbb{x}$ und $\mathbb{y}$) und die Mittelwerte der Benchmarks relativ zu den Benchmarks von X angeben, wäre es laut Fleming u. Wallace (1986) nicht korrekt, das arithmetische Mittel zu verwenden. Für das korrekte Zusammenfassen verschiedener normalisierter Benchmarks verwendet man das *geometrische Mittel* Fleming u. Wallace (1986). Falls $\tilde{x}, \bar{\mathbb{x}}$ und $\tilde{y}, \bar{\mathbb{y}}$ die zu $\bar{x}$ bzw. $\bar{\mathbb{x}}$ normalisierten Mittelwerte bzgl. X und Y sind, kann der zusammenfassende geometrische Mittelwert für X und Y wie folgt berechnet werden (Henze, 2013, S. 36)

$$\sqrt{\tilde{x} \cdot \tilde{\mathbb{x}}} \qquad \text{bzw.} \qquad \sqrt{\tilde{y} \cdot \tilde{\mathbb{y}}}$$

Oder allgemein für eine Liste z mit den Werten $z_1, \ldots, z_n$

$$\bar{z} = \sqrt[n]{\prod_{i=1}^{n} z_i}$$

Wichtig zu erwähnen ist noch, dass die normalisierten Werte bei der Verwendung des *geometrischen Mittels* per Definition positiv sein müssen[1]. Dies ist aber im Normalfall immer der Fall.

Ein weiterer Mittelwert ist der Median, der für die sortierten Elemente von x $(x_{(1)}, \ldots, x_{(n_x)})$ wie folgt definiert ist (Henze, 2013, S. 303)

$$\bar{x}_{\text{med}} = \begin{cases} x_{\left(\frac{n_x+1}{2}\right)} & n_x, \text{ ungerade,} \\ \frac{1}{2}\left(x_{\left(\frac{n_x}{2}\right)} + x_{\left(\frac{n_x}{2}+1\right)}\right) & n_x, \text{ gerade.} \end{cases}$$

Die Betrachtung des Medians kann bei Messungen mit vielen Ausreißern hilfreich sein, da einzelne Ausreißer wenig am Median ändern. Wobei man in diesem Fall auch die Frage stellen sollte, ob die Messungen nicht fehlerhaft waren, oder was zu so vielen Ausreißern geführt haben könnte.

2.1.2. Standardabweichung und Varianz

Die Varianz von x ist

Für den arithmetischen Mittelwert

$$\sigma^2_{\bar{x}_{\text{arith}}} = \frac{1}{n_x} \sum_{i=1}^{n_x} (x_i - \bar{x}_{\text{arith}})^2$$

und für den geometrischen Mittelwert

$$\sigma^2_{\bar{x}_{\text{geom}}} = \exp\left(\sqrt{\frac{1}{n_x} \cdot \sum_{i=1}^{n_x} \left[\ln\left(\frac{x_i}{\bar{x}_{\text{geom}}}\right)\right]^2}\right)^2$$

Die Standardabweichung ist definiert als die Wurzel der Varianz, und damit (siehe Kapitel A)

$$\sigma_{\bar{x}_{\text{arith}}} = \sqrt{\frac{1}{n_x} \sum_{i=1}^{n_x} (x_i - \bar{x}_{\text{arith}})^2} \qquad \sigma_{\bar{x}_{\text{geom}}} = \exp\left(\sqrt{\frac{1}{n_x} \cdot \sum_{i=1}^{n_x} \left[\ln\left(\frac{x_i}{\bar{x}_{\text{geom}}}\right)\right]^2}\right)$$

[1] Würden negative Werte zugelassen, würde ein neu zu z hinzukommender negativer Wert z_{n+1} den Mittelwert von z stark verändern, selbst wenn z sehr groß und der Betrag von z_{n+1} relativ nahe beim vorherigen Mittelwert von z liegt. Dies würde jede weitere Schlussfolgerung aus dem Mittel erschweren. Das gleiche gilt für den Fall, dass der Wert 0 zugelassen wird.

Beide Werte sind ein Maß für die Streuung einer Verteilung. Liegen ihre jeweiligen Werte bei 0 (bei der Arithmetischen) bzw. bei 1 (bei der geometrischen Standardabweichung) sind alle x_i identisch. Je größer ihre jeweiligen Werte werden, desto größer ist der Bereich über den sich die x_i verteilen. Die Werte der geometrischen Standardabweichung um 1 verringert sind mit jenen der (arithmetischen) Standardabweichung vergleichbar. Sofern im folgenden nicht explizit, sondern nur implizit, von der geometrischen Standardabweichung die Rede ist, wird diese als um 1 reduziert angegeben.

Die Standardabweichung wird im Normalfall der Varianz vorgezogen, da ihre Einheit dieselbe, wie die der x_i ist und sie damit besser begreifbar ist. Oft wird sie relativ zum Mittelwert der jeweiligen Verteilung angegeben[2].

Die Standardabweichung sollte bei Messungen möglichst gering sein. Benchmarks mit hohen Standardabweichungen sind unbrauchbar. Gernot Heiser schreibt dazu Heiser:

> Always do several runs, and check the standard deviation. Watch out for abnormal variance.

Außerdem können die Standardabweichungen der Benchmarks zweier Programme einen guten Anhaltspunkt geben, ob eine Differenz der Mittelwerte zwischen beiden Programmen signifikant ist. Dazu Gernot Heiser Heiser:

- Don't believe any effect that is less than a standard deviation
- Be highly suspicious if it is less than two standard deviations

Eine Motivation dieser Faustregel auf Grundlage von statistischen Betrachtungen wird im Abschnitt A.2 gegeben.

Im von *temci* generierten, Report gibt es im zweiten Fall eine Warnung und im ersten Fall zusätzlich einen Error, sodass direkt klar ist, wenn die Differenz relativ zum Maximum der beiden Standardabweichungen (beider Benchmarks) zu klein ist.

2.1.3. Quartile und Boxplots

Neben der Standardabweichung gibt es noch eine Art statistischer Kennzahlen, welche Aufschluss über die Streuung der Messungen geben. Es sind die sogenannten **Quartile** (Henze, 2013, S. 303). Sie unterteilen die Messungen in 4 gleich große

[2]Diese Kennziffer ist auch als Variationskoeffizient bekannt.

Mengen. Es gibt 3 Quartile, die für die sortierten Elemente von x $(x_{(1)}, \ldots, x_{(n_x)})$ wie folgt berechnet werden

$$\tilde{x}_i = \begin{cases} x_{\left(i \cdot \frac{n_x+1}{4}\right)}, & n_x \text{ ungerade}, \\ \frac{1}{2}\left(x_{\left(i \cdot \frac{n_x}{4}\right)} + x_{\left(i \cdot \frac{n_x}{4}+1\right)}\right), & n_x \text{ gerade}. \end{cases} \qquad i = 1, 2, 3$$

Die 3 Quartile werden auch 25%, 50% und 75% Dezil (Henze, 2013, S. 304) genannt, das zweite Quartil ist außerdem als Median bekannt. Wichtig für Varianzbetrachtungen ist der **Quartilsabstand** (QR) (Henze, 2013, S. 304)

$$\mathrm{QR} = \tilde{x}_3 - \tilde{x}_1$$

Dieser ist vergleichbar mit der Standardabweichung, wobei er robuster gegenüber Ausreißern ist. Der QR kann verwendet werden, um grob zu sehen, welche Messungen Ausreißer sind (Henze, 2013, S. 34). Hierzu wird ein Wertebereich definiert, außerhalb dessen die Messungen eher Ausreißer sind. Der Bereich wird durch zwei Grenzen beschrieben, welche folgende sind

$$\begin{aligned} \text{Untere Grenze} &= \min\{\ e \mid e \in x \ \wedge\ e < \tilde{x}_1 - 1.5 \cdot \mathrm{QR}\ \} \\ \text{Obere Grenze} &= \max\{\ e \mid e \in x \ \wedge\ e > \tilde{x}_3 + 1.5 \cdot \mathrm{QR}\ \} \end{aligned}$$

Mit den 3 Quartilen und den beiden Grenzen kann eine Verteilung auch visualisiert werden. Solch eine Visualisierung wird auch als **Boxplot** (Henze, 2013, S. 34) bezeichnet und wird gerne für diesen Zweck verwendet (zum Beispiel beim *Computer Language Benchmarks Game* Fulgham u. Gouy). Sein schematischer Aufbau ist in Abbildung 2.1 gegeben. Außerdem wird in Abbildung 2.2 ein Boxplot für das synthetische Einführungsbeispiel gezeigt.

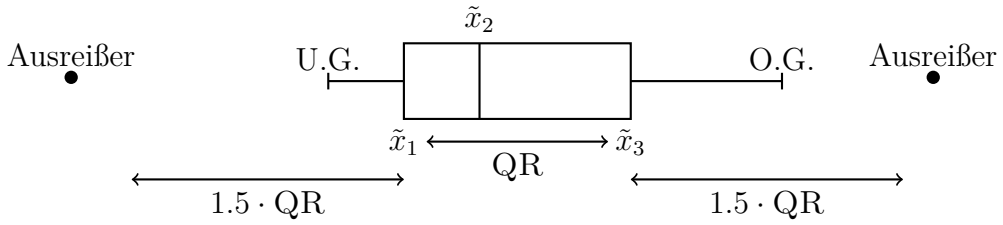


Abbildung 2.1.: Schematischer Aufbau eines Boxplots

Das 1. und das 3. Quartil spannen jeweils den Nichtausreißerbereich auf und sind der Grund für die Bezeichnung als Boxplot. Die einzelnen Punkte stellen die Ausreißer dar.

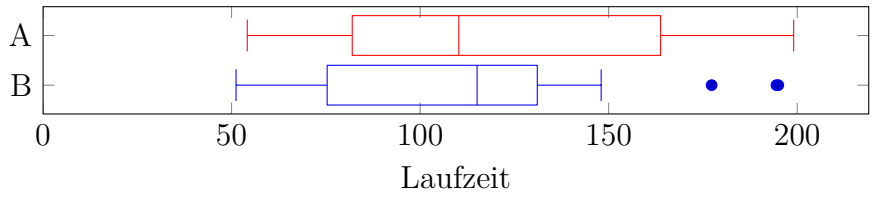


Abbildung 2.2.: Boxplot für das synthetische Einführungsbeispiel

2.1.4. Anzahl

Die Anzahl der Messungen ist wichtig. Sofern die Anzahl der Messungen zu klein ist, kann jede weitere Messung das Benchmarksergebnis ändern, was besonders bei Messungen mit hoher Streuung relevant ist. Mit dem Gesetz der Großen Zahlen (Henze, 2013, S. 217) folgt, dass das Ergebnis sich mit der größer werdenden Anzahl an Messungen stabilisiert.

Im von *temci* generierten Report gibt es eine Warnung, falls die Anzahl der Messung kleiner als 30 ist, und einen Error, falls die Anzahl kleiner als 15 ist. Der untere Grenzwert ist rund 1.5-mal so groß wie die durchschnittliche Anzahl von Messungen Diwan u. a. (2009) und soll in Kombination mit dem großen oberen Wert dazu anhalten, mehr Messungen durchzuführen. Denn je mehr Messungen durchgeführt werden, desto eindeutiger wird das Ergebnis. Die Grenzwerte selbst können in *temci* konfiguriert werden, sofern es einen guten Grund gibt dies zu tun. Bei längeren Benchmarks (wie zum Beispiel dem SPEC CPU2000) ist die Zeit ein solcher Grund.

2.2. T-Test

Der hier betrachtete Zweistichproben-t-Test ist ein statistischer Signifikanztest (Henze, 2013, S. 350) der verwendet wird um die Signifikanz eines Unterschieds der Messungen zweier Programme zu bestimmen. Er stellt eine Ergänzung zu der Verwendung der relativen Standardabweichung in diesem Feld da und sollte im Zweifel verwendet werden Heiser. Dies ist besonders wichtig, wenn die Standardabweichung relativ hoch ist, zum Beispiel wenn man die im Kapitel 4 erläuterten Randomisierungsmechanismen verwendet.

Hypothese Die in der Regel aufgestellte Hypothese ist, dass die Messwertemengen zweier Programme aus verschiedenen Grundverteilungen (auch Populationen genannt) stammen. Das also die beiden Programme ein unterschiedliches Verhalten bzgl. der gemessenen Eigenschaft zeigen.

Nullhypothese Als Nullhypothese nun die Hypothese bezeichnet, das die Hypothese nicht gilt. Das also jegliche Unterschiede in Standardabweichung, Mittelwert, usw. nur daher rühren, dass zu wenige Messungen gemacht wurden.

Statistischer Signifikanztest Ein statistischer Signifikanztest schätzt die Wahrscheinlichkeit, dass die Nullhypothese gilt.

Aber es gilt, dass die Gültigkeit der Nullhypothese nicht impliziert, dass die beiden Messverteilungen aus der selben Grundverteilung stammen. Das also Statistische Signifikanztests nicht zeigen können, dass zwei Programme sich bzgl. der gemessenen Eigenschaft identisch verhalten (Janczyk u. Pfister, 2015, S. 38).

Signifikanzniveau Wenn die geschätzte Wahrscheinlichkeit unter einem gewissem Wert (üblicherweise 5% oder 1% (Henze, 2013, S. 334)) liegt, wird die Nullhypothese verworfen und damit eine Differenz zwischen den Mittelwerten zweier Messwertemengen, zumindest bzgl. dem verwendeten Test, signifikant.

Es gibt neben dem von William Sealy Gosset unter dem Pseudonym Student publizierten t-Test (Henze, 2013, S. 339), noch weitere Tests. Der t-Test wird oft bevorzugt verwendet, weil er relativ robust gegenüber nicht normalverteilten Messwertemengen ist Sawilowsky u. Blair (1992).

temci ermöglicht es mehrere Programme so lange zu benchmarken, bis die Signifikanztests aller Programmpaare entweder eine hohe Wahrscheinlichkeit für die Nichtsignifikanz oder Signifikanz ergeben.

3. Implementierung und Entwurf

Dieses Kapitel behandelt zum einen die für die Implementierung gewählte Plattform und zum anderen den groben Entwurf von *temci*.

3.1. Plattform

Als Plattform für *temci* wurde Linux[1] in Kombination mit Python 3 gewählt.

Linux wurde gewählt, weil es am Lehrstuhl von Prof. Dr.-Ing. Gregor Snelting das vorherrschende Betriebssystem ist und der vom Lehrstuhl entwickelte Compiler auf diesem System läuft. Damit ist es möglich diesen Compiler mit *temci* zu benchmarken (siehe Kapitel 6).

Python wurde gewählt, weil es für diese Sprache zum einen viele wissenschaftliche Bibliotheken wie *SciPy*, *NumPy* oder *Matplotlib* gibt, welche die Implementierung von *temci* erleichtert haben. Zum anderen ermöglicht es Python eleganten Code mit wenig Aufwand zu produzieren und schnell Fortschritte zu erzielen. Dies ist auch der Grund, warum *temci* viel Funktionalität bietet.

Als Python Version wurde die Version 3[2] gewählt, da sie gegenüber der Version 2 eine einfachere Arbeit mit Zeichenketten und Funktionsannotationen Collin Winter (2006) bietet. Letzteres hilft dabei die Argumenttypen und Rückgabewerttypen von Funktionen genauer zu dokumentieren van Rossum u. a. (2014), was besonders bei Projekten, welche über einen längeren Zeitraum entwickelt werden, von Vorteil ist.

Als vorrangig unterstützte Hardwarearchitektur wurde `x86` (in 64 Bit und 32 Bit) gewählt, da diese am Lehrstuhl von Prof. Dr.-Ing. Gregor Snelting die vorherrschende Architektur ist. Außerdem unterstützt *temci* auch ARM (in 64 Bit und 32 Bit) als zweitrangige Hardwareplattform, wobei einige Funktionen darauf nicht zur Verfügung stehen.

[1] Neben Linux wird Apples OS X Betriebssystem rudimentär unterstützt.
[2] Es werden alle Python Versionen oberhalb der Version 3.2 unterstützt.

3.2. Entwurf

Beim Entwurf von *temci* wurde ein modulares Design gewählt. *temci* besteht aus mehreren Programmteilen, welche stark voneinander getrennt sind. Das heißt, dass es keine impliziten Abhängigkeiten zwischen ihnen in Bezug auf die Daten gibt. Die meisten Programmteile haben selbst wiederum in der Regel auch eine modularisierte Struktur.

Die 3 großen Programmteile sind *Build* (Randomisiertes Bauen, siehe Kapitel 4), *Run* (Benchmarken) und *Report* (Reporterzeugung). Sie bestehen jeweils aus einem sogenannten *Processor*, welcher als Fassade die Benutzung der anderen Teile des jeweiligen Programmteils vereinfacht und direkt von der Kommandozeilenschnittstelle aufgerufen wird. Die 3 Teile werden explizit getrennt vom Benutzer verwendet. Sie bilden dabei eine Pipeline, das heißt zuerst wird der *Build* Teil verwendet, dieser bekommt eine Konfigurationsdatei und gibt eine Konfigurationsdatei für den danach verwendeten *Run* Teil aus, der *Run* Teil produziert ein Benchmarkergebnisdatei welche vom *Report* Teil verwendet wird um einen Report zu generieren. Die Formate der jeweiligen Dateien sind darauf hin konzipiert, dass sie auch einfach von anderen Anwendungen oder dem Benutzer selbst produziert werden können. Was alle 3 Dateiformate eint, ist das sie auf dem YAML Format Ben-Kiki u. a. (2005) aufbauen.

Im Folgenden werden die jeweiligen Programmteile von *temci* erläutert und dabei auch die jeweiligen Dateiformate beschrieben.

3.3. Build

temci build ist für das Bauen von Programmen aus ihrem Quellcode zuständig. Dabei kann es mit Versionskontrollsystemen umgehen und das jeweilige Programm auch mehrfach randomisiert bauen (näheres dazu im Kapitel 4). Die Struktur von *temci build* ist einfach, es gibt einen *BuildProcessor* welcher die Verarbeitung der Konfigurationsdateien übernimmt und Instanzen der *Builder* Klasse verwendet um die Programme zu bauen.

Die Konfigurationsdatei besteht wie bei jedem der 3 großen Teile von *temci* aus einer Liste von Konfigurationsblöcken für das jeweilige Programm, diese haben die folgende Struktur:

`attributes` Die Attribute die das jeweilige Programm beschreiben.

`description` Das optionale Attribut zur textuellen Beschreibung des jeweiligen Programms.

`base_dir` Das Verzeichnis, indem der Quellcode und die später zum Benchmarken verwendeten Dateien liegen. Alle weiteren Pfadangaben sind relativ zu diesem Verzeichnis.

`branch` Sofern ein Versionskontrollsystem verwendet wird, gibt dies den verwendeten Zweig an und

`revision` die verwendete Revision.

`working_dir` Das Verzeichnis, indem das Programm gebaut wird.

`build_cmd` Das Kommando um das Programm zu bauen.

`number` Die Anzahl der Male, die das Programm gebaut wird.

`randomization` Die Konfiguration für die Assemblerrandomisierung, welche im Kapitel 4 ausgeführt wird.

Der `attributes` und `run_config` Teil werden in die produzierte Konfigurationsdatei für den *Run* Teil übernommen, wobei der `run_config` Teil leicht modifiziert wird. Es ist wichtig zu erwähnen, dass der *Build* Teil nur mit einem bestimmten *RunDriver* (dem *ExecRunDriver*, siehe folgender Abschnitt) zusammenspielt.

3.4. Run

Der *Run* Teil von *temci* ist für das Benchmarken von Programmen zuständig. Er hat im Wesentlichen die folgende (Klassen-)Struktur:

RunProcessor Dieser verwendet die weiteren Teile des *Run* Teils um die Programme zu Benchmarken.

RunWorkerPool Die Instanzen dieser Klasse dienen dazu, das eigentliche Benchmarken (auch parallel) mithilfe von *RunDriver* Instanzen durchzuführen. Es gibt zwei Varianten: Eine zum sequenziellen und eine weitere zum parallelen Benchmarken.

RunDriver Unterklassen dieser Klasse kümmern sich um das eigentliche Benchmarken und können Plug-in-artig registriert werden. Der einzige zurzeit implemen-

tierte *RunDriver* ist der *ExecRunDriver*. Dieser ist für das Benchmarken von Programmen, die auf der Kommandozeile ausgeführt werden, zuständig.

ExecRunDriver Dieser *RunDriver* hat wiederum verschiedene sogenannte Runner, welche sich um die konkreten Messungen kümmern und dabei zum Beispiel auch SPEC CPU Benchmarks zu unterstützen.

RunDriverPlugins Viele weitere Funktionen können mithilfe sogenannte *RunDriverPlugins* implementiert werden und beim jeweiligen *RunDriver* registriert werden. Die meisten Features, die in den folgenden zwei Kapiteln beschrieben werden sind als Plug-ins implementiert.

Die Konfigurationsdatei besteht wie bei jedem der 3 großen Teile von *temci* aus einer Liste von Konfigurationsblöcken für das jeweilige zu benchmarkende Programm, diese haben die folgende Struktur:

`attributes` Die Attribute die das jeweilige Programm beschreiben

`description` Das optionales Attribut zur textuellen Beschreibung des jeweiligen Programms

`run_config` Dieser Teil dient zur Konfiguration des jeweiligen *RunDrivers*. Die konkrete Struktur hängt von dem gewählten *RunDriver* ab.

Ein Beispiel für eine solche Datei findet sich im Abschnitt 6.4.3.

3.5. Report

temci report ermöglicht die Generierung von anspruchsvollen Reports, das heißt von visuellen und textuellen Darstellungen des Benchmarkergebnisses. Der *Report* Teil besteht im Wesentlichen aus verschiedenen *Reportern* und einem Statistikframework, welches die Arbeit mit den Benchmarkergebnissen vereinfacht und auch die Erzeugung von Diagrammen ermöglicht. Weitere Reporter können über eine Plug-in-Architektur hinzugefügt werden.

Die benötigte Datendatei besteht aus einer Liste von Blöcken für das jeweilige gebenchmarkte Programm. Ein solcher Block hat die folgende Struktur:

`attributes` Die von den anderen *temci*-Teilen bekannten Attribute.

`data` Dies ist eine Verzeichnisdatenstruktur, welche für jede gemessene Eigenschaft eine Liste von Messwerten beinhaltet.

4. Normalisierung durch Randomisierung

4.1. Einführung und Grundlagen

Das Problem beim Benchmarking sind Effekte, die durch unbedeutend scheinende Änderungen der Benchmarkingumgebung oder des gebenchmarkten Programms hervorgerufen werden. Im Normalfall treten solche Effekte nicht bei der Durchführung von Benchmarks auf. Denn die Benchmarkingumgebung ändert sich im Verlauf eines Benchmarks meistens genauso wenig wie das gebenchmarkte Programm. Deswegen werden diese Effekte in den wenigsten Publikationen erläutert (vgl. Kapitel B). Damit ist aber das, aus den Benchmarks abgeleitete, Ergebnis potenziell von einer ganz bestimmten Systemumgebung und einer ganz bestimmten Ausprägung des gebenchmarkten Programms abhängig. Dies ist ein Problem, weil darunter die Reproduzierbarkeit der Ergebnisse leidet.

Im Folgenden wird angenommen, dass ein laufendes Programm aus einem Prozess besteht. Besteht es aus mehreren, so treffen die Aussagen bzgl. des einen Prozesses auf alle Prozesse zu.

Konkret verändern diese scheinbar unbedeutenden Änderungen das Speicherlayout des laufenden Programms (auch Adressraumlayout genannt). Das Layout sieht bei einem Linuxsystem auf der x86-Architektur wie folgt aus (Silberschatz u. a., 2010, S. 359, S. 827), hierbei werden nur die wichtigsten Speichersegmente angegeben.

Stack Das Stack-Segment beginnt an der größten Speicheradresse und wächst nach unten.

Memory-gemappte Dateien In diesem Bereich finden sich zum Beispiel die zum Programm dazugebundenen Bibliotheken.

Heap Das Heap-Segment wächst von unten nach oben.

BSS Beim Programmstart noch nicht initialisierte Daten.

Data Beim Programmstart bereits initialisierte Daten.

Code Der (Assembler-)Code des Programms.

Der (physikalische) Hauptspeicher selbst ist in gleichgroße Blöcke, sogenannte Frames, aufgeteilt (Silberschatz u. a., 2010, S. 370). Jedes laufende Programm bekommt jeweils einen bestimmten Teil dieser Frames zugewiesen. Die Abbildung der Speichersegmente auf die Frames und die Ausrichtung von Daten in ihnen ist nun von Bedeutung, denn davon hängt zum Beispiel ab, wie gut ein bestimmtes Datum in einem Segment gecached werden kann oder welche Speicherbereiche mit welchen anderen beim Caching im Konflikt stehen. Solche Effekte können nicht hervorgesagt werden, da das dafür nötige Wissen über die Hardware von den Herstellern nicht veröffentlicht wird Diwan u. a. (2009).

In Diwan u. a. (2009) werden konkret Änderungen von zwei unbedeutend scheinenden Dingen erläutert:

Größe der Umgebungsvariablen Die Umgebungsvariablen dienen als flexible Möglichkeit um Programme zu konfigurieren. Sie werden von einem Prozess zu den von ihm erzeugten Kindprozessen automatisch weitergereicht (Silberschatz u. a., 2010, S. 813). Ein typische Umgebungsvariable ist `LANG`, die die aktuelle Systemsprache angibt.

Die Umgebungsvariablen werden am Anfang (am oberen Ende) des Stacks eines Programms gespeichert. Damit folgt aus einer Änderung der Größe der Umgebungsvariablen automatisch eine Veränderung der Speicherausrichtung der Daten auf dem Stack.

Da in den Umgebungsvariablen oft auch das aktuelle Arbeitsverzeichnis gespeichert wird, kann damit bereits dessen bloßer Wechsel zu Änderungen beim Benchmark führen.

Linkreihenfolge beim Bauen eines Programms Durch die Änderung der Reihenfolge mit welcher die einzelnen Teile (die Objektdateien) eines Programms beim Bauen zusammengefügt werden, kommt es zur Veränderung des Layout der Code-, Data- und BSS-Segmente. Daraus resultieren die oben bereits erwähnten Cacheeffekte. Außerdem kann durch die Veränderung des Code-Segments auch das Verhalten der Sprungvorhersage Einheit des Prozessors beeinflusst werden.

Außerdem wird in Curtsinger u. Berger (2013) erläutert, dass auch feingranularere Änderungen im Code-, BSS- und Data- Segment ergebnisverändernde Auswirkungen haben können. Mit feingranularen Änderungen ist in Bezug auf das Code Segment die relative Positionierung von Funktionen zueinander gemeint. Und in Bezug auf

das Data-, wie auch das BSS-Segment, die relative Positionierung der einzelnen Datenblöcken in ihnen.

Die Auswirkung dieser Änderungen können nur schwer hervorgesagt werden und die Prozessorcaches können zwar ausgeschaltet werden, was aber, wie in Abschnitt 4.5 erläutert, nicht praxistauglich ist. Deswegen bleibt nur die Möglichkeit, die erwähnten Änderungen bei jeder Messung zufällig neu durchführt. Das heißt, dass bei jedem Messvorgang die Umgebungsvariablen eine andere Größe haben und das Speicherlayout ein anderes ist. Letzteres kann durch das vielfach randomisierte Bauen der Anwendung geschehen, womit bei jeder Messung eine andere Binärdatei verwendet werden kann. Durch diese Randomisierung wird die Streuung der Messwerte vergrößert. Die Benchmarkergebnisse welche auch unter dieser erhöhten Streuung noch signifikant sind, sind damit robust gegen diese Änderungen.

Die folgenden drei Abschnitte beschreiben die konkret in *temci* implementierten Mechanismen zur Randomisierung. Der darauf folgende Abschnitt 4.5 beschreibt das Ausschalten der Prozessorcaches via *temci* und der letzte Abschnitt beschreibt kurz, die Randomisierung der Messreihenfolge beim sequenziellen Benchmarken mehrerer Programme.

4.2. Randomisierung der Umgebungsvariablen

Dieser Mechanismus fügt zufällig Umgebungsvariablen zu jenen hinzu, welche einem zu benchmarkenden Programm bei dessen Aufruf mitgegeben werden. Dies geschieht, indem zufällig viele Umgebungsvariablen mit jeweils zufälliger Größe zu den bereits vorhandenen Umgebungsvariablen hinzugefügt werden. Da die Größe der hinzugekommenen Variablen in der Regel um Größenordnungen größer als jede „unbedeutende Änderungen“ der eigentlichen sind, haben diese Änderungen keine signifikante Bedeutung mehr.

Ein konkretes Beispiel für die Auswirkungen dieses Mechanismus ist im Abschnitt 6.4.4 gegeben.

4.3. Randomisierung der Linkreihenfolge

Dieser Mechanismus randomisiert, wie im Abschnitt 4.1 näher erläutert wurde, verschiedene Speichersegmente, indem er die Reihenfolge ändert, in welcher die einzelnen Codedateien zusammengefügt werden.

Ein konkretes Beispiel für die Auswirkungen dieses Mechanismus ist im Abschnitt 6.4.4 gegeben.

4.3.1. Implementierung

Die Implementierung dieses Mechanismus in *temci* macht sich zu nutze, dass der GCC (und auch der CParser des libFirm Projektes) grob wie folgt funktioniert:

1. Übersetze alle Codedateien zu Assemblerdateien.

2. Verwende das Programm `as`[1] um die Assemblerdateien zu assemblieren. `as` bekommt im Wesentlichen eine Assemblerdatei übergeben und produziert eine sogenannte Objektdatei.

3. Verwende das Programm `ld`[2] um die verschiedenen Objektdateien, Bibliotheken und weitere Dateien zu einer fertigen Binärdatei zusammenzufassen. Dieser Vorgang wird auch „Linken“ genannt.

Zur Umsetzung des Mechanismus wurde nun ein kleines Programm Namens `ld` geschrieben. Dieses Programm ruft das originale `ld` Programm mit den gleichen Argumenten, mit denen es selbst aufgerufen wurde, auf. Der einzige Unterschied ist, dass es die Reihenfolge der an `ld` übergebenen Dateien und Bibliotheken vertauscht wurde[3]. Wird nun der Ordner, in welchem sich das *temci* eigene Version befindet, an den Anfang der `PATH` Umgebungsvariable[4] angefügt, wird bei jedem Aufruf des Programms `ld` statt dem originalen, die Version von *temci* aufgerufen.

Damit kann die Linkreihenfolge bei allen Compiler- und Buildskripten recht einfach randomisiert werden, sofern diese das Programm `ld` zum Zusammenfügen von Codedateien verwenden. Zu dieser Sorte Compiler zählen, wie bereits erwähnt, GCC und CParser, jedoch nicht der Compiler Clang des LLVM Projektes. Letzterer ruft weder zum Linken noch zum Assemblieren ein externes Programm auf, weshalb eine Randomisierung der Linkreihenfolge bei diesem Compiler nicht ohne Änderung des Buildprozesses selbst funktioniert.

[1] Auf Linuxsystemen ist `as` im Normalfall der GNU Assembler.
[2] Auf Linux-Systemen ist `ld` im Normalfall der GNU Linker.
[3] Die *temci* Version von `ld` kann mit mehreren Umgebungsvariablen konfiguriert, die Randomisierung ein- und ausgeschalten und den Pfad zur „richtigen“ Version von `ld` angeben werden.
[4] Die `PATH` Umgebungsvariable enthält eine Liste von Verzeichnissen. Wird ein Programm ohne konkrete Angabe seines Dateipfades aufrufen, wird nach ihm in den Verzeichnissen dieser Liste gesucht. Hierbei wird die Liste von vorne nach hinten sequentiell durchgegangen.

4.4. Randomisierung des Assemblers

Im Folgenden wird der grundlegende Mechanismus zum Randomisieren von Assemblercode an einem Beispiel erläutert. Der Beispielassemblercode 2 wurde mithilfe des CParsers aus dem C-Code 1 erzeugt. Der Code macht nichts weiter als in einer Methode `func` Speicher auf dem Heap zu allozieren und danach „Hallo Welt“ auszugeben.

```c
#include <stdio.h>
#include <stdlib.h>

void func(int bytes){
        (void)malloc(bytes);
}

int main(int argc, char **argv){
        func(10);
        printf("Hallo Welt");
        return 0;
}
```

Code 1: Einfaches C-Programm welches 10 Bytes auf dem Heap alloziert und „Hallo Welt“ ausgibt.

Am Anfang jeder Randomisierung steht das Aufteilen des Assemblercodes in semantisch zusammengehörige Blöcke. Wichtig ist dabei besonders, dass der Assemblercode jeder Funktion (wie `func` oder `main`) in einem eigenen Block liegt. Für diese Trennung in Blöcke wird eine eigene Heuristik für CParser-Assembler und den GCC-Assembler verwendet.

Hierbei wird Assemblercode als CParser-Assembler betrachtet, falls mindestens eine Zeile im Assembler zum regulären Ausdruck `#[- ]* Begin` passt. Dieser reguläre Ausdruck passt zu allen Zeilen welche mit einem Doppelkreuz, gefolgt von einem oder mehreren Leerzeichen und Bindestrichen und dem Wort `Begin`, beginnen. Der Assemblercode wird als GCC-Assembler betrachtet, falls mindestens eine Zeile mit `.cfi` beginnt.

Die Heuristiken für beide Compiler sind folgende:

CParser-Assembler Wie man im Code 2 sieht, sind die einzelnen Blöcke durch Leerzeilen getrennt. Diese einfache Heuristik funktioniert gut, da der vom CParser produzierte Assembler eine klare Struktur hat.

GCC-Assembler Die einzelnen Blöcke sind in dieser Art Assembler oft durch Segmentdirektiven (zum Beispiel `.text` oder `.section rodata`) getrennt. Diese Segmentdirektiven geben an, zu welchem Segment die nachfolgenden Zeilen gehören.

```
1  .text
2  # -- Begin  func
3  .p2align 4,,15
4  .globl func
5  .type        func, @function
6  allocate_heap:
7  .L0:
8  pushq %rbp
9  mov %rsp, %rbp
10 movslq %edi, %rdi
11 call malloc
12 leave
13 ret
14 .L1:
15 .size        func, .-func
16 # -- End  func
17
18 # -- Begin  main
19 .p2align 4,,15
20 .globl main
21 .type        main, @function
22 main:
23 .L200:
24 pushq %rbp
25 mov %rsp, %rbp
26 movl $0xA, %edi
27 call func
28 movl $0x8, %eax
29 movq $.Lstr.0, %rdi
30 call printf
31 movl $0x0, %eax
32 leave
33 ret
34 .L201:
35 .size        main, .-main
36 # -- End  main
37
38 .section        .rodata
39 .Lstr.0:
40 .asciz "Hallo Welt"
```

Code 2: Vom CParser generierter Assemblercode für den Code 1, wobei die hier unwichtigen Kommentare weggelassen wurden.

Das Problem ist, dass diese Heuristik nur bei rund der Hälfte des Codes zu trifft.

Für die einzelnen Blöcke wird nun noch entschieden ob sie zu einer Funktion oder zu einem Datensegment gehören. Falls eine der Zeilen im Block zum regulären Ausdruck `#[- ]* Begin` passt oder ein Label ist, welches nicht mit einem . beginnt, handelt es sich beim Block um einen Funktionenblock, ansonsten um einen Datensegmentblock.

Nun ist es möglich, den Assembler zu randomisieren. Die verschiedenen Randomisierungen werden im Folgenden erläutert. Welche Randomisierungen konkret verwendet werden, können in *temci* konfiguriert werden.

4.4.1. Randomisierung der Blockreihenfolge

Bei der Randomisierung der Blockreihenfolge, werden alle Blöcke zufällig miteinander vertauscht. Wobei beim GCC-Assembler der erste Block festgehalten wird, da sich in

ihm die folgende Präambel befindet, hier im zu 2 vergleichbaren GCC-Assemblercode, deren Position festgehalten werden sollte:

```
.file        "hello.c"
```

4.4.2. Randomisierung des Heaps

Hierfür wird jeder Aufruf der Methoden `malloc` `calloc`, `_Znwm` und `_Znam` im Assembler betrachtet, wobei die letzten beiden zum `new` Operator in C++ gehören und weitere Methoden einfach hinzugefügt werden können.

Ein solcher Aufruf sieht zum Beispiel wie folgt aus (er entspricht den Zeilen 10 und 11 aus Code 2):

```
movslq %edi, %rdi
call malloc
```

Das erste und einzige Argument, das `malloc` übergeben bekommt (bei 64 Bit Systemen im Register `rdi`, bei 32 Bit Systemen in `edi`) ist die Anzahl an Bytes, die auf dem Heap alloziert werden sollen. Auf diese Anzahl wird nun zur Randomisierung ein zufälliger Wert addiert. Dies funktioniert, indem mithilfe des Assembler-Kommandos `addq` der Wert auf das Register `rdi` bzw. `edi` addiert wird. Der obige Code sieht dann (auf einem 64 Bit System) wie folgt aus, wobei als zufälliger Wert `10` angenommen wird:

```
movslq %edi, %rdi
addq 10, %rdi
call malloc
```

Durch diese Modifikation wird mehr Speicher als vom Code benötigt alloziert. Dies hat damit keine negativen Auswirkungen auf den Code, abgesehen vom erhöhten Speicherverbrauch und einer zusätzlichen Addition pro Allokation. Diese negativen Auswirkungen sind nur bei Code relevant, welcher viele Allokationen durchführt und lange läuft.[5]

Die Struktur des Heaps und die Speicherausrichtung der einzelnen Daten auf dem Heap werden hierdurch verändert.

[5] Deswegen wird die Randomisierung des Heaps nicht beim Benchmarken von Programmen wie jenen des SPEC CPU2000 Benchmarks verwendet.

4.4.3. Randomisierung der Datensegmente

Zur Randomisierung der Datensegmente werden die dazugehörigen Assemblerblöcke betrachtet. Ein Beispiel aus dem Code 2 ist folgender Block:

```
38          .section        .rodata
39          .Lstr.0:
40          .asciz "Hallo Welt"
```

Oft folgen hinter den beiden letzten Zeilen noch weitere, welche der gleichen Natur sind.

Diese Blöcke werden in Subblöcke aufgeteilt, welche aus einer Labelzeile (hier `.Lstr.0:`) und einer Wertzeile (hier `.asciz "Hallo Welt"`) bestehen und einem Umgebungsteil. Die Subblöcke werden nun zufällig miteinander vertauscht. Damit ändert sich die Speicherausrichtung der einzelnen Daten im jeweiligen Datensegment.

4.4.4. Implementierung

Die Implementierung dieses Mechanismus ist ähnlich zur Implementierung des vorherigen Mechanismus (siehe Abschnitt 4.3.1), nur dass diesmal ein Hüllprogramm für den Assembler `as` entwickelt wurde. Dieses Hüllprogramm bekommt, wie auch `as` selbst, als erstes Argument den Pfad der zu verarbeitenden Assemblerdatei übergeben. Diese wird nun, wie vorher beschrieben, randomisiert, unter dem gleichen Namen wieder gespeichert. Danach ruft es das „richtige“ `as` Programm mit den Argumenten auf, mit denen es selbst aufgerufen wurde. Falls dies fehlschlägt, weil der randomisierte Assembler fehlerhaft ist, wird der gesamte Vorgang mehrfach wiederholt, mit immer schwächerer Randomisierung, bis das `as` Programm die randomisierte Assemblerdatei akzeptiert. Deswegen führt ein Fehler beim Randomisieren nicht dazu, dass die Assemblerdatei gar nicht assembliert wird und der ganz Buildvorgang fehlschlägt.

Diese Art der Implementierung hat ihre Schwächen, zum einen werden nur Compiler unterstützt, welche das `as` Programm explizit aufrufen. Zum anderen müssen die verwendeten Heuristiken auf jedes Assemblerformat, auf jeden Compiler, neu angepasst werden. Zurzeit werden, wie weiter oben bereits erwähnt, nur der GCC[6] und der CParser unterstützt, wobei die Unterstützung für den GCC noch fehleranfällig ist und es bei der Ausführung von randomisierten Anwendungen zu Fehlern kommen kann.

[6] Beim GCC werden C und C++ unterstützt und über Umwege auch Haskell unter der Verwendung des GHCs.

Einer der Vorteile dieser Implementierung ist, dass sie bei allen Buildskripten, die einer der unterstützten Compiler verwenden, verwendet werden kann, ohne das diese selbst verändert werden müssen. Es reicht das Verzeichnis, welches das Hüllprogramm enthält, an den Anfang der `PATH` Umgebungsvariable anzufügen.

4.4.5. Weitere Ansätze

Der in *temci* implementierte Ansatz randomisiert die Assemblerdateien und damit die resultierenden Binärdateien statisch. Das heißt, dass keinerlei Randomisierung zur Laufzeit vorgenommen wird. Werden viele Modifikationen, wie zum Beispiel die Randomisierung des Heaps, zur Laufzeit dynamisch vorgenommen, hat dies den Vorteil, dass es mehr Möglichkeiten gibt das Programmverhalten zu beeinflussen. Der Nachteil dieser dynamischen Modifikationen ist aber, dass die Entwicklung viel aufwendiger ist und den Rahmen dieser Arbeit gesprengt hätte. Dieser Ansatz wird vom Programm Stabilizer Curtsinger u. Berger (2013) verfolgt, welches als Compilerstufe für GCC und Clang implementiert wurde. Dadurch, dass Stabilizer mit dem Compiler direkt verknüpft ist, sind weitere Mechanismen zur Randomisierung möglich, die mehr Kontextinformationen benötigen, als im Assembler vorhanden sind.

Die Randomisierung mit diesem Programm ist besser, als mit der in *temci* implementierten, dafür ist es nicht möglich Stabilizer, ohne große Veränderungen der Buildskripte, zu verwenden. Da Stabilizer als Eingabe nur C-, C++- oder Fortran-Dateien akzeptiert ist es auch nicht möglich Stabilizer einfach in *temci* zu integrieren, um die Vorteile beider Ansätze zu vereinen. Außerdem unterstützt Stabilizer nicht die aktuellen Versionenen von GCC und Clang, was die Integration zusätzlich erschwert.

4.5. Ausschalten der CPU Caches

Dieser Mechanismus ist eher exotischer Natur, da es wegen der Vergrößerung der Laufzeit eines Programms um Größenordnungen eher unpraktikabel ist und es sehr wenige konkrete Anwendungsfälle gibt, in denen die CPU Caches ausgeschaltet sind. Weswegen es zu einem völlig synthetischen Benchmark wird.

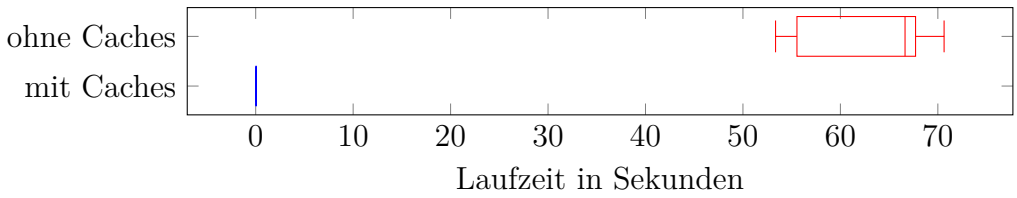
Er ist zwar konkret keine Form der Randomisierung, bezweckt aber, genauso wie die anderen Mechanismen die Reduzierung der Cacheeffekte.

In CPUs auf Basis der x86er Architektur ist es möglich die CPU Caches auszuschal-

ten. Konkret bedeutet dies, dass jeder Cachezugriff zu einem Zugriffsfehler, einem sogenannten Cache Miss, führt und damit für jeden jeden Speicherzugriff ein Zugriff auf den Hauptspeicher ausgeführt wird. Dies führt dazu, dass die Caches keinen Effekt mehr auf die Laufzeit haben und jeder Speicherzugriff, der nicht direkt auf Register zugreift, um Größenordnungen langsamer wird. Selbst die Laufzeit von Programmen, die relativ wenige Speicherzugriffe tätigen, wird damit von der Speicherperformanz bestimmt.

4.5.1. Kurzevaluation

Als Beispiel, wie sich das Ausschalten der Caches auf die Ausführungszeit auswirkt, sei der Benchmark eines Programms zur Berechnung von 30.000 Nachkommastellen von π gegeben. Das Programm Granlund (2002) stammt von Hanhong Xue, wurde mithilfe der C-Mathematikbibliothek GMP entwickelt und mit dem GCC mit der Optimierungsstufe `-O3` kompiliert. Die Messungen fanden auf dem unter Benchmarkingplattform beschriebenen System statt. Die gemessene Laufzeit ist im Folgenden angegeben.



	Mittelwert	$\frac{\text{Standardabweichung}}{\text{Mittelwert}}$	Anzahl der Messungen
ohne Caches	62.779	11%	60
mit Caches	0.039	3%	205

Ein vergleichbares Bild ergibt sich bei der Anzahl der Cache Misses:

	Mittelwert	$\frac{\text{Standardabweichung}}{\text{Mittelwert}}$	Anzahl der Messungen
ohne Caches	508734546	7%	60
mit Caches	202380	14%	205

Ohne Caches läuft das Programm damit um rund Faktor 16 000 langsamer und hat um rund Faktor 2500 mehr Cache Misses.

4.5.2. Implementierung

Um die CPU Caches auszuschalten muss im Kernel Modus, dass heißt in einem Betriebssystemmodul, das sogenannte Cache *Disable Bit* des CPU-Kontrollregisters CR0 auf 1 gesetzt werden. Dieser Bit wird in Inc. (2013) für die AMD64 Architektur (entspricht der x86 Architektur in der 64 Bit Variante) wie folgt beschrieben:

> **Cache Disable (CD) Bit.** Bit 30. When CD is cleared to 0, the internal caches are enabled. When CD is set to 1, no new data or instructions are brought into the internal caches. However, the processor still accesses the internal caches when CD = 1 under the following situations:
>
> - Reads that hit in an internal cache cause the data to be read from the internal cache that reported the hit.
>
> - Writes that hit in an internal cache cause the cache line that reported the hit to be written back to memory and invalidated in the cache.
>
> Cache misses do not affect the internal caches when CD = 1. Software can prevent cache access by setting CD to 1 and invalidating the caches.

Davor werden die Daten in den CPU Caches mit der Instruktion `WBINVD` (Write Back and Invalidate Cache) noch in den Hauptspeicher zurückgeschrieben, um Datenverluste zu vermeiden.

Zu Beginn des gesamten Messvorgangs wird ein Betriebssystemmodul geladen, dessen Code im Wesentlichen aus dem Quellcode des Werkzeugs `memtest86++` stammt und die Caches ausschaltet. Nach der Beendigung wird das Modul entfernt und dabei die Caches wieder aktiviert.

4.6. Randomisierung der Reihenfolge

Dieser Mechanismus funktioniert, indem die Reihenfolge, in der die einzelnen Programme gebenchmarkt werden, randomisiert wird. Dadurch wird die Wahrscheinlichkeit verringert, dass das resultierende Benchmarkergebnis von der Aufrufreihenfolge der Programme abhängt. Außerdem ist erkennbar, falls ein Programm mit einer bestimmten Programmreihenfolge signifikant schneller läuft. Denn dabei ist die Standardabweichung höher als erwartet und man sieht im Boxplot einen Ausreißer.

Dieser Mechanismus wird von *temci* als einziger der Mechanismen dieses Kapitels

standardmäßig verwendet.

5. Normalisierung durch Separation

Ein weiteres Problem beim Benchmarking ist der Einfluss von anderen Programmen auf die Messergebnisse und die gegenseitige Beeinflussung von nacheinander gebenchmarkten Programmen (siehe Abschnitt 5.4). Diese Beeinflussungen vergrößern die Streuung der Messwerte und reduzieren damit die Qualität der Benchmarks.

Um die gebenchmarkten Programme von den Restprogrammen, das heißt Programme die nicht gerade gebenchmarkt werden, zu separieren, gibt es zwei Möglichkeiten: Das Anhalten oder Verlangsamen der Ausführung der Prozesse der Restprogramme (siehe Abschnitte 5.1 und 5.2) und die (physische) Trennung von den gebenchmarkten Programmen (siehe Abschnitt 5.3). Wobei Letztere auch gegen die gegenseitige Beeinflussung parallel gebenchmarkter Programme hilft.

5.1. Prozessprioritäten

Bei dieser Möglichkeit zur Reduktion des Einflusses der Restprogramme werden die Prioritäten der Prozesse aller selbiger auf einen sehr geringen und jene der gebenchmarkten Programme auf einen sehr hohen Wert gesetzt. Damit werden Letztere bei der Ausführung bevorzugt, was dazu führt, dass sie weniger oft während der Ausführung unterbrochen werden (Silberschatz u. a., 2010, S. 816). Außerdem wird bei diesen Prozesse auch die Priorität beim Zugriff auf den Hintergrundspeicher erhöht, sodass sie bevorzugten Zugriff auf diesen bekommen. Damit haben die Speicherzugriffsmuster der Restprogramme weniger Einfluss auf die Messungen.

5.1.1. Implementierung

Konkret ist dieser Mechanismus implementiert, indem der sogenannten `nice` Werte jedes Prozesses gesetzt werden. Der `nice` Wert kann in einem Bereich von -20 (am vorteilhaftesten) bis 19 (am unvorteilhaftesten für den Prozess) liegen. Bei den Prozessen der Restprogramme wird dieser Wert auf 19 gesetzt und bei jenen

gebenchmarkter Prozesse auf -15[1].

Ausgenommen von der Herabsetzung der Prozesspriorität sind alle Prozesse, welche bereits eine hohe Priorität, einen `nice` Wert kleiner als -10, besitzen oder zur aktuellen *temci* Instanz gehören. Damit wird vermieden, dass Prozesse, die für das Funktionieren des Systems selbst wichtig sind, beeinträchtigt werden.

Die Höhe des `nice` Wertes gibt nicht direkt die Priorität an, um aus man-pages project (2014b) zu zitieren:

> The degree to which their relative nice value affects the scheduling of processes varies across UNIX systems, and, on Linux, across kernel versions. Starting with kernel 2.6.23, Linux adopted an algorithm that causes relative differences in nice values to have a much stronger effect. This causes very low nice values (+19) to truly provide little CPU to a process whenever there is any other higher priority load on the system, and makes high nice values (-20) deliver most of the CPU to applications that require it (e.g., some audio applications).

Die Erhöhung der Speicherzugriffspriorität für die Prozesse gebenchmarkter Programme funktioniert über das Setzen der I/O Scheduling Klasse via `ionice` auf die „Realtime“ Klasse. Damit bekommen diese Prozesse bevorzugten Zugriff auf den Hintergrundspeicher vor allen anderen Benutzerprozessen.

5.2. Anhalten anderer Prozesse

Dieser Mechanismus hält alle Prozesse an, welche nicht zu einem der gebenchmarkten Programme gehören oder wichtig für das Funktionieren des Systems als solches sind.

5.2.1. Implementierung

Um einen Prozess anzuhalten, wird ihm ein POSIX-Signal Namens `Stop` an den selbigen gesendet. Dieses Signal kann von Prozessen nicht ignoriert werden.

Das Problem bei der Implementierung ist die Entscheidung, welche Prozesse für das Funktionieren des Systems und das Benchmarken wichtig sind und welche nicht. Da

[1] Bei einem geringeren Wert als -15 treten Probleme auf und die Ausführungszeit von Programmen wird deutlich langsamer, wie Experimente gezeigt haben.

unwichtige Prozesse angehalten werden, hat ein Fehler bei der Entscheidung große Auswirkungen auf die dazugehörigen Programme.

Die verwendete Heuristik nimmt an, dass Prozesse nur dann sicher unwichtig sind, falls sie einen `nice` Wert >= −10 (vgl. Abschnitt 5.1.1, entspricht einer nicht sehr hohen Priorität), der Prozessname nicht mit „dbus“ oder „kworker“ beginnt (Linux Kernel eigene Prozesse) und die Prozessidentifikationsnummer mindestens 1500 ist oder der Prozessname mit „ssh“, „xorg“ oder „bluetoothd“ beginnt oder einen Elternprozess hat, welcher mit „dm“ endet (im Normalfall alle vom Benutzer gestarteten Prozesse).

5.3. Physische Separation

Bei diesem Mechanismus geht es um die physische Trennung von gebenchmarkten und Restprogrammen. Hierbei werden im Normalfall alle Restprogramme auf eine bestimmte CPU eingeschränkt und die gebenchmarkten Programme (wie auch *temci*) auf die anderen CPUs verteilt.

Im Folgenden werden die Möglichkeiten erläutert, um diese physische Trennung durchzuführen.

5.3.1. CPU Affinitäten

Die einfachste Möglichkeit hierfür ist dass setzen der sogenannten CPU Affinitäten für jeden Prozess jedes Programms. Damit wird für jeden Prozess die Menge an CPUs festgesetzt, auf denen der jeweilige läuft man-pages project (2014c).

Das Problem hierbei ist, dass Prozesse ihre eigenen CPU Affinitäten setzen können Love (2014) und damit die Beschränkungen umgehen können. Dieses Problem gibt es nicht, wenn man sogenannte CPU Sets verwendet, die im Folgenden Unterabschnitt erläutert werden.

5.3.2. CPU Sets

Für die Prozesse nicht gebenchmarkter und gebenchmarkter Programme wird jeweils ein CPU Set erzeugt, dem sie zugewiesen werden. Jedes CPU Set bekommt wiederum eine Liste von CPUs zur Verfügung gestellt. Jeder Prozess in einem CPU Set darf

nur auf genau diesen Prozessoren laufen. Für das Ändern der CPU Sets (und der zugehörigen Prozesse) werden Root-Rechte benötigt, weswegen es für die meisten Prozessen unmöglich ist, diese Änderungen vorzunehmen man-pages project (2014a). Außerdem können CPU Affinitäten nur im Rahmen des zugewiesenen CPU Sets gesetzt werden.

5.3.3. Ausschalten von Hyper-Threading

Trotz der Trennung von Programmen auf verschiedenen CPUs kann es noch aufgrund von Hyper-Threading dazu kommen, dass nicht gebenchmarkte und gebenchmarkte Programme auf den gleichen physischen CPUs laufen.

Mit dem von Intel entwickelten Hyper-Threading ist es möglich eine physische CPU als zwei (logische) CPUs für das Betriebssystem auszugeben. Dies funktioniert, weil die Speicherzugriffslatenzen im Normalfall hoch sind und die CPU die meiste Zeit auf eher langsame Speicherzugriffe wartet Marr u. a. (2002).

Da beim Hyper-Threading beide (logischen) CPUs dieselbe physischen CPU und damit auch dieselben Caches und Speicheranbindungen teilen, kann es zu negativen Beeinflussungen der zwei gleichzeitig auf derselben physischen CPU laufenden Prozessen kommen Magro u. a. (2002).

Deswegen ist es hilfreich bei Benchmarks Hyper-Threading auszuschalten. Dies kann dadurch passieren, dass jeweils einer der beiden (logischen) CPUs einer physischen CPU ausgeschaltet werden.

5.4. Separation von Messungen

Im Folgenden werden zwei Möglichkeiten erläutert, wie man die gegenseitige Beeinflussung von nacheinander durchgeführten Messungen reduzieren kann. Konkret geht es dabei darum, dass das System beim Start jeder Messungen einen möglichst vergleichbaren Zustand hat.

5.4.1. Invalidieren der Dateisystemcaches

Indem die Caches des Dateisystems vor jeder Messung ungültig gemacht werden, haben die Caches beim Start jeder Messung den gleichen Inhalt und es kommt nicht

zu negativen Beeinflussungen durch vorherige Messungen. Da jede Datei, auf die im Laufe der Messung zugegriffen wird, neu vom Hintergrundspeicher geladen werden muss.

5.4.2. Pausen zwischen den Messungen

Indem vor jeder Messung eine längere Pause gemacht wird, kann das System abkühlen, womit der thermische Hardwarezustand beim Start jeder Messung vergleichbar ist. Denn durch die Temperatur des Prozessors wird zum Beispiel bei der Verwendung des sogenannte Turbo Boost bei Intel Prozessoren die Taktfrequenz mit steigender Temperatur verringert Intel (2008), womit der thermale Zustand die Ausführungsgeschwindigkeit direkt beeinflusst.

Außerdem hat das System durch eine längere Pause Zeit um Ein- und Ausgabe-Puffer zu leeren und Aufräumarbeiten (zum Beispiel bei der Verwendung von SSDs fertigzustellen).

6. Evaluation

6.1. Benchmarkingplattform

Das System, auf dem alle Benchmarks in dieser Arbeit angefertigt wurden, soll im Folgenden beschrieben werden, um die Einordnung der gemessenen Werte zu ermöglichen.

Es handelt sich auf der Softwareseite um ein frisch aufgesetztes Ubuntu 15.10 mit dem Linux Kernel Version 4.2.0, der Desktopumgebung Unity und dem Dateisystem `ext4` in Kombination mit `ecrypt fs`. Die GCC Version ist Version 5.2.1. Dieser GCC ist der Standardcompiler auf dem System. Zusätzlich hierzu wurde ansonsten nur noch *temci* (Version 0.5.5) samt Abhängigkeiten und mehreren Versionen des Haskell-Compilers GHC installiert. Im Hintergrund läuft ein SSH Daemon und das `/tmp` Verzeichnis liegt im Arbeitsspeicher (via `tmpfs`). Für die Abschnitte 6.6 und 6.4 wurden libFirm 1.22 und CParser 1.22.1 verwendet.

Auf der Hardwareseite handelt es sich um ein System mit einem x64_64 AMD FX^{TM}-4100 Quadcore Prozessor mit 8 GiB DDR3 RAM und 8 GiB Auslagerungsspeicher. Als einzige Festplatte ist eine 120 GiB SSD verbaut, die nur zu rund einem Drittel belegt ist. Das System als solches ist als Tower aufgebaut.

Alle Benchmarks in dieser Arbeit wurden jeweils am Stück auf dem neu gestarteten System angefertigt. Währenddessen wurde das System nicht benutzt, abgesehen von kurzen `top` Aufrufen zur Kontrolle des Benchmarkvorgangs. Außerdem wurden alle Mechanismen zur Trennung von gebenchmarkten und nicht gebenchmarkten Programmen (siehe Kapitel 5) und die Randomisierung der Benchmarkingreihenfolge (siehe Abschnitt 4.6) verwendet.

6.2. GHC Performanz über die Zeit

Im Folgenden wird die Performanz des GHCs über die Zeit evaluiert, indem die Performanz der GHC Versionen 7.0.1[1], 7.2.1, 7.4.1, 7.6.1, 7.8.1, 7.10.1 und 8.0.1[2] miteinander verglichen wurde. Für die Evaluierung der Entwicklung der Performanz des GHCs über mehrere Versionen, wurde das *Computer Language Benchmarks Game* Fulgham u. Gouy in Teilen reimplementiert.

6.2.1. Computer Language Benchmarks Game

Dieses Spiel besteht daraus, dass dieselben Algorithmen in verschiedenen Programmiersprachen implementiert und mit verschiedenen Eingaben ausgeführt werden. Dies ermöglicht den groben Vergleich verschiedener Programmiersprachen. Dieser Ansatz hat seine Schwächen, weswegen das Ganze auch als „Game“, also als Spiel bezeichnet wird. Eine Schwäche besteht zum Beispiel darin, dass manche der Implementierungen handoptimiert sind und andere nicht. Diese Schwächen treten nicht so stark zutage, wenn nur einzelne Implementierungen einer Sprache verglichen werden. Zu diesem Zwecke wurde auf *temci* aufbauend eine Anwendung Namens *game.py* entwickelt, die genau dies tut.

Im Fall des Vergleichs der verschiedenen GHC Versionen, wurde aus den verfügbaren Algorithmenimplementierungen jene ausgewählt, die übersetzbar und lauffähig mit allen GHC Versionen waren.

Der Haskell-Code im *Computer Language Benchmarks Game* ist in der Mehrheit nicht in einem funktionalen Stil geschrieben und verwendet auch wenig Lazy Evaluation. Deswegen sind die Ergebnisse im weiteren Verlauf dieses Abschnitts mit Vorsicht zu betrachten.

6.2.2. Benchmarken

Jede Algorithmenimplementierung wurde mit allen GHC Versionen übersetzt und pro Eingabe jeweils 20-mal ausgeführt. Die Kombination aus Implementierung und Eingabe wird im Folgenden als „Programm“ bezeichnet.

[1] Die GHC Version 7.0.1 wurde am 16. November 2010 veröffentlicht haskell.org (a).
[2] Als GHC Version 8.0.1 wird der GHC Entwicklungsstand Mitte Januar 2016 bezeichnet.

6.2.3. Ergebnisse

Keine signifikanten Verbesserungen über die Zeit

Im Folgenden wird der Mittelwert der Laufzeiten jedes Programms für jede GHC Version relativ zur GHC Version 7.0.1 betrachtet. Wobei die Standardoptimierungsstufe `-O` haskell.org (b) verwendet wurde. Die Verteilung ist in Diagramm 6.1 dargestellt und deutet an, dass der GHC zwar tendenziell mit der Zeit besser wurde, diese Tendenz aber nicht signifikant ist.

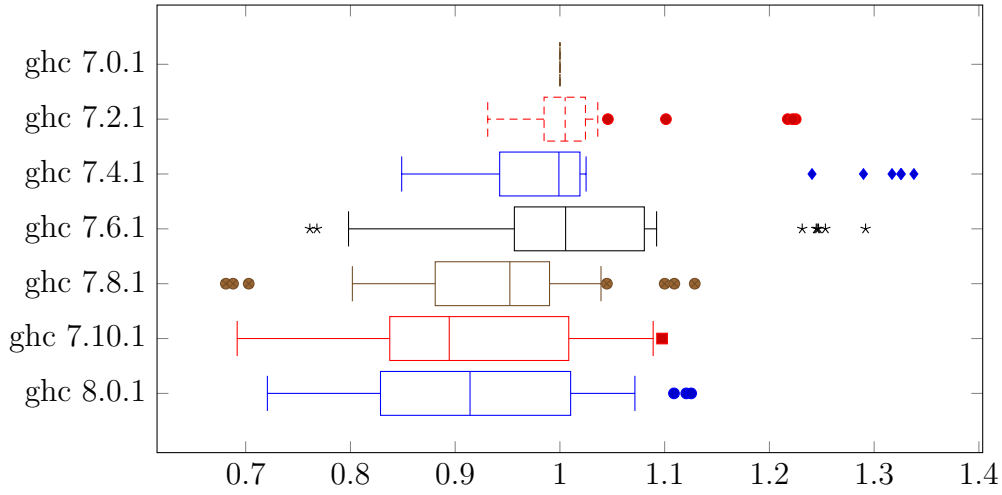


Abbildung 6.1.: Mittelwerte für jedes Programm relativ zu GHC 7.0.1

Dies zeigt sich auch in der zum Diagramm gehörenden Tabelle 6.1.

In ihr wird der geometrische Mittelwert über alle relativen Programmmittelwerte für jede GHC Version angegeben. Es gibt zwar von der GHC Version 7.0.1 zur GHC Version 8.0.1 eine Verbesserung auf 91.9%, diese ist aber deutlich kleiner als die Standardabweichung von 11.5% und damit nicht signifikant. Zwar unterscheiden sich die Programmlaufzeiten selbst in 93.1% der Fälle signifikant, aber dies ist hier nicht relevant, da die Schwankungsbreite viel zu hoch ist.

Zum selben Ergebnis kommt man bei der Betrachtung der Optimierungsstufen `-O2` und `-Odph`.

Signifikante Differenz zwischen `-O` und `-O2`

Neben den verschiedenen GHC Versionen wurden auch verschiedenen Optimierungsstufen (`-O`, `-O2` und `-Odph`) miteinander verglichen.

	Geom. Mittelwert	Geom. Std.	Mittlere Std.	X != 7.0.1
ghc 7.0.1	100.0%	0.0%	2.3%	0.0%
ghc 7.2.1	102.3%	7.6%	2.6%	65.5%
ghc 7.4.1	103.9%	14.3%	2.7%	62.1%
ghc 7.6.1	102.2%	14.3%	2.4%	79.3%
ghc 7.8.1	92.9%	11.3%	2.2%	89.7%
ghc 7.10.1	91.5%	11.5%	2.1%	89.7%
ghc 7.0.1	91.9%	11.5%	2.4%	93.1%

Tabelle 6.1.: Zum Diagramm 6.1 gehörende Tabelle. In der zweiten Spalte ist der geometrische Mittelwert über alle relativen Programmmittelwerte angegeben, in der dritten Spalte deren Standardabweichung, in der vierten Spalte die mittlere Standardabweichung der Laufzeiten aller Programme und in der fünften Spalte ist der Anteil der Programme aufgeführt bei denen sich die jeweilige Laufzeitenverteilung vergleichen mit jener beim GHC 7.0.1 nur insignifikant (bzgl. eines t-Tests) unterscheidet.

Laut der Haskell Dokumentation sollte es keinen (signifikanten) Unterschied zwischen dem Laufzeitverhalten von Code geben, welcher mit `-O` und mit `-O2` übersetzt wurde haskell.org (b):

> At the moment, `-O2` is unlikely to produce better code than `-O`.

Dies ist zumindest bei den betrachteten Programmen dennoch der Fall. Die Tabelle 6.2 zeigt für jede GHC Version den Anteil der Programme, bei denen sich die Laufzeit signifikant zwischen zwei Optimierungsstufen unterscheidet. Hierfür wurde der t-Test als Signifikanztest mit dem Signifikanzniveau 5% verwendet (siehe Abschnitt 2.2).

	-O != -O2?	-O != -Odph?	-O2 != -Odph?
ghc 7.0.1	83%	79%	21%
ghc 7.2.1	90%	83%	0%
ghc 7.4.1	76%	66%	14%
ghc 7.6.1	76%	76%	3%
ghc 7.8.1	83%	83%	3%
ghc 7.10.1	72%	69%	10%
ghc 8.0.1	79%	83%	17%
Durchschnitt	80%	77%	10%

Tabelle 6.2.: Prozentzahl der einzelnen Programme, bei denen das Laufzeitverhalten sich für eine bestimmte Implementierung signifikant unterscheidet.

Im Schnitt zeigen bei `-O` und `-O2` 80% der Programme ein unterschiedliches Lauf-

zeitverhalten, weswegen es einen signifikanten Unterschied zwischen `-O` und `-O2` gibt. Der Unterschied von `-O2` im Vergleich zu `-Odph` ist dagegen mit im Schnitt 10% vernachlässigbar.

Aufgrund dieses Ergebnisses wurde der oben zitierte Satz aus der Dokumentation der aktuell entwickelten Haskell Version, siehe Breitner (2016), entfernt.

6.3. Rustc Performanz über die Zeit

Mithilfe der im vorherigen Abschnitt vorgestellten Teilimplementierung des *Computer Language Benchmarks Game* wird im Folgenden die Performanz des Standard Rust Compilers Rustc über die Zeit evaluiert. Es werden dabei die Rustc Versionen 1.0.0[3], 1.1.0, 1.2.0, 1.3.0, 1.4.0, 1.5.0, 1.6.0, 1.7.0[4] und 1.0.8[5] verglichen.

Hierbei wurde die stärkste Optimierungsstufe gewählt. Die im Abschnitt 6.2.1 angesprochenen Probleme bei der Verwendung der Programme des *Computer Language Benchmarks Game* treffen auch hier zu.

6.3.1. Keine signifikanten Verbesserungen über die Zeit

Es scheint keine signifikanten Verbesserungen des Rust Compilers über die Zeit zu geben. Jegliche Verbesserungen liegen innerhalb der Schwankungsbreite. Das heißt nicht, dass es keine Performanzveränderungen zwischen den Compilerversionen bzgl. der einzelnen Programme gibt. Im Durchschnitt unterscheiden sich jeweils rund 50% der Laufzeiten eines Programms zwischen der ersten betrachteten Compilerversion und den anderen Versionen signifikant.

Im Folgenden wird der Mittelwert jedes Programms für jede Rustc Version relativ zur Version 1.0.0 betrachtet. Die jeweilige Verteilung für jede Version ist in Diagramm 6.2 zu finden.

Die große Schwankungsbreite bei der Performanz der verschiedenen Compiler Versionen bei den verschiedenen Programmen zeigt sich auch in der zum Diagramm gehörigen Tabelle 6.3.

[3] Die Rustc Version 1.0.0 kam am 15. Mai 2015 heraus rust lang.org (2015)
[4] Die Rustc Version 1.7.0 ist die Betaversion des Compilers vom 27. Februar 2016.
[5] Die Rustc Version 1.8.0 ist der aktuelle Entwicklungsstand des Compilers vom 27. Februar 2016.

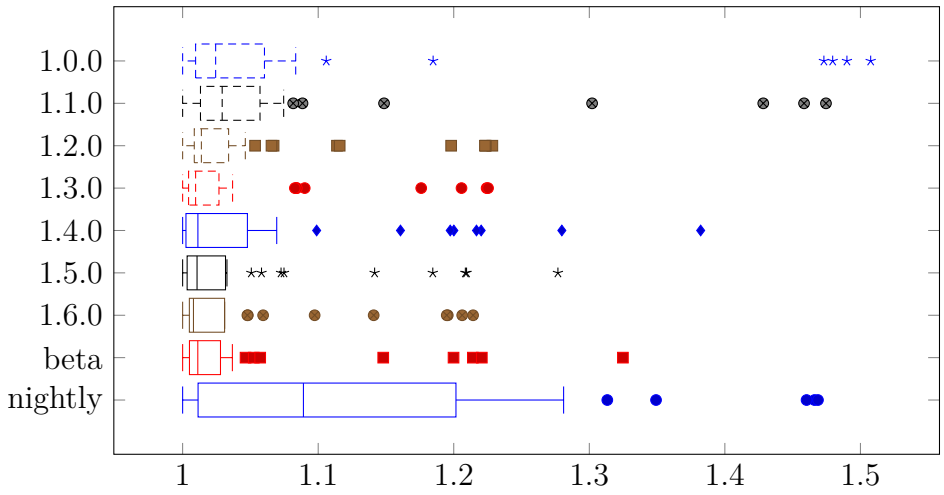


Abbildung 6.2.: Mittelwerte für jedes Programm relativ zu Rustc 1.0.0

	Geom. Mittelwert	Geom. Std.	Mittlere Std.	X != 1.0.0
1.0.0	107.0%	112.2%	2.1%	0.0%
1.1.0	106.5%	110.5%	2.2%	12.5%
1.2.0	104.1%	106.0%	2.1%	45.0%
1.3.0	103.5%	105.8%	2.0%	55.0%
1.4.0	105.1%	108.3%	2.2%	50.0%
1.5.0	103.7%	106.2%	2.1%	45.0%
1.6.0	103.5%	105.8%	2.1%	52.5%
1.0.7	103.8%	106.7%	2.0%	60.0%
1.0.8	111.8%	112.3%	1.9%	57.5%

Tabelle 6.3.: Zum Diagram 6.2 gehörende Tabelle. In der zweiten Spalte ist der geometrische Mittelwert über alle relativen Programmmittelwerte angegeben, in der dritten Spalte deren Standardabweichung, in der vierten Spalte die mittlere Standardabweichung der Laufzeiten aller Programme und in der fünften Spalte ist der Prozent der Programme aufgeführt bei denen sich die jeweilige Laufzeitenverteilung verglichen mit jener bei Version 1.0.0 nicht signifikant (bzgl. eines t-Tests) unterscheidet.

6.4. Registerallokatoren

In der Einführung wurden einige der möglichen und oft gemachten Fehler sowie Unachtsamkeiten anhand einer synthetischen und zweier realer Beispiele erläutert. In

diesem Kapitel wird nun anhand einer in Buchwald u. a. (2011) angegebenen Tabelle (Tabelle 6.4) erläutert wie *temci* konkret dabei helfen kann einige dieser Fehler zu vermeiden. Die angegebene Tabelle vergleicht zwei verschiedene Registerallokatoren von libFirm in Bezug auf den SPEC CPU2000 Integer Benchmark (im 32 Bit Modus).

Benchmark	Recoloring	PBQP	Ratio
164.gzip	345	350	101.4%
175.vpr	446	444	99.7%
176.gcc	179	179	99.8%
181.mcf	336	335	99.6%
186.crafty	233	231	99.4%
197.parser	468	467	99.7%
253.perlbmk	355	354	99.8%
254.gap	252	253	100.4%
255.vortex	417	418	100.1%
256.bzip2	374	371	99.4%
300.twolf	684	680	99.4%
Average			99.9%

Tabelle 6.4.: Comparison of execution time in seconds with recoloring and PBQP. Buchwald u. a. (2011)

In der Tabelle 6.4 und im dazugehörigen Text werden weder die Standardabweichung noch die konkret verwendete Version des Compilers angegeben.

6.4.1. Gründe für das Fehlen von statistischen Kennwerten

Auf das Fehlen der Standardabweichung angesprochen, gibt einer der Autoren der Publikation folgende mögliche Gründe an:

Zeitdruck Benchmarken und die Erfassung von Standardabweichungen kosten Zeit, von welcher es vor Abgabefristen in der Regel zu wenig gibt.

Fehlender Druck von Peer-Reviewern Es besteht kein Druck vonseiten der Peer-Reviewer, dass Standardabweichungen und ähnliche Kennwerte im Publikation explizit erwähnt werden.

Fehlender Platz Jeder Wert mehr in einer Tabelle kostet Platz, der bei Publikationen oft beschränkt ist. Damit fallen nicht wichtige (weil von Peer-Reviewern nicht

geforderte) Werte unter den Tisch, welche nicht für die gemachten Aussagen konkret wichtig sind.

Unkenntnis Vielen Informatikern fehlen auch schlicht die statistischen Grundlagen, weswegen ihnen die Relevanz der Standardabweichung nicht klar ist.

Diese allgemeinen Punkte werden von einem Mitarbeiter der Forschungsgruppe von Prof. Dr.-Ing. Henning Meyerhenke bestätigt.

6.4.2. Beschränkungen bei der Reevaluierung mit *temci*

Die exakt verwendete Version des in Buchwald u. a. (2011) betrachteten Compilers war, wie bereits erwähnt, nicht in Erfahrung zu bringen. Deshalb wurde eine aktuelle Version mit denselben Einstellungen verwendet. Da CParser und libFirm in den letzten 5 Jahren stetig weiterentwickelt wurden, führt dies unweigerlich zu Verfälschungen der resultierenden Benchmarkingergebnisse.

Die Zeitstempel der beim Benchmark verwendeten Dateien geben, laut einem der Autoren der Publikation, Grund zur Annahme, dass es sich bei der verwendeten Compilerversion um jene von Anfang September 2010 handelt. Da diese Version aber nicht baubar war[6], wurde die nächste sicher funktionierende Version von Anfang Dezember 2011 verwendet. Die Ergebnisse mit dieser Compilerversion werden am Ende des Ergebnisabschnittes angegeben.

In der Tabelle 6.4 wurden nur die Laufzeitenminima miteinander verglichen. Zwar gilt die Faustregel bzgl. der relativen Standardabweichungen (siehe Abschnitt 2.1.2) mathematisch gesehen nur beim Vergleich von Mittelwerten, die Messungen haben aber gezeigt dass die Unterschiede zwischen den Minima und den Mittelwerten im konkreten Fall vernachlässigbar sind. Deswegen wird die angesprochene Faustregel auch für die angegebene Differenz der Laufzeitenminima relativ zur Standardabweichung verwendet. Außerdem erleichtert dies den Vergleich der Ergebnisse mit jener der betrachteten Publikation.

6.4.3. Benchmarken mit *temci*

Konkret wurde die Konfigurationsdatei 3 für *temci* verwendet. Wobei mit der CParser Option `-bra-chordal-coloring pbqp` der verwendete Algorithmus für die Registerallokation geändert wird. Der Standardalgorithmus wurde in der Tabelle 6.4 als

[6]Es fehlt eine Datei Namens `configure` im Grundverzeichnis des Compilercodeverzeichnisses.

Recoloring bezeichnet.

Das verwendete Programm `spec.py` wurde von Andreas Zwinkau geschrieben und erleichtert die Arbeit mit SPEC Benchmarks.

```
- attributes:
    description: cparser with default coloring
  run_config:
    cwd: .
    run_cmd: './spec.py --no-fortran --no-cpp -i 1
     ↪ --c-compiler-args="-std=gnu89 -m32" --c-compiler cparser'
    runner: spec.py
- attributes:
    description: cparser with pbpq coloring
  run_config:
    run_cmd: './spec.py --no-fortran --no-cpp -i 1
     ↪ --c-compiler-args="-std=gnu89 -m32
     ↪ -bra-chordal-coloring=pbqp" --c-compiler cparser'
    runner: spec.py
```

Code 3: Konfigurationsdatei für *temci* um die beiden Registerallokationsalgorithmen miteinander zu vergleichen.

6.4.4. Ergebnisse

In der Tabelle 6.5 sieht man das Resultat der Benchmarks. Diese Tabelle enthält einen zur originalen Tabelle 6.4 vergleichbaren Inhalt und zusätzlich die wichtigsten statistischen Kennzahlen, welche in der publizierten Tabelle gefehlt haben.

Die Tabelle wurde (auch in ihrer LaTeX Version) von *temci* automatisch für den via *temci report* erzeugten HTML-Report produziert.

Die Tabelle ist, bis auf den Ausreißer bei `177_mesa`, auch von den Werten vergleichbar mit Tabelle 6.4. Man erkennt an den Ergebnissen des t-Tests (siehe Abschnitt 2.2) unschwer, dass bei der Mehrheit der Teilprogramme des SPEC CPU2000 Integer Benchmarks wohl kein signifikanter Unterschied zwischen den bei Registerallokationsalgorithmen besteht (konkret bei 7 von 10 Teilprogrammen, sofern man die Nullhypothese beim üblichen Wert von 5% verwirft, siehe Abschnitt 2.2).

Genauso sind die Minimadifferenzen im Vergleich zu den Standardabweichungen,

	Diff. of mins	... per first min	... per max std dev	t test
164_gzip	0.01167	0.0%	17.4%	41.2%
175_vpr	-0.04890	-0.1%	-5.5%	77.7%
176_gcc	-0.17303	-0.5%	-637.4%	0.0%
181_mcf	0.13752	0.4%	91.6%	46.7%
186_crafty	-0.09865	-0.2%	-32.7%	9.7%
197_parser	0.13510	0.1%	61.5%	11.0%
254_gap	0.16493	0.4%	129.3%	0.1%
255_vortex	-0.06716	-0.1%	-10.4%	3.5%
256_bzip2	0.04722	0.1%	7.0%	42.0%
300_twolf	0.36649	0.3%	28.9%	6.2%

Tabelle 6.5.: Von *temci* generierte Tabelle, welche in etwa der Tabelle 6.4 entspricht. Es wird hier *Recoloring* gegenüber *PBQP* verglichen. *Diff. of mins* ist hierbei die Differenz der minimalen Laufzeit von *Recoloring* und *PBQP*, *... per first min* ist diese Differenz relativ zur minimalen Laufzeit von *Recoloring*, *... per max std dev* ist die Differenz relativ zur maximalen Standardabweichung der Messungen von *Recoloring* und *PBQP* und *t test* ist die mittels t-Test berechnete Nullhypothesenwahrscheinlichkeit der beiden Messverteilungen (siehe Abschnitt 2.2). Ein *t test* Wert größer 5% bedeutet üblicherweise, dass die beiden Messverteilungen nicht signifikant unterschiedlich sind.

abgesehen von 2 Ausnahmen, relativ gering. Denn nach Heiser sollten die Differenzen relativ zu den Standardabweichungen mindestens 200% sein, um signifikant zu sein.

Damit bestätigt sich die weiter oben gemachte Aussage, dass wohl eher kein Unterschied zwischen den beiden Registerallokationsalgorithmen besteht. In Bezug auf die Tabelle 6.5 steht in der dazugehörigen Publikation Buchwald u. a. (2011):

> To evaluate the quality of our approach, we compare the best execution time out of five runs of the SPEC CPU2000 benchmark programs with the recoloring approach. The results in Table 2 show a slight improvement of 0.1% on average.

Dieser Aussage kann nicht zugestimmt werden. Denn, wie bereits erwähnt, scheint es keinen signifikanten Unterschied zugeben.[7]

Anmerkung zu `perlbmk` Das Programm `perlbmk` findet sich nicht in der Tabelle 6.5, da der CParser beim Kompilieren, unter Verwendung von *PBQP* als Registeral-

[7] Es handelt sich bei der Aussage allerdings nicht um eine zentrale Aussage der Publikation.

lokationsalgorithmus, den folgenden Fehler wirft:

> ir/be/bepbqpcoloring.c:642: libFirm panic in be_pbqp_coloring: no PBQP solution found

Der gleiche Fehler tritt auch bei der Verwendung der CParser Version von Anfang Dezember 2011 deterministisch auf.

Randomisierung der Umgebungsvariablen

Zum selben Schluss, dass es keinen signifikanten Unterschied zwischen beiden Algorithmen gibt, kommt man auch, wenn man die Größe der Umgebungsvariablen randomisiert (siehe Kapitel 4)[8]. Die Tabelle 6.6 zeigt das Ergebnis.

	Diff. of mins	... per first min	... per max std dev	t test
164_gzip	-0.02905	-0.0%	-26.9%	99.7%
175_vpr	0.15951	0.2%	60.6%	7.0%
176_gcc	-0.13744	-0.4%	-311.5%	0.0%
181_mcf	-0.04762	-0.1%	-24.3%	99.5%
186_crafty	-0.08000	-0.2%	-50.0%	20.2%
197_parser	0.17991	0.2%	56.4%	11.6%
254_gap	0.09162	0.2%	85.0%	15.9%
255_vortex	-1.64475	-2.6%	-218.3%	5.2%
256_bzip2	0.05448	0.1%	9.5%	94.2%
300_twolf	0.53047	0.5%	154.4%	0.1%

Tabelle 6.6.: Von *temci* generierte Tabelle, welche in etwa der Tabelle 6.4 entspricht. Bei den Messungen wurde, im Vergleich zur Tabelle 6.5, die Größe der Umgebungsvariablen randomisiert.

Randomisierung der Linkreihenfolge

Neben der Größe der Umgebungsvariablen kann auch die Linkreihenfolge (siehe Abschnitt 4.4 beim Bauen des SPEC CPU2000 Integer Benchmarks randomisieren. Dies geschieht, indem man die *temci* Konfigurationsdatei 6.4.3 zu 4 abändert.

[8] Hierzu übergibt man *temci* beim Aufruf zusätzlich `-env_randomize` als Parameter.

```
- attributes:
    description: cparser with default coloring
  run_config:
    cwd: .
    run_cmd: './spec.py --no-fortran --no-cpp -i 1
     ↪ --c-compiler-args="-std=gnu89 -m32" --c-compiler cparser'
    runner: spec.py
    spec.py:
      rand_conf:
        linker: True
      randomize: True
- attributes:
    description: cparser with pbpq coloring
  run_config:
    run_cmd: './spec.py --no-fortran --no-cpp -i 1
     ↪ --c-compiler-args="-std=gnu89 -m32
     ↪ -bra-chordal-coloring=pbqp" --c-compiler cparser'
    runner: spec.py
    spec.py:
      rand_conf:
        linker: True
      randomize: True
```

Code 4: Abgeänderte Konfigurationsdatei 3 bei welcher die Linkreihenfolge randomisiert wird.

Die Tabelle 6.7 zeigt das Ergebnis. Es fällt auf, dass nun nur noch bei einem Programm, `175_vpr`, die Laufzeiten signifikant unterschiedlich sind. Statt bei 3 von 10, wie beim Benchmark ohne Randomisierung.

Randomisierung des Assemblers

Den Assembler (siehe Abschnitt 4.4) kann man auf dieselbe Weise wie die Linkreihenfolge beim SPEC CPU2000 Integer Benchmark randomisieren, indem man die *temci* Konfigurationsdatei 6.4.3 zu 5 abändert:

	Diff. of mins	... per first min	... per max std dev	t test
164_gzip	-0.06318	-0.1%	-6.0%	31.3%
175_vpr	0.66569	0.8%	29.3%	3.2%
176_gcc	-0.25374	-0.7%	-132.2%	76.0%
181_mcf	-0.14689	-0.4%	-61.1%	39.3%
186_crafty	-0.28466	-0.6%	-52.8%	42.7%
197_parser	0.05354	0.1%	18.5%	96.1%
254_gap	0.12690	0.3%	36.1%	30.0%
255_vortex	0.97278	1.5%	100.1%	65.3%
256_bzip2	0.07983	0.1%	9.6%	77.8%
300_twolf	-0.19229	-0.2%	-18.4%	15.6%

Tabelle 6.7.: Von *temci* generierte Tabelle, welche in etwa der Tabelle 6.4 entspricht. Bei den Messungen wurde, im Vergleich zu Tabelle 6.5, die ausgeführten Binärdateien randomisiert, indem die Linkreihenfolgen randomisiert wurden.

```
- attributes:
    description: cparser with default coloring
  run_config:
    cwd: .
    run_cmd: './spec.py --no-fortran --no-cpp -i 1
     ↪ --c-compiler-args="-std=gnu89 -m32" --c-compiler cparser'
    runner: spec.py
    spec.py:
      rand_conf:
        bss: true
        data: true
        rodata: true
        file_structure: true
      randomize: true
- attributes:
    description: cparser with pbpq coloring
  run_config:
    run_cmd: './spec.py --no-fortran --no-cpp -i 1
     ↪ --c-compiler-args="-std=gnu89 -m32
     ↪ -bra-chordal-coloring=pbqp" --c-compiler cparser'
    runner: spec.py
    spec.py:
      rand_conf:
        bss: true
        data: true
        rodata: true
        file_structure: true
      randomize: true
```

Code 5: Abgeänderte Konfigurationsdatei 3 bei welcher der Assembler randomisiert wird.

Die Tabelle 6.8 zeigt das Ergebnis. Es fällt auf, dass es bei keinem Programm einen Unterschied der Laufzeiten zwischen beiden Registerallokationsalgorithmen gibt. Wobei die Assemblerrandomisierung bei zwei Programmen reproduzierbar fehlschlug und deswegen hier nicht aufgeführt sind.

	Diff. of mins	... per first min	... per max std dev	t test
175_vpr	-0.27572	-0.3%	-8.7%	33.5%
181_mcf	0.03647	0.1%	17.2%	96.5%
186_crafty	0.06896	0.2%	47.9%	6.8%
197_parser	0.05334	0.1%	21.0%	70.2%
254_gap	0.11628	0.3%	26.9%	88.0%
255_vortex	0.89163	1.4%	156.7%	6.0%
256_bzip2	-0.09435	-0.1%	-16.3%	98.6%
300_twolf	0.18116	0.2%	30.0%	26.7%

Tabelle 6.8.: Von *temci* generierte Tabelle, welche in etwa der Tabelle 6.4 entspricht. Bei den Messungen wurde, im Vergleich zu Tabelle 6.5, die ausgeführten Binärdateien randomisiert, indem der Assembler randomisiert wurde.

6.4.5. CParser Version von Mitte Dezember 2011

Zu einem vergleichbaren Ergebnis kommt man auch, wenn man die CParser und libFirm Version von Mitte Dezember 2011 verwendet und dabei den Assembler, wie auch die Linkreihenfolge beim Bauen randomisiert. Der einzige wesentliche Unterschied ist, dass es bei zwei Programmen einen signifikanten Unterschied gibt.

Schlussfolgerung

Es scheint mit diesen Ergebnissen, als ob, der Effekt des Wechsels der Registerallokationsalgorithmen insignifikant im Vergleich zu einfachen Änderungen der Linkreihenfolge, der Umgebungsvariablen oder des Assemblers ist.

6.5. CParser und libFirm Performanz über die Zeit

In diesem Abschnitt wird die Performanz zweier zeitlich rund 4 Jahre auseinander liegenden Versionen des CParsers (und damit auch der libFirm) miteinander vergli-

	Diff. of mins	... per first min	... per max std dev	t test
164_gzip	-0.02100	-0.0%	-16.4%	30.2%
175_vpr	2.32243	2.9%	265.9%	22.1%
176_gcc	-0.29086	-0.8%	-524.8%	0.0%
181_mcf	-0.01656	-0.0%	-9.6%	23.2%
186_crafty	-0.13725	-0.3%	-99.2%	29.4%
197_parser	0.03230	0.0%	16.4%	98.9%
254_gap	0.15946	0.4%	103.9%	4.4%
255_vortex	1.72220	2.6%	144.0%	97.9%
256_bzip2	-0.26892	-0.4%	-38.2%	26.1%
300_twolf	0.47620	0.4%	77.7%	21.6%

Tabelle 6.9.: Von *temci* generierte Tabelle, welche in etwa der Tabelle 6.4 entspricht. Bei den Messungen wurde, im Vergleich zu Tabelle 6.5, die ausgeführten Binärdateien randomisiert, indem die Linkreihenfolgen und der Assembler randomisiert wurden. Dabei wurde als Compilerversion jene von Mitte Dezember 2011 (statt Mitte Januar 2016) verwendet wurde.

chen: Die Version von Mitte Dezember 2011 mit der Version von Mitte Januar 2016. Der Vergleich fand anhand des SPEC CPU2000 Benchmarks statt. Beim Benchmarken wurden alle Möglichkeiten zur Randomisierung genutzt, die *temci* bietet: Es wurden der Assembler und die Linkreihenfolge beim Bauen, die Größe der Umgebungsvariablen und die Benchmarkingreihenfolge randomisiert (siehe Abschnitt 6.4).

Es zeigt sich, siehe Tabelle 6.10, dass die Verbesserungen, die in den rund 4 Jahren stattfanden, nicht signifikant sind und im Rauschen der Randomisierungen untergehen. Nur bei 2 Programmen (`164_gzip` und `300_twolf`) kam es zu signifikanten Unterschieden der Laufzeitenverteilung bzgl. des t-Tests. Diese Unterschiede liegen aber deutlich unter der jeweiligen doppelten Standardabweichung.

6.6. CParser und GCC

Im Folgenden wird zum einen der CParser mit dem GCC mit den Optimierungsstufen `-O2` und `-O3` verglichen. Hierfür wurden jeweils 10 Messungen mithilfe des SPEC CPU2000 Benchmarks (siehe vorheriger Abschnitt) durchgeführt, wobei jeweils die Linkreihenfolge und die Umgebungsvariablen randomisiert wurden (siehe Abschnitt 4).

	Diff. of means	... per first mean	... per max std dev	t test
164_gzip	0.06824	0.1%	92.6%	1.9%
175_vpr	-0.02596	-0.0%	-8.2%	84.0%
176_gcc	0.02014	0.1%	71.2%	14.6%
177_mesa	2.06495	0.8%	33.9%	32.8%
179_art	-0.15886	-0.3%	-46.4%	25.1%
181_mcf	0.03253	0.1%	22.3%	60.2%
183_equake	0.11952	0.2%	3.9%	93.5%
186_crafty	0.08200	0.2%	39.1%	33.3%
188_ammp	-0.26627	-0.2%	-21.0%	54.5%
197_parser	-0.00453	-0.0%	-2.6%	95.0%
253_perlbmk	0.08652	0.1%	22.7%	54.6%
254_gap	0.02205	0.1%	20.0%	65.8%
255_vortex	0.02506	0.0%	3.2%	94.1%
256_bzip2	-0.07021	-0.1%	-11.0%	80.2%
300_twolf	-0.35646	-0.3%	-151.7%	0.2%

Tabelle 6.10.: Der CParser Version Mitte Dezember 2011 verglichen mit der Version Mitte Januar 2016 bei der Optimierungsstufe `-O2`. *Diff. of means* ist hierbei die Differenz der durchschnittlichen Laufzeiten der beiden Versionen, *... per first mean* ist diese Differenz relativ zur durchschnittlichen Laufzeit der ersten Version, *... per max std dev* ist die Differenz relativ zur maximalen Standardabweichung der Messungen der beiden Versionen und *t test* ist die mittels t-Test berechnete Nullhypothesenwahrscheinlichkeit der beiden Messverteilungen (siehe Abschnitt 2.2). Ein *t test* Wert größer 5% bedeutet üblicherweise, dass die beiden Messverteilungen nicht signifikant unterschiedlich sind.

Die gezeigten Tabellen wurden automatisch von *temci* generiert.

6.6.1. Vergleich von CParser und GCC

Die Tabelle 6.11 vergleicht beide Compiler bei der Optimierungsstufe `-O2`. Es fällt auf, dass der CParser bei 5 von 14 Programmen signifikant schneller und bei 8 von 14 Programmen signifikant langsamer als der GCC ist. Nur beim Programm `181_mcf` sind beide Compiler vergleichbar schnell (mit der Signifikanzgrenze 5%). Der GCC ist im Mittel über alle Programme rund 1.3% schneller als der CParser. Da die Standardabweichung dieses Mittelwerts mit 19% deutlich höher als die Differenz ist, ist diese Verbesserung nicht signifikant.

	Diff. of means	... per first mean	... per max std dev	t test
164_gzip	3.00186	3.1%	1667.8%	0.0%
175_vpr	-8.75769	-12.0%	-2971.8%	0.0%
176_gcc	-0.48395	-1.4%	-982.8%	0.0%
177_mesa	28.01540	13.4%	1044.5%	0.0%
179_art	10.49756	16.2%	692.6%	0.0%
181_mcf	-0.16613	-0.4%	-95.1%	5.1%
183_equake	-8.75428	-17.4%	-856.7%	0.0%
186_crafty	6.95928	13.6%	4936.9%	0.0%
188_ammp	-70.17884	-70.0%	-5520.8%	0.0%
197_parser	6.84918	6.8%	4290.8%	0.0%
253_perlbmk	-7.93010	-12.9%	-3006.8%	0.0%
255_vortex	4.72354	6.7%	426.5%	0.0%
256_bzip2	5.43985	7.5%	794.2%	0.0%
300_twolf	4.65952	4.0%	2118.8%	0.0%

Tabelle 6.11.: CParser verglichen mit GCC bei der Optimierungsstufe `-O2`. *Diff. of means* ist hierbei die Differenz der durchschnittlichen Laufzeiten von CParser und GCC, *... per first mean* ist diese Differenz relativ zur durchschnittlichen Laufzeit von CParser, *... per max std dev* ist die Differenz relativ zur maximalen Standardabweichung der Messungen von CParser und GCC und *t test* ist die mittels t-Test berechnete Nullhypothesenwahrscheinlichkeit der beiden Messverteilungen (siehe Abschnitt 2.2). Ein *t test* Wert größer 5% bedeutet üblicherweise, dass die beiden Messverteilungen nicht signifikant unterschiedlich sind.

Der CParser und der GCC unterscheiden sich bei der Optimierungsstufe `-O2` im Mittel nur insignifikant.

Zu einem ähnlichen Ergebnis kommt man beim Vergleich beider Compiler bei der Optimierungsstufe `-O3` in der Tabelle 6.12. Der CParser ist hierbei bei 6 von 14 Programmen signifikant schneller und bei 5 Programmen signifikant langsamer als der GCC. Wieder sind beide Compiler nur beim Programm `181_mcf` vergleichbar schnell. Im Mittel ist der GCC zwar mit rund 7.5% schneller als der CParser, dies ist aber vernachlässigbar gegenüber der Standardabweichung von 21%.

Es gibt damit im Mittel keinen signifikanten Unterschied zwischen dem CParser und dem GCC bei den Optimierungsstufen `-O2` und `-O3` bzgl. der Laufzeiten der mit ihnen übersetzten Programmen im SPEC CPU2000 Benchmark.

	Diff. of means	... per first mean	... per max std dev	t test
164_gzip	1.03182	1.1%	745.1%	0.0%
175_vpr	-9.85565	-13.8%	-1609.2%	0.0%
176_gcc	-0.78407	-2.2%	-172.3%	0.0%
177_mesa	-25.37822	-16.5%	-2795.6%	0.0%
179_art	8.30816	13.1%	480.7%	0.0%
181_mcf	-0.07629	-0.2%	-28.7%	44.6%
183_equake	-21.83763	-56.9%	-2157.4%	0.0%
186_crafty	7.64929	14.7%	4224.4%	0.0%
188_ammp	-68.21435	-66.8%	-5221.5%	0.0%
197_parser	-3.37610	-3.7%	-1070.0%	0.0%
253_perlbmk	-8.35554	-13.7%	-2273.9%	0.0%
255_vortex	-1.72015	-2.7%	-100.6%	1.6%
256_bzip2	3.69892	5.2%	570.2%	0.0%
300_twolf	1.44600	1.3%	554.6%	0.0%

Tabelle 6.12.: CParser verglichen mit GCC bei der Optimierungsstufe `-O3`.

6.6.2. Vergleich von `-O2` und `-O3`

Im Folgenden werden beim CParser und beim GCC jeweils die Optimierungsstufen `-O2` und `-O3` verglichen.

In der Tabelle 6.13 werden die beiden Optimierungsstufen beim CParser verglichen. Hierbei fällt auf, dass es nur bei den Programmen `175_vpr` und `183_equake` einen signifikanten Unterschied bzgl. des t-Tests gibt. Bei allen anderen Programmen ist kein signifikanter Unterschied erkennbar.

In der Tabelle 6.14 ist derselbe Vergleich für den GCC. Hier zeigt sich, dass es, außer bei drei Programmen, einen jeweils signifikanten Unterschied zwischen `-O2` und `-O3` gibt. `-O3` ist bei 9 von 14 Programmen deutlich schneller und bei 2 Programmen deutlich langsamer als `-O3`.

	Diff. of means	... per first mean	... per max std dev	t test
164_gzip	0.00295	0.0%	6.2%	89.4%
175_vpr	0.27749	0.3%	94.2%	2.0%
176_gcc	-0.01162	-0.0%	-42.3%	35.2%
177_mesa	0.95330	0.5%	35.5%	32.6%
179_art	-0.68383	-1.3%	-39.6%	26.0%
181_mcf	-0.02807	-0.1%	-17.4%	68.4%
183_equake	-1.22906	-2.1%	-120.3%	2.0%
186_crafty	-0.05106	-0.1%	-28.2%	51.3%
188_ammp	0.14041	0.1%	10.7%	82.0%
197_parser	-0.02997	-0.0%	-9.5%	80.2%
253_perlbmk	0.02800	0.0%	13.9%	75.2%
254_gap	-0.01200	-0.0%	-9.4%	84.1%
255_vortex	-0.15809	-0.2%	-14.3%	74.7%
256_bzip2	-0.27397	-0.4%	-42.2%	36.3%
300_twolf	-0.14791	-0.1%	-56.7%	14.9%

Tabelle 6.13.: Vergleich der beiden Optimierungsstufen `-O2` und `-O3` beim CParser.

	Diff. of means	... per first mean	... per max std dev	t test
164_gzip	1.97299	2.1%	1096.2%	0.0%
175_vpr	1.37544	1.9%	224.6%	0.0%
176_gcc	0.28850	0.8%	63.4%	7.5%
177_mesa	54.34692	26.1%	4731.8%	0.0%
179_art	1.50557	2.3%	99.3%	1.2%
181_mcf	-0.11791	-0.3%	-44.3%	28.1%
183_equake	11.85429	23.6%	6608.5%	0.0%
186_crafty	-0.74106	-1.4%	-517.4%	0.0%
188_ammp	-1.82408	-1.8%	-197.1%	0.0%
197_parser	10.19531	10.2%	7847.6%	0.0%
253_perlbmk	0.45344	0.7%	123.4%	0.8%
255_vortex	6.28560	8.9%	367.8%	0.0%
256_bzip2	1.46696	2.0%	214.2%	0.0%
300_twolf	3.06561	2.7%	1394.0%	0.0%

Tabelle 6.14.: Vergleich der beiden Optimierungsstufen `-O2` und `-O3` beim CParser.

7. Fazit und Ausblick

In den Monaten in denen ich an *temci* gearbeitet habe ist aus einer groben Idee ein benutzbares Benchmarking Werkzeug geworden, welches Fähigkeiten hat, die es sonst in keinem anderen Werkzeug zusammengefasst gibt. Es trägt hoffentlich dazu bei, dass in Zukunft weniger grobe Fehler (siehe Einführung) beim Benchmarken rein aus Zeit- oder Bequemlichkeitsgründen gemacht werden. Als Fazit kann ich persönlich sagen, dass mir die Arbeit an *temci* viel Spaß gemacht hat und ich dabei viel über das Benchmarken gelernt habe. Ich hätte noch gern viele weitere Mechanismen und Fähigkeiten in *temci* integriert, aber irgendwann muss man auch einen Schlussstrich ziehen. Im verbleibenden Teil dieses Abschnitts sind noch ein paar Dinge beschrieben, die man in *temci* integrieren oder verbessern könnte.

7.1. Assemblerrandomisierung

Wie im Abschnitt 4.4 erläutert funktioniert die Assemblerrandomisierung in *temci* zurzeit mit relativ einfachen Heuristiken. Diese Heuristiken müssen für jedes neue Assemblerlayout neu entwickelt werden und sind noch fehleranfällig. Es wäre interessant zu sehen, ob man den Assembler Code in einer Weise verarbeiten könnte, dass man Assembler aus vielen Quellen randomisieren kann. Dies würde voraussetzen, dass man aus dem Assembler Code nicht nur oberflächlich betrachtet, sondern auch semantisch versteht.

Eine weitere wichtige Verbesserung wäre in diesem Bereich die Implementierung in einer performanteren Sprache, wie zum Beispiel C++ oder Rust. Da der Code zur Assemblerrandomisierung bei Bauen von Programmen oft aufgerufen wird ist Performanz in diesem Bereich wichtig.

7.2. Integration mit Versuchung

Es gibt mit *Versuchung* und *DataRef* Dietrich u. Lohmann (2015) zwei Werkzeuge, die es in Kombination ermöglichen die Resultate von Benchmarkingexperimenten

direkt in LaTeX Dokumente einzubinden und die Benchmarkexperimente selbst zu automatisieren. Sofern die Quelldateien zu einer Publikation veröffentlicht werden ist es für andere damit relativ einfach möglich die Ergebnisse zu verifizieren und die Experimente zu wiederholen.

Versuchung ist hierbei das in Python geschriebene Werkzeug zur Beschreibung und Durchführung von Experimenten und *DataRef* ermöglicht die Integration in LaTeX. Zurzeit verwendet Versuchung noch das `time` Werkzeug zur Zeitmessung. Es wäre schön, wenn man stattdessen auch *temci* verwenden könnte, samt allen Dingen die *temci* ausmachen.

7.3. Handbuch

Zurzeit gibt es für *temci* noch kein Handbuch in welchem alle Funktionen von *temci* und deren Benutzung ausführlich erklärt wird. Dies schränkt *temci* hinsichtlich der Benutzung durch andere Leute ein obwohl die Code Dokumentation ausführlich ist. Es ist geplant ein solches Handbuch in Anschluss auf die Bachelorarbeit zu schreiben und in Kombination mit Vorträgen bei Konferenzen mehr Leute auf *temci* aufmerksam zu machen.

A. Mathematische Ergänzungen

In diesem Kapitel finden sich mathematische Ausführungen deren Ergebnisse in den vorherigen Kapiteln verwendet wurden.

A.1. Herleitungen

Herleitung 1 *Die Standardabweichung (und damit auch die Varianz) des geometrischen Mittelwertes $\bar{x}_{geom}$ für die Menge $x = \{x_1, \ldots, x_n\}$ ist*

$$\sigma_{\bar{x}_{geom}} = \exp\left(\sqrt{\frac{1}{n_x} \cdot \sum_{i=1}^{n_x} \left[\ln\left(\frac{x_i}{\bar{x}_{geom}}\right)\right]^2}\right)$$

Diese Formel kann wie folgt mithilfe der arithmetischen Standardabweichung hergeleitet werden:

$$\bar{x}_{geom} = \sqrt[n]{\prod_{i=1}^{n} x_i} \qquad \Rightarrow \qquad \ln \bar{x}_{geom} = \frac{1}{n_x} \sum_{i=1}^{n} \underbrace{\ln x_i}_{=:z_i} = \bar{z}_{arith}$$

$\bar{z}_{arith}$ ist hierbei das arithmetische Mittel der logarithmierten x_i. Mit Nauta (2010) und der Standardabweichung für das arithmetische Mittel folgt nun

$$\sigma_{\bar{x}_{geom}} = \exp \sigma_{\bar{z}_{arith}} = \exp\left(\sqrt{\frac{1}{n_x} \sum_{i=1}^{n} (z_i - \bar{z}_{arith})^2}\right) = \exp\left(\sqrt{\frac{1}{n} \sum_{i=1}^{n} \left[\ln\left(\frac{x_i}{\bar{x}_{geom}}\right)\right]^2}\right)$$

■

A.2. Motivation der Faustregel

Im Abschnitt 2.1.2 wurde die Faustregel angesprochen, dass die Differenz der Mittelwerte[1] zweier Stichproben als nicht signifikant zu verwerfen ist, falls diese kleiner als

[1] Der Mittelwert und der Erwartungswert einer Verteilung werden hier synonym verwendet.

die zweifache (maximale) Standardabweichung ist. Diese Faustregel soll im Folgenden statistisch motiviert werden.

Die beiden betrachteten Verteilungen seien X und Y mit den Stichproben $\mathbb{X} \subset X$ und $\mathbb{Y} \subset Y$. Die Verteilungsmittelwerte $\bar{x}$ und $\bar{y}$. Es gilt weiterhin, dass der Mittelwert der ersten Verteilung $\bar{x}$ kleiner als jener der zweiten Verteilung $\bar{y}$ ist.

Die Frage ist nun wie hoch die Wahrscheinlichkeit ist, dass bei den Stichprobenmittelwerten $\bar{\mathbb{x}} \geq \bar{\mathbb{y}}$ gilt, abhängig von der Differenz der Stichprobenmittelwerte relativ zur maximalen Standardabweichung (im Folgenden als k bezeichnet). Wie hoch also die Wahrscheinlichkeit ist, dass man durch die Stichproben zu einem falschen Schluss bzgl. des Verhältnisses des Verteilungsmittelwertes kommt. Diese Wahrscheinlichkeit wird als $P[\bar{\mathbb{x}} \geq \bar{\mathbb{y}}]$ bezeichnet.

Zur Vereinfachung werden folgende Annahmen über die Verteilungen und Stichproben gemacht, weswegen nur eine grobe Abschätzung der genannten Wahrscheinlich möglich ist: Beide betrachteten Stichproben haben die gleiche Größe n und haben die gleiche positive Standardabweichung σ (die maximale Standardabweichung beider Stichproben) welche auch der jeweiligen Verteilungsstandardabweichung entspricht. Die Verteilungen selbst seien Normalverteilungen womit auch die Verteilung der Stichprobenmittelwerte selbst Normalverteilungen mit dem selben Erwartungswert und der Standardabweichung $\sigma_m = \frac{\sigma}{\sqrt{n}}$ sind[2]. Zur weiteren Vereinfachung seien die Werte der beiden Verteilungen so transponiert, dass $\bar{x} = 0$ und damit auch (wie oben erwähnt) $\bar{y} = k \cdot \sigma_m$.

Nun kann man eine Form für $P_k[\bar{\mathbb{x}} \geq \bar{\mathbb{y}}]$ wie folgt herleiten (getreu den Rechenregeln aus Henze (2013)):

[2]Dies ist einfach aus dem 1. Satz von Gramer (1936) herleitbar.

$$
\begin{aligned}
P[\bar{\mathbb{x}} \geq \bar{\mathbb{y}}] &= \int_{-\infty}^{\infty} P[\bar{\mathbb{x}} = \alpha] \cdot P[\bar{\mathbb{y}} \leq \alpha] \mathrm{d}\alpha \\
&\overset{\text{Def.}}{=} \int_{-\infty}^{\infty} P[\bar{\mathbb{x}} = \alpha] \cdot \frac{1}{2} \left(1 + \operatorname{erf} \left(\frac{\alpha - \bar{y}}{\sqrt{2\sigma_m^2}} \right) \right) \\
&= \frac{1}{2} \left(\underbrace{\int_{-\infty}^{\infty} P[\bar{\mathbb{x}} = \alpha] \mathrm{d}\alpha}_{=1} + \int_{-\infty}^{\infty} P[\bar{\mathbb{x}} = \alpha] \cdot \operatorname{erf} \left(\frac{\alpha - \overbrace{\bar{y}}^{=k \cdot \sigma_m}}{\sqrt{2\sigma_m^2}} \right) \mathrm{d}\alpha \right) \\
&\overset{\text{Def.}}{=} \frac{1}{2} \left(1 + \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma_m^2}} e^{-\frac{\overbrace{(\alpha - \bar{x})^2}^{=x^2}}{2\sigma_m^2}} \cdot \operatorname{erf} \left(\frac{\alpha - k \cdot \sigma_m}{\sqrt{2\sigma_m^2}} \right) \mathrm{d}\alpha \right) \\
&= \frac{1}{2} \left(1 + \frac{1}{\sqrt{2\pi\sigma_m^2}} \int_{-\infty}^{\infty} e^{-\frac{x^2}{2\sigma_m^2}} \cdot \operatorname{erf} \left(\frac{\alpha - k \cdot \sigma_m}{\sqrt{2\sigma_m^2}} \right) \mathrm{d}\alpha \right)
\end{aligned}
$$

Das Integral ist nur schwer vereinfach- und auflösbar. Es zeigt sich aber numerisch, dass der Wert von $P[\bar{\mathbb{x}} \geq \bar{\mathbb{y}}]$ scheinbar nicht von dem Wert der Standardabweichung σ_m sondern nur von der Mittelwertedifferenz relativ zur Standardabweichung (k) abhängt[3]. Die Werte von $P[\bar{\mathbb{x}} \geq \bar{\mathbb{y}}]$ sind in der Abbildung A.1 visualisiert.

Bei $k = 2$ gilt $P[\bar{\mathbb{x}} \geq \bar{\mathbb{y}}] \approx 7.86496\%$[4]. Damit ist die Wahrscheinlichkeit, dass die Ordnung der beiden betrachteten Stichprobenmittelwerte nicht der Ordnung der jeweiligen Verteilungsmittelwerte entspricht, vernachlässigbar klein.

[3] Es wurde hierfür numerisch der maximale absolute Wert der Ableitung des Integrals über σ_m (im Intervall $(0, 10)$) bei $k = 2$ bestimmt. Der berechnete Wert liegt im Bereich der Berechnungsungenauigkeit des verwendeten Verfahrens.

[4] Für $k = 1$ gilt $P[\bar{\mathbb{x}} \geq \bar{\mathbb{y}}] \approx 23.97501\%$

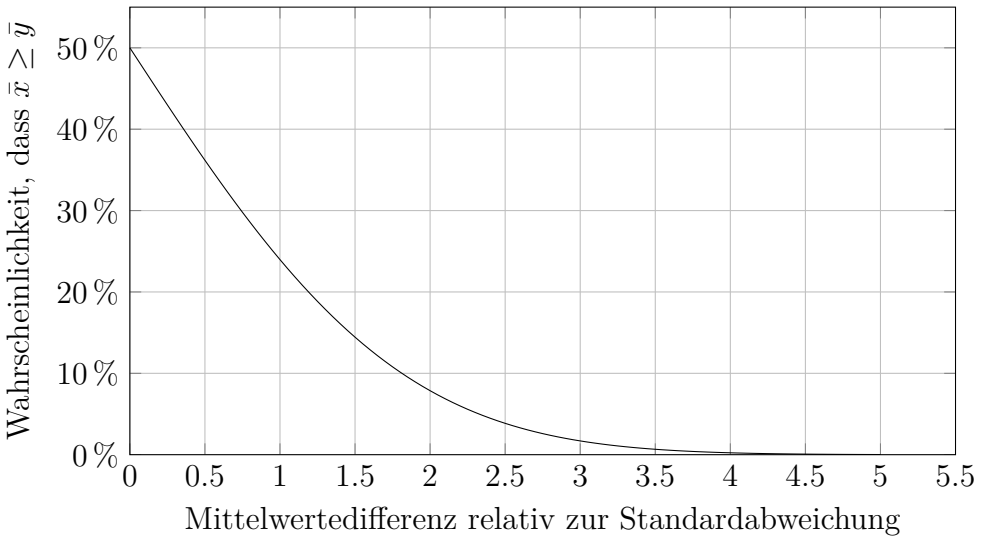


Abbildung A.1.: Die Werte von $P[\bar{\mathbb{x}} \geq \bar{\mathbb{y}}]$ in Abhängigkeit von k

B. Metaanalyse von Publikationen

Im Folgenden wurden alle Publikationen von 4 verschiedenen Lehrstühlen der Fakultät für Informatik des KITs, daraufhin untersucht ob bei Laufzeitmessungen die Standardabweichung oder ein anderes Streumaß angegeben wurde. Die Untersuchung hat zum Zweck zu zeigen, dass das Weglassen einen Streumaßes bei Laufzeitangaben in der Informatik weit verbreitet ist.

Lehrstühle Die für diese Untersuchung ausgewählten Lehrstühle sind der Lehrstuhl für Betriebssysteme von Prof. Dr.-Ing. Frank Bellosa, der Lehrstuhl für Computergrafik von Prof. Dr.-Ing. Carsten Dachsbacher, die Forschungsgruppe Paralleles Rechnen von Prof. Dr.-Ing. Henning Meyerhenke und der Lehrstuhl Programmierparadigmen von Prof. Dr.-Ing. Gregor Snelting. Diese wurden ausgewählt, weil in vieler ihrer Publikationen die Ergebnisse von Laufzeitmessungen angegeben werden.

Laufzeitmessung Als Laufzeitmessung wird im Folgenden alles betrachtet, was auf einer realen Hardware gemessen und entweder als konkrete Zeitspanne oder in der Form $\frac{x}{Zeit}$, angegeben wurde. Die letztere Form wird betrachtet, da sie die vorherrschende Angabeform bei allen Lehrstühlen außer dem Lehrstuhl von Prof. Dr.-Ing. Gregor Snelting ist.

Publikation Als Publikation eines Lehrstuhls wurde jede Veröffentlichung angesehen, die entweder auf der Veröffentlichungsseite der Lehrstuhl-Website oder der Website des leitenden Professors geführt wird. Die dort angegebenen Diplomarbeiten, populärwissenschaftlichen Publikationen und Vorlesungs- und Vortragsnotizen wurden ignoriert. Das Gleiche gilt für alle Publikationen auf die kein Zugriff möglich war oder in denen keine Laufzeitangabe zu finden war. Dabei wurde in Kauf genommen, dass auch Publikationen des leitenden Professors aus der Zeit, an welcher er noch nicht seinen Lehrstuhl leitete, betrachtet wurden.

Untersuchungsmethode In jeder betrachteten Publikation wurden zuerst Laufzeitangaben gesucht. Aufgrund der großen Anzahl von zu untersuchenden Texten wurde

davon ausgegangen, dass relevante Laufzeitangaben entweder in Form einer Tabelle oder eines Balkendiagramms angegeben wurden. Laufzeitangaben im Fließtext oder in Linien- und Punktdiagrammen wurden ignoriert. Bei jeder der gefundenen Laufzeitangaben wurde innerhalb des dazugehörigen Abschnitt und der Tabelle bzw. dem Diagramm nach Aussagen zur Streuung gesucht. Als Streumaßangabe wurden Boxplot (siehe Abschnitt 2.1.3), Standardabweichung (und vergleichbare Maße), Konfidenzintervalle und selbst eine Aussage im Stile von „Die Standardabweichung ist im Vergleich zu X gering, deswegen wird sie ignoriert“ akzeptiert. Es kam nur ein einziges Mal vor, dass in einer Publikation gleichzeitig Laufzeitangaben mit und ohne Streumaß angegeben wurden. Außerdem wurde noch untersucht, ob mögliche systembedingte Verzerrungen (siehe Kapitel 4) der jeweiligen Laufzeitangaben in der Publikation betrachtet wurden.

B.1. Beschränkungen

Es können, einfach aufgrund der schieren Menge an betrachteten Texten und der darin behandelten Forschungsgebiete, Fehler unterlaufen sein. Der einzige Zweck dieser Metaanalyse ist zu zeigen, dass das Weglassen von Streumaßen in Publikationen in der Informatik weit verbreitet ist. Es ist aber mitnichten so, dass durch das Weglassen des Streumaßes die aus den Laufzeitangaben gezogenen Schlüsse fehlerhaft sein müssen. Denn in vielen Fällen ist die Messwertestreuung entweder vernachlässigbar, oder die Laufzeitangaben wurden nicht zur Stützung einer Hauptaussage der jeweiligen Publikation verwendet.

Ersteres ist zum Beispiel bei vielen Publikationen des Lehrstuhls von Prof. Dr.-Ing. Carsten Dachsbacher der Fall. Auf das Fehlen der Streumaße bei Laufzeitangaben angesprochen, antwortete er:

> Unsere Benchmarks werden immer so durchgeführt, dass das System weitestgehend frei von anderen Aufgaben ist. Bei FPS-Messungen[1] (im Millisekundenbereich) werden viele Messungen mit denselben Eingabedaten gemittelt, da in der Praxis dieser Wert relevant ist und es in der Regel keine durch die Algorithmen verursachte Streuung gibt. Bei variierenden Eingabedaten werden entweder in deren Abhängigkeit mehrere Messungen durchgeführt oder aggregierte Werte angegeben (hier ist nur die Laufzeitcharakteristik oder Vergleiche zu anderen Methoden wichtig).
>
> Bei langen Rechenzeiten (mehrere Sekunden, bis hin zu Stunden) gehen wir davon aus, dass „Rauschen“ keine Rolle spielt. Würde man ein Ver-

[1] Messungen der Frames per Second (der Bildwiederholrate), die im Bereich der Computergrafik wichtig sind.

fahren das 1h rechnet 10-mal laufen lassen, so sind die Unterschiede in der Regel vernachlässigbar.

Wenn unsere Verfahren von externen Faktoren (zum Beispiel CPU/GPU-Temperatur und Drosselung) oder anderen Faktoren abhängig sind, dann werden natürlich Abweichungen untersucht.

Bei der Betrachtung spezieller Anwendungen oder Anforderungen kann von diesen Vorgehensweisen abgewichen werden, sofern dies erforderlich oder sinnvoll ist.

Auch deswegen sind die folgenden Ergebnisse nur als Diskussionsgrundlage gedacht, mithilfe welcher über die wissenschaftliche Methodik diskutiert werden sollte. Dieser Meinung sind auch die befragten Mitarbeitern der Lehrstühle von Prof. Dr.-Ing. Gregor Snelting und Prof. Dr.-Ing. Henning Meyerhenke. Die Ergebnisse sind aber nicht in der vereinfachenden Form „Lehrstuhl X hat eine schlechte wissenschaftliche Methodik“ zu interpretieren.

B.2. Ergebnisse

In 90% der 148 untersuchten Publikationen wurde bei Laufzeitangaben weder ein Streumaß angegeben, noch eine Aussage über die jeweilige Messwertestreuung gemacht. In einer Publikation wurde das Streumaß nur in Teilen angegeben. Die Verteilung für jeden Lehrstuhl wird in Abbildung B.1 gezeigt. Mögliche systembedingte Verzerrungen (siehe Kapitel 4) wurden in keiner Publikation betrachtet (zu einem vergleichbaren Ergebnis bei anderen Publikationen kommt Diwan u. a. (2009)).

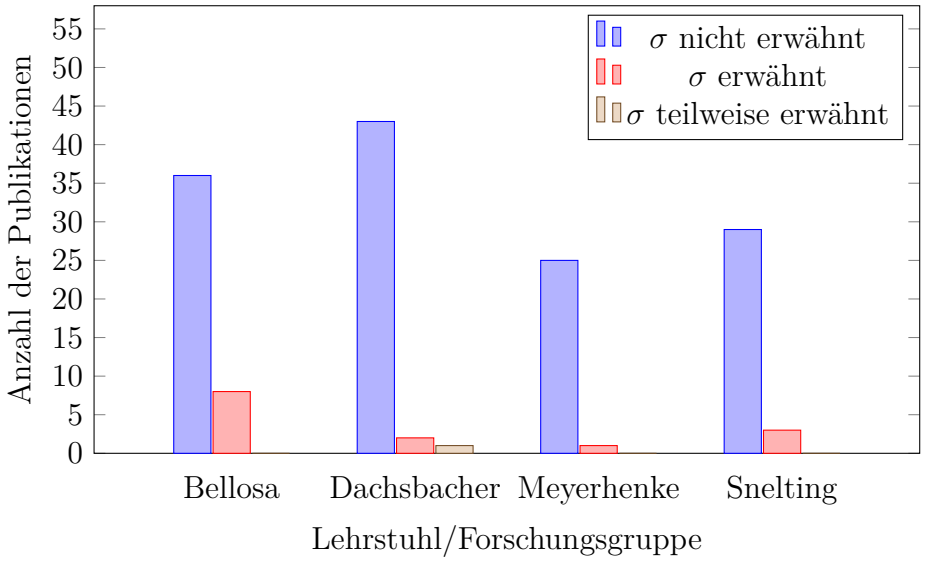


Abbildung B.1.: Anzahl der Publikationen pro Lehrstuhl bei denen bei Laufzeitangaben ein Streumaß erwähnt, nicht oder nur teilweise erwähnt wurde. Insgesamt wurden bei nur 15 von 148 Publikationen Streumaße zumindest teilweise erwähnt.

Im Folgenden wird die Verteilung der Publikationen über die Zeit für jeden der betrachteten Lehrstühle angegeben.

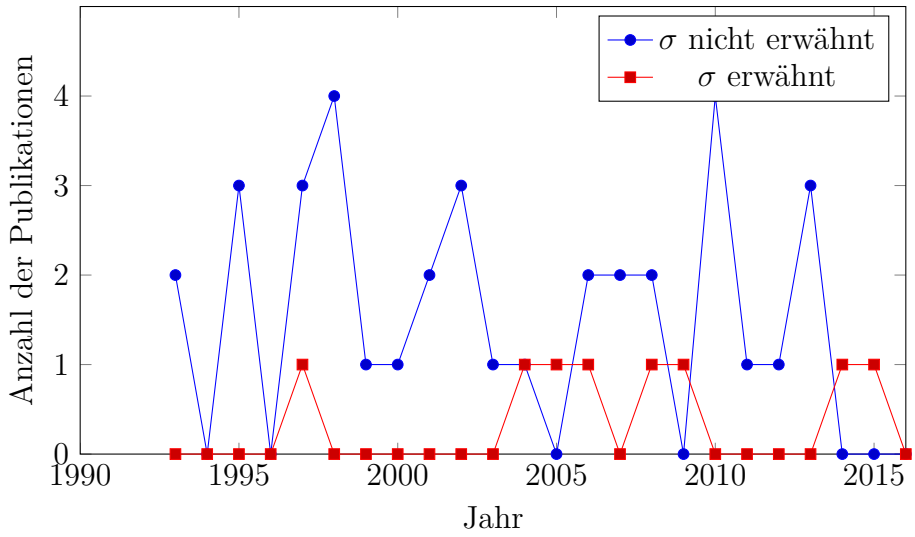


Abbildung B.2.: Anzahl der Publikationen des Lehrstuhls von Prof. Dr.-Ing. Frank Bellosa bei denen bei Laufzeitangaben ein Streumaß erwähnt, nicht oder nur teilweise erwähnt wurde. Insgesamt wurden bei nur 8 von 44 Publikationen Streumaße zumindest teilweise erwähnt.

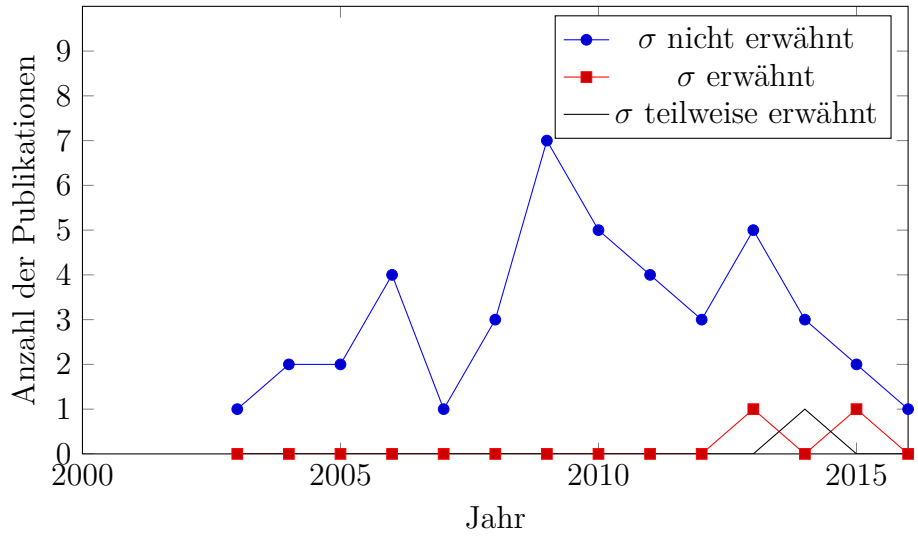


Abbildung B.3.: Anzahl der Publikationen des Lehrstuhls von Prof. Dr.-Ing. Carsten Dachsbacher bei denen bei Laufzeitangaben ein Streumaß erwähnt, nicht oder nur teilweise erwähnt wurde. Insgesamt wurden bei nur 3 von 46 Publikationen Streumaße zumindest teilweise erwähnt.

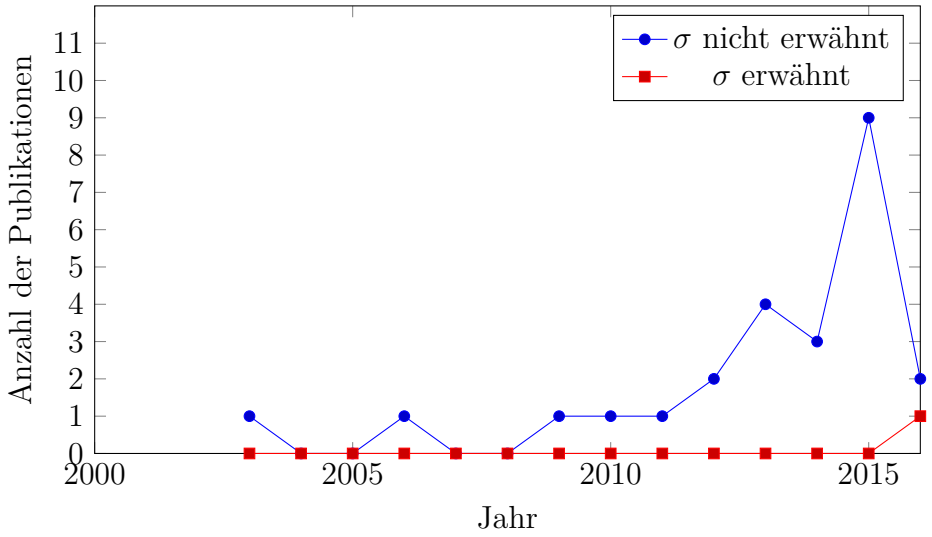


Abbildung B.4.: Anzahl der Publikationen der Forschungsgruppe von Prof. Dr.-Ing. Henning Meyerhenke bei denen bei Laufzeitangaben ein Streumaß erwähnt, nicht oder nur teilweise erwähnt wurde. Insgesamt wurden bei nur einer von 26 Publikationen Streumaße zumindest teilweise erwähnt.

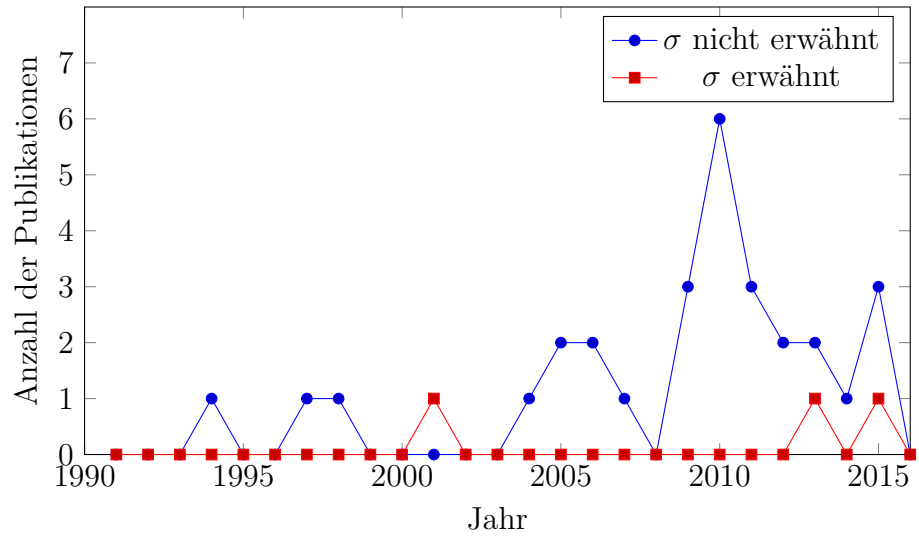


Abbildung B.5.: Anzahl der Publikationen des Lehrstuhl von Prof. Dr.-Ing. Gregor Snelting bei denen bei Laufzeitangaben ein Streumaß erwähnt, nicht oder nur teilweise erwähnt wurde. Insgesamt wurden bei nur 3 von 32 Publikationen Streumaße zumindest teilweise erwähnt.

Literaturverzeichnis

[Ben-Kiki u. a. 2005] Ben-Kiki, Oren ; Evans, Clark ; Ingerson, Brian: YAML Ain't Markup Language (YAML™) Version 1.1. In: *yaml.org, Tech. Rep* (2005)

[Breitner 2016] Breitner, Joachim: *GHC-Commit: Do not claim that -O2 does not do better than -O.* http://git.haskell.org/ghc.git/commit/3d245bf5255ebfb72813596fa93b9051f7518321. Version: 2016

[Buchwald u. a. 2011] Buchwald, Sebastian ; Zwinkau, Andreas ; Bersch, Thomas: SSA-Based Register Allocation with PBQP. In: Knoop, Jens (Hrsg.): *Compiler Construction* Bd. 6601, Springer Berlin / Heidelberg, 2011 (Lecture Notes in Computer Science), S. 42–61. – 10.1007/978-3-642-19861-8_4

[Collin Winter 2006] Collin Winter, Tony L.: *PEP 3107 — Function Annotations.* https://www.python.org/dev/peps/pep-3107/. Version: Dezember 2006

[Crawford 2013] Crawford, Drew: *Why mobile web apps are slow.* http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/. Version: July 2013

[Curtsinger u. Berger 2013] Curtsinger, Charlie ; Berger, Emery D.: STABILIZER: statistically sound performance evaluation. In: *ACM SIGARCH Computer Architecture News* Bd. 41 ACM, 2013, S. 219–228

[Dietrich u. Lohmann 2015] Dietrich, Christian ; Lohmann, Daniel: The Dataref Versuchung: Saving Time Through Better Internal Repeatability. In: *SIGOPS Oper. Syst. Rev.* 49 (2015), Januar, Nr. 1, S. 51–60. http://dx.doi.org/10.1145/2723872.2723880. – DOI 10.1145/2723872.2723880. – ISSN 0163–5980

[Diwan u. a. 2009] Diwan, Todd Mytkowicz A. ; Hauswirth, Matthias ; Sweeney, Peter F.: Producing Wrong Data Without Doing Anything Obviously Wrong! (2009)

[Fleming u. Wallace 1986] Fleming, Philip J. ; Wallace, John J.: How not to lie with statistics: the correct way to summarize benchmark results. In: *Communications of the ACM* 29 (1986), S. 218–221

[Fulgham u. Gouy] Fulgham, Brent ; Gouy, Isaac: *The Computer Language Benchmarks Game*. http://benchmarksgame.alioth.debian.org

[Gramer 1936] Gramer, H.: Über eine Eigenschaft der normalen Verteilungsfunktion. In: *Mathematische Zeitschrift* 41 (1936), S. 405–414

[Granlund 2002] Granlund, Hanhong Xue; T.: *Pi computation using Chudnovsky's algorithm*. https://gmplib.org/download/misc/gmp-chudnovsky.c. Version: 2002 – 2005. – [Online, Source Code; Stand 14. Februar 2016]

[haskell.org a] haskell.org: *The User's Guide — 1.5. Release notes for version 7.0.1*. https://downloads.haskell.org/~ghc/7.0.1/docs/html/users_guide/release-7-0-1.html

[haskell.org b] haskell.org: *The User's Guide — 4.10 Optimisation (code improvement)*, https://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/options-optimise.html

[Heiser] Heiser, Gernot: *Systems Benchmarking Crimes*. https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html

[Henze 2013] Henze, Norbert: *Stochastik für Einsteiger*. Springer Spektrum, 2013. http://dx.doi.org/10.1007/978-3-658-03077-3. http://dx.doi.org/10.1007/978-3-658-03077-3. – ISBN 978–3–658–03076–6

[Inc. 2013] Inc., Advanced Micro D.: AMD64 Architecture Programmer's Manual: Volume 2: System Programming. In: *AMD Pub* (2013), Nr. 24593

[Intel 2008] Intel: *Intel® Turbo Boost Technology in Intel® CoreTM Microarchitecture (Nehalem) Based Processors*. 2008. – Whitepaper

[Janczyk u. Pfister 2015] Janczyk, Markus ; Pfister, Roland: *Stochastik*. Springer Spektrum, 2015. http://dx.doi.org/10.1007/978-3-662-47106-7. http://dx.doi.org/10.1007/978-3-662-47106-7. – ISBN 978–3–662–47105–0

[rust lang.org 2015] lang.org rust: *Announcing Rust 1.0*. http://blog.rust-lang.org/2015/05/15/Rust-1.0.html. Version: 2015

[LLG] LLG: *The Logical Language Group: Online Dictionary Query - temci*. http://jbovlaste.lojban.org/lookup.pl?Form=lookup.pl2&Database=*&Query=temci

[Love 2014] Love, Robert M.: *sched_setaffinity — User Commands*, 2014. http://man7.org/linux/man-pages/man1/taskset.1.html

[Magro u. a. 2002] Magro, William ; Petersen, Paul ; Shah, Sanjiv: Hyper-Threading Technology: Impact on Compute-Intensive Workloads. In: *Intel Technology Journal* 6 (2002), Nr. 1, S. 1. – ISSN 1535864X

[Marr u. a. 2002] Marr, Deborah T. ; Binns, Frank ; Hill, David L. ; Hinton, Glenn ; Koufaty, David A. ; Miller, J. A. ; Upton, Michael: Hyper-Threading Technology Architecture and Microarchitecture. In: *Intel Technology Journal* 6 (2002), Nr. 1, S. 1. – ISSN 1535864X

[Nauta 2010] Nauta, Jozef: *Statistics in Clinical Vaccine Trials*. Berlin, Heidelberg, 2010 (SpringerLink : Bücher)

[man-pages project 2014a] project, Linux man-pages: *cpuset — Linux Programmer's Manual*, 2014. http://man7.org/linux/man-pages/man7/cpuset.7.html

[man-pages project 2014b] project, Linux man-pages: *getpriority — Linux Programmer's Manual*, 2014. http://man7.org/linux/man-pages/man2/getpriority.2.html

[man-pages project 2014c] project, Linux man-pages: *sched_setaffinity — Linux Programmer's Manual*, 2014. http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html

[van Rossum u. a. 2014] Rossum, Guido van ; Lehtosalo, Jukka ; Langa, Lukasz: *PEP 0484 — Type Hints*. https://www.python.org/dev/peps/pep-0484/. Version: September 2014

[Sawilowsky u. Blair 1992] Sawilowsky, Shlomo S. ; Blair, R. C.: A more realistic look at the robustness and Type II error properties of the t test to departures from population normality. In: *Psychological bulletin* 111 (1992), 352. https://www.researchgate.net/publication/232463233

[Silberschatz u. a. 2010] Silberschatz, Abraham ; Galvin, Peter B. ; Gagne, Greg: *Operating system concepts*. 8. ed., internat. student version. Hoboken, NJ : Wiley, 2010. – ISBN 978–0–470–23399–3

Erklärung

Hiermit erkläre ich, Johannes Bechberger, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum

Unterschrift