Dynamische Anpassung Compiler-basierter Instrumentierung unter Nutzung von Laufzeitstatistiken

Verteidigung

Philipp Trommler
philipp.trommler@tu-dresden.de

07. November 2016

Betreuer: Joseph Schuchart, Robert Schöne
Hochschullehrer: Prof. Dr. W. E. Nagel, Prof. Dr. J. Castrillon

ZIH
Zentrum für Informationsdienste und Hochleistungsrechnen

Gliederung

1. Einleitung
2. Technischer Hintergrund & Implementierung
3. Ergebnisse
4. Diskussion
5. Fazit & Ausblick

Gliederung

Einleitung – Was ist Instrumentierung?

Performance-Analyse dient dem Zweck, genauere Informationen über den Programmablauf zu erhalten:

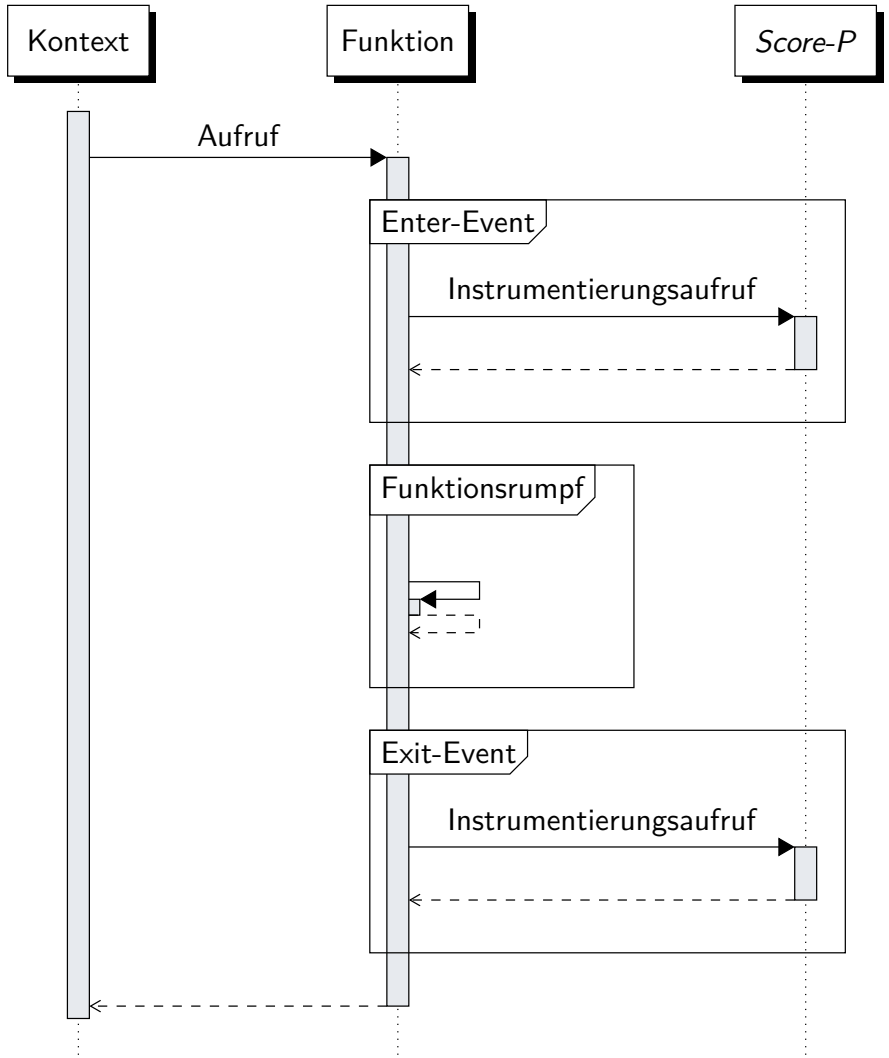
- Aufrufanzahl/-dauer einzelner Funktionen
- I/O
- Kommunikation
- ...

Hierfür kann Instrumentierung genutzt werden:

- Compilerinstrumentierung (im Folgenden betrachtet)
- Instrumentierung von Bibliotheken (z. B. Wrapping via `LD_PRELOAD`)
- Nutzerinstrumentierung (z. B. `SCOREP_User_RegionEnter`)
- ...

Einleitung – Was ist Instrumentierung?

Bei der Instrumentierung werden zusätzliche Aufrufe in den Programmablauf integriert:



Einleitung – Warum Filtern?

- Negativer Einfluss der Instrumentierung auf Programmlaufzeit
  - *Performance Perturbation*
  - Abwägen zwischen Informationsgewinn und Overhead
- Entstehende Datenmengen mitunter sehr groß
- Lineare Abhängigkeit von der Anzahl der Instrumentierungsaufrufe
- Hochfrequente aber kurze Funktionen haben oft geringen Informationsgehalt
  $\rightarrow$ Potential zur Senkung des Laufzeit-Overheads und der Menge der entstehenden Performance-Daten

Einleitung – Warum diese Arbeit?

Score-P enthält bereits die Möglichkeit zu filtern

- Liste mit den zu filternden Funktionen muss übergeben werden
- Instrumentierung verbleibt im Code

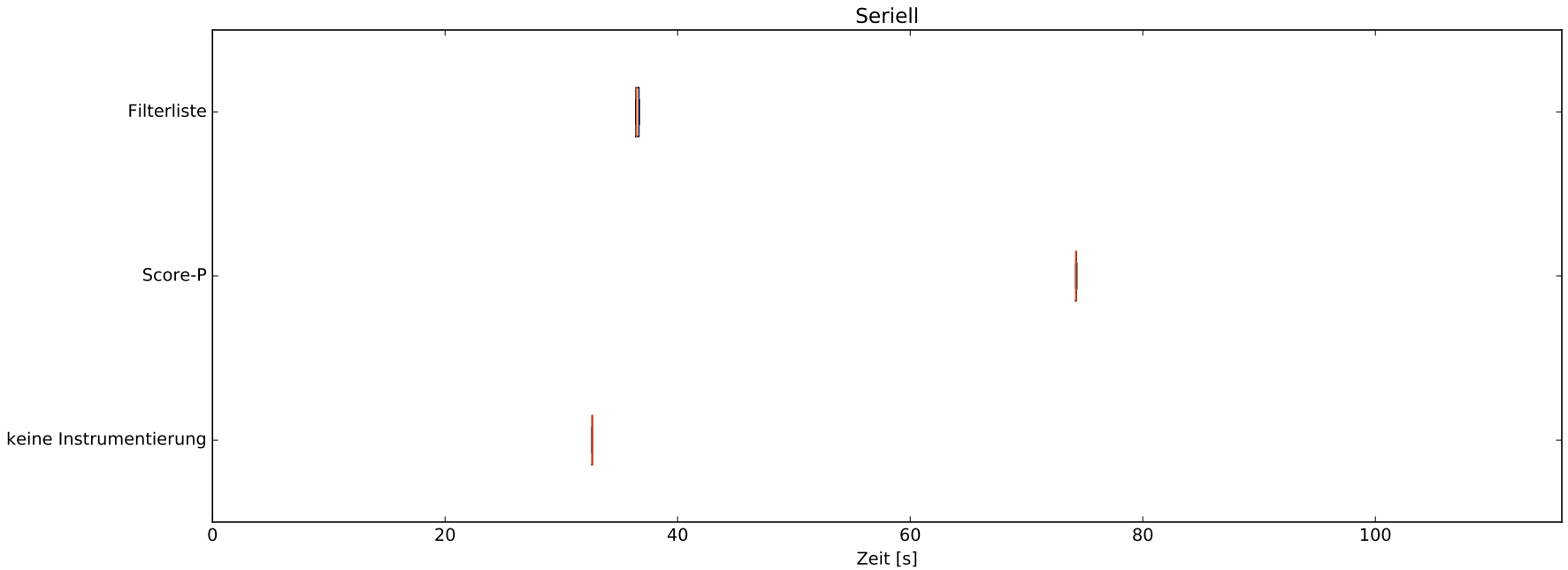
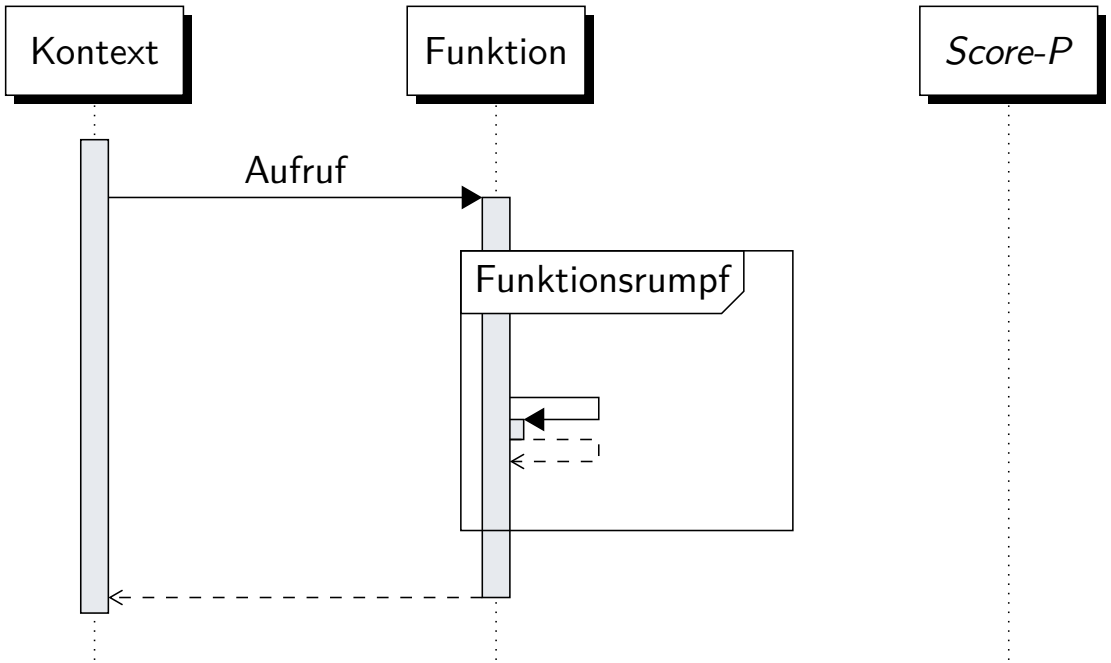


Abbildung: Vergleich der Ausführungszeiten des *NPB bt-mz*-Benchmarks Größe A für den uninstrumentierten und den vollständig instrumentierten Fall sowie unter Nutzung von *Score-Ps* Filterfunktion.

Einleitung – Warum diese Arbeit?

Besser wäre:

- Zu filternde Funktionen
  - werden **zur Laufzeit**
  - anhand gegebener Parameter **selbständig** erkannt
- Instrumentierung wird **vollständig** entfernt



Gliederung

Implementierung

- Implementierung als prototypisches Plugin für *Score-Ps* „Substrates“-Schnittstelle
- Aufgabe des Plugins in mehrere Teilaufgaben gegliedert:
  - Bestimmen der Aufrufanzahl und -dauer für jede instrumentierte Funktionen
  - Bewertung der Funktionen anhand dieser Werte mithilfe von Metriken
  - Entscheidung zu filtern
  - Suchen der Instrumentierungsaufrufe
  - Überschreiben der Instrumentierungsaufrufe

Implementierung – Metriken

Entscheidung zum Filtern anhand von zwei verschiedenen Metriken:

- Absolute Metrik

$$\texttt{irrelevant(region)} = \begin{cases} \texttt{true} & \text{für } \texttt{mean\_duration(region) < threshold} \\ \texttt{false} & \text{sonst} \end{cases} \tag{1}$$

- Relative Metrik

$$\texttt{condition} = \frac{\sum_{i=1}^{n} \texttt{mean\_duration(i)}}{n} \texttt{ - threshold}$$

$$\texttt{irrelevant(region)} = \begin{cases} \texttt{true} & \text{für } \texttt{mean\_duration(region) < condition} \\ \texttt{false} & \text{sonst} \end{cases} \tag{2}$$

- `threshold` wird vom Benutzer festgelegt (angegeben in *Cycles*)

Implementierung – Metriken

Entscheidung zum Filtern anhand von zwei verschiedenen Metriken:

- Absolute Metrik

$$\texttt{irrelevant(region)} = \begin{cases} \texttt{true} & \text{für } \texttt{mean\_duration(region) < threshold} \\ \texttt{false} & \text{sonst} \end{cases} \tag{1}$$

- Relative Metrik

$$\texttt{condition} = \frac{\sum_{i=1}^{n} \texttt{mean\_duration(i)}}{\texttt{n}} \texttt{ - threshold}$$

$$\texttt{irrelevant(region)} = \begin{cases} \texttt{true} & \text{für } \texttt{mean\_duration(region) < condition} \\ \texttt{false} & \text{sonst} \end{cases} \tag{2}$$

- `threshold` wird vom Benutzer festgelegt (angegeben in *Cycles*)

Implementierung – Metriken

Entscheidung zum Filtern anhand von zwei verschiedenen Metriken:

- Absolute Metrik

$$\texttt{irrelevant(region)} = \begin{cases} \texttt{true} & \text{für } \texttt{mean\_duration(region) < threshold} \\ \texttt{false} & \text{sonst} \end{cases} \tag{1}$$

- Relative Metrik

$$\texttt{condition} = \frac{\sum_{\texttt{i=1}}^{\texttt{n}} \texttt{mean\_duration(i)}}{\texttt{n}} \texttt{ - threshold}$$

$$\texttt{irrelevant(region)} = \begin{cases} \texttt{true} & \text{für } \texttt{mean\_duration(region) < condition} \\ \texttt{false} & \text{sonst} \end{cases} \tag{2}$$

- `threshold` wird vom Benutzer festgelegt (angegeben in *Cycles*)

Technischer Hintergrund – Stack

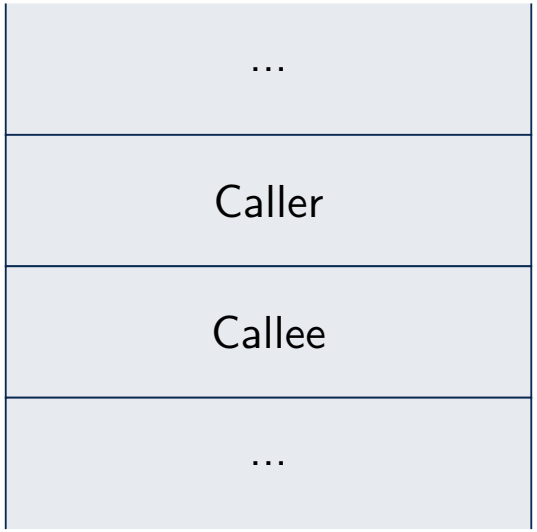


Abbildung: Schematische Darstellung eines Kellerspeichers (*Stack*).

- *Stack* enthält Speicherbereich (*Stack-Frame*) für jede aufgerufene Funktion
- *Frame* der Aufgerufenen (*Callee*) direkt hinter *Frame* der Aufrufenden (*Caller*) im Speicher
- *Frame* bleibt erhalten, bis Funktion verlassen wird
- *Frame* enthält alle relevanten lokalen Informationen (Variablen, `%ip`)

Implementierung – Finden der Instrumentierungsaufrufe

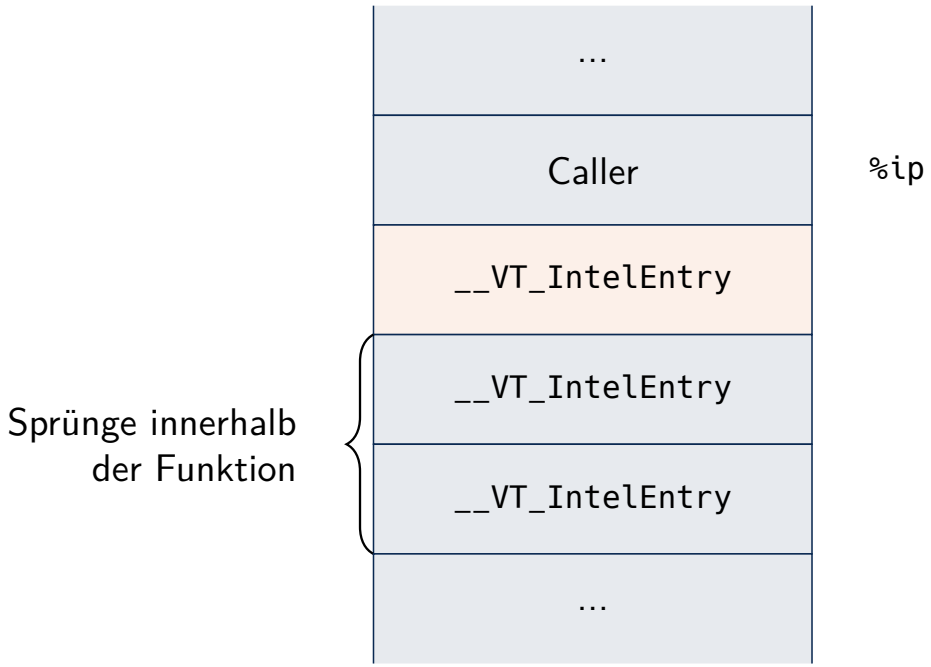


Abbildung: Beispielhafter Ausschnitt aus einem *Stack*.

- *Stack-Unwinding* mit `libunwind`
- Finden der **tatsächlichen**, aufrufenden Funktion
- Ermitteln von `%ip`

Technischer Hintergrund – Funktionsaufrufe

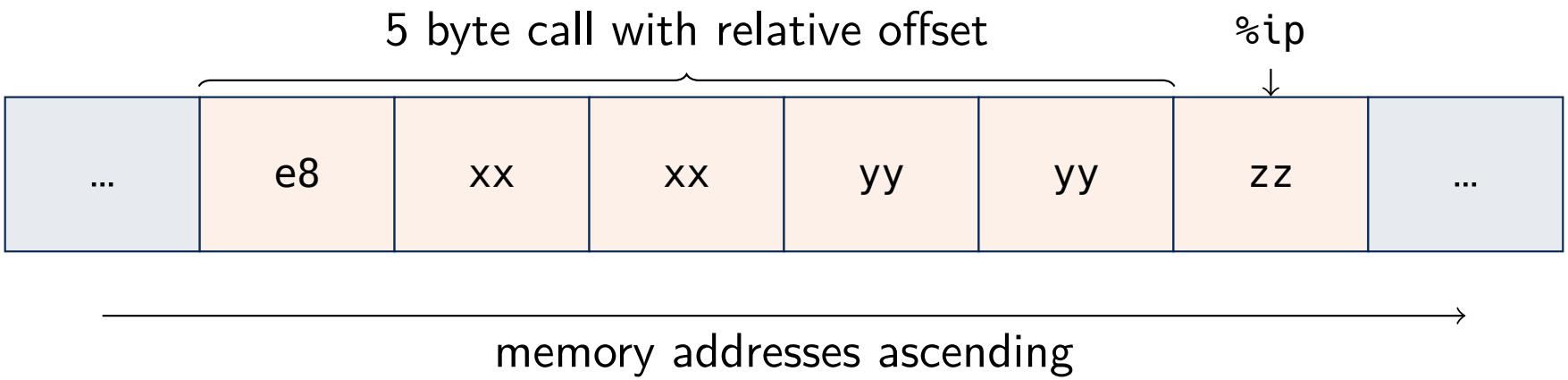


Abbildung: Ausschnitt aus dem Textsegment eines Programms.

- *OpCode* e8 leitet 5 Byte langen Sprungbefehl ein
- Sprungadresse relativ angegeben (*Offset* 16 oder 32 Bit lang)
- `%ip` zeigt auf erstes Byte hinter Sprungbefehl

Implementierung – Entfernen

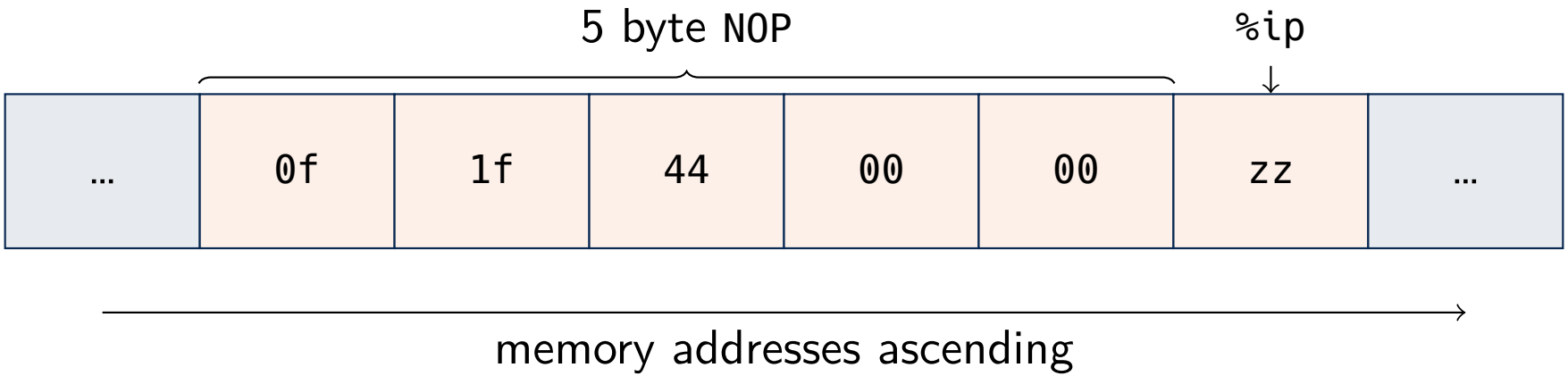
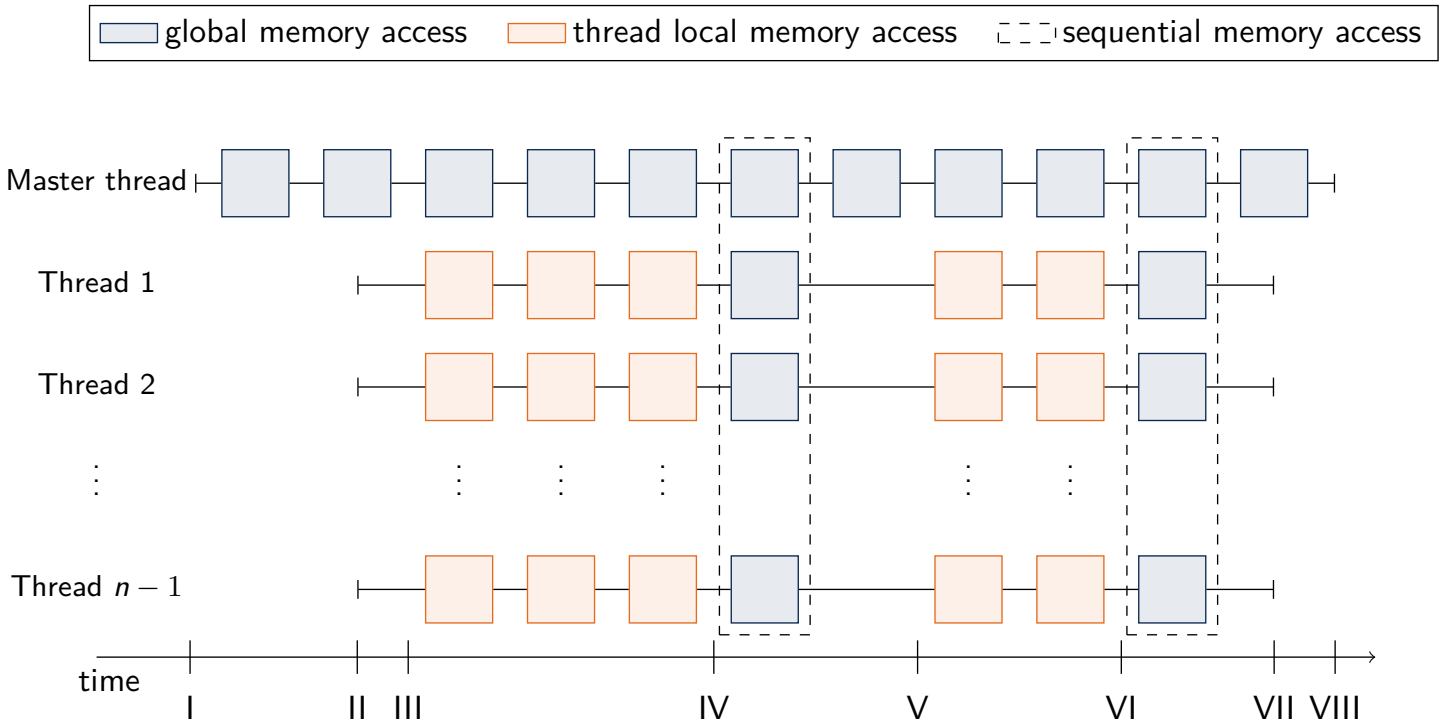


Abbildung: Mit `NOP` überschriebener Funktionsaufruf.

→ 5 Byte vor `%ip` werden mit `NOP` überschrieben

Implementierung – Multithreading

Gemeinsames Textsegment der Threads verhindert das Überschreiben während thread-paralleler Ausführung



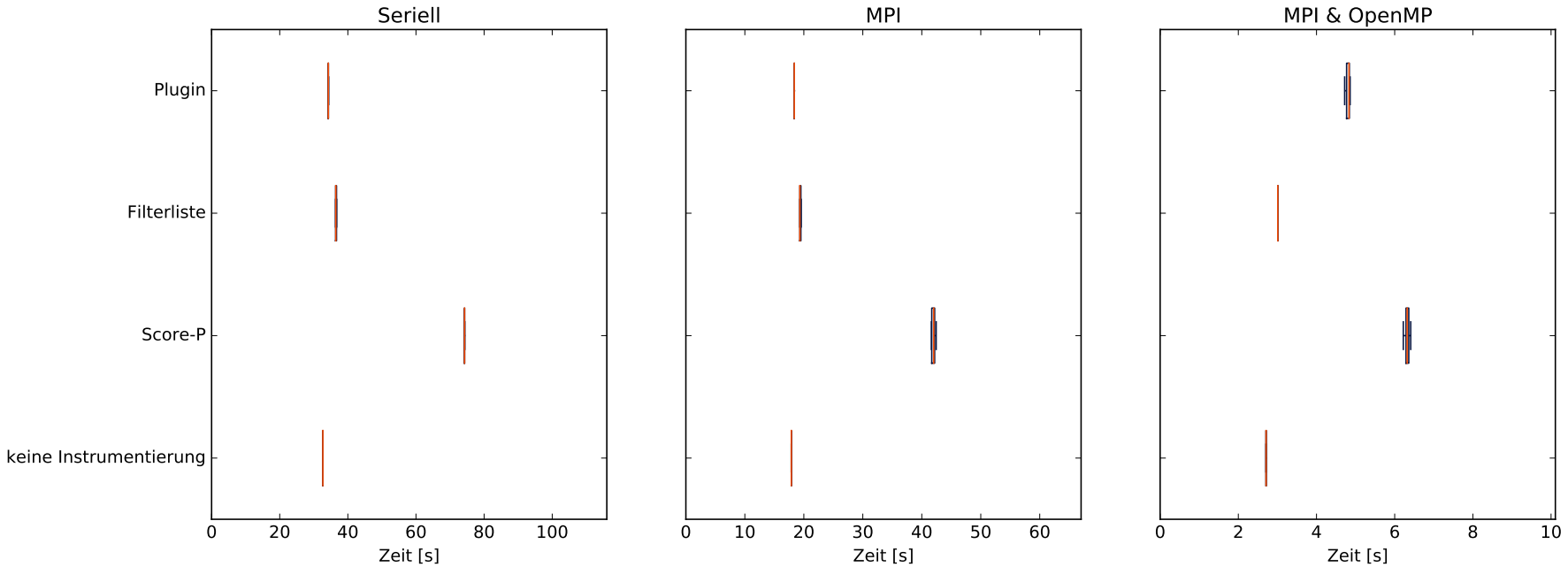
- Thread-lokales Speichern der Informationen
- Zusammenführen beim *team end* der Threads
- Filtern nur während der seriellen Phasen des Programms

Gliederung

Ergebnisse

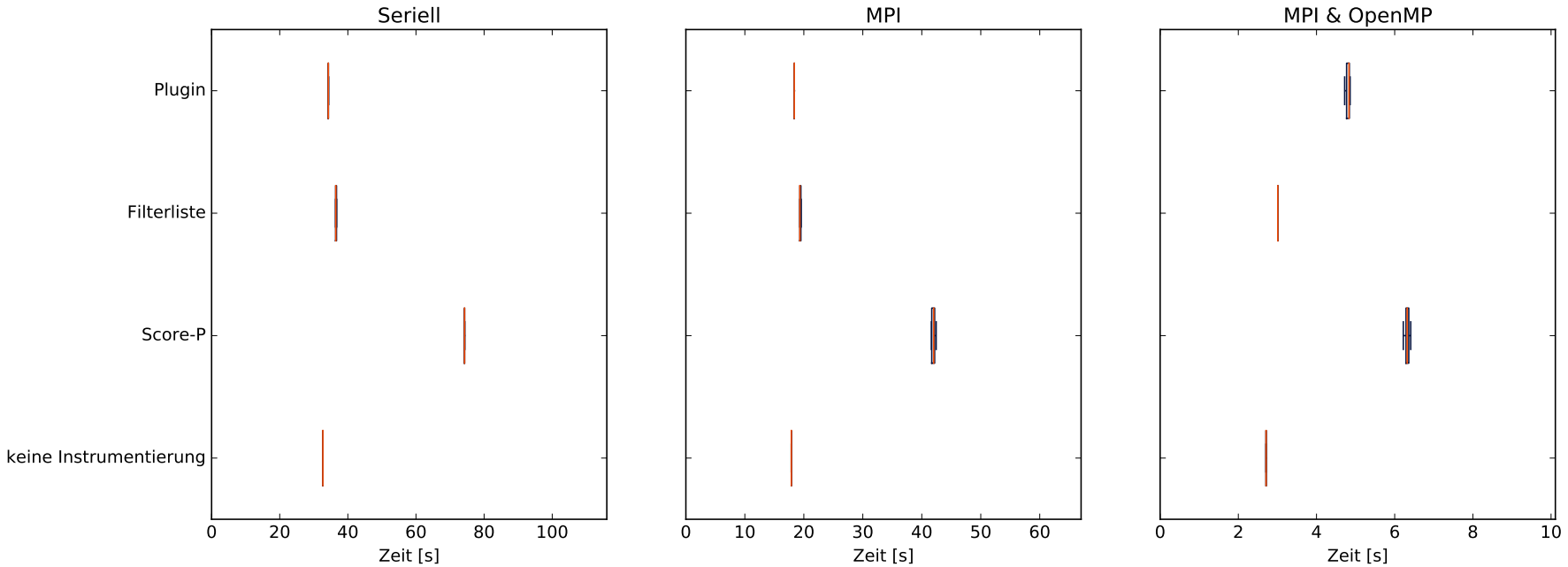
- Testsystem
  - 2 × *Intel Xeon E5-2680 v3*
    - Je 12 Kerne
    - 2,5 *GHz* (Turbo bis 3,3 *GHz*)
  - 8 × 16 *GB* DDR4-2133 RAM
- *NPB bt-mz & sp-mz*
  - seriell
  - *MPI* (2 Prozesse)
  - *MPI & OpenMP* (2 Prozesse, je 12 Threads)
- *LULESH*
  - seriell
  - *OpenMP* (12 Threads)
- Absolute Metrik, Grenzwert 1000 *Cycles*
- Jeweils 10 Durchführungen

Ergebnisse – Ausführungszeiten – *NPB bt-mz A Intel*



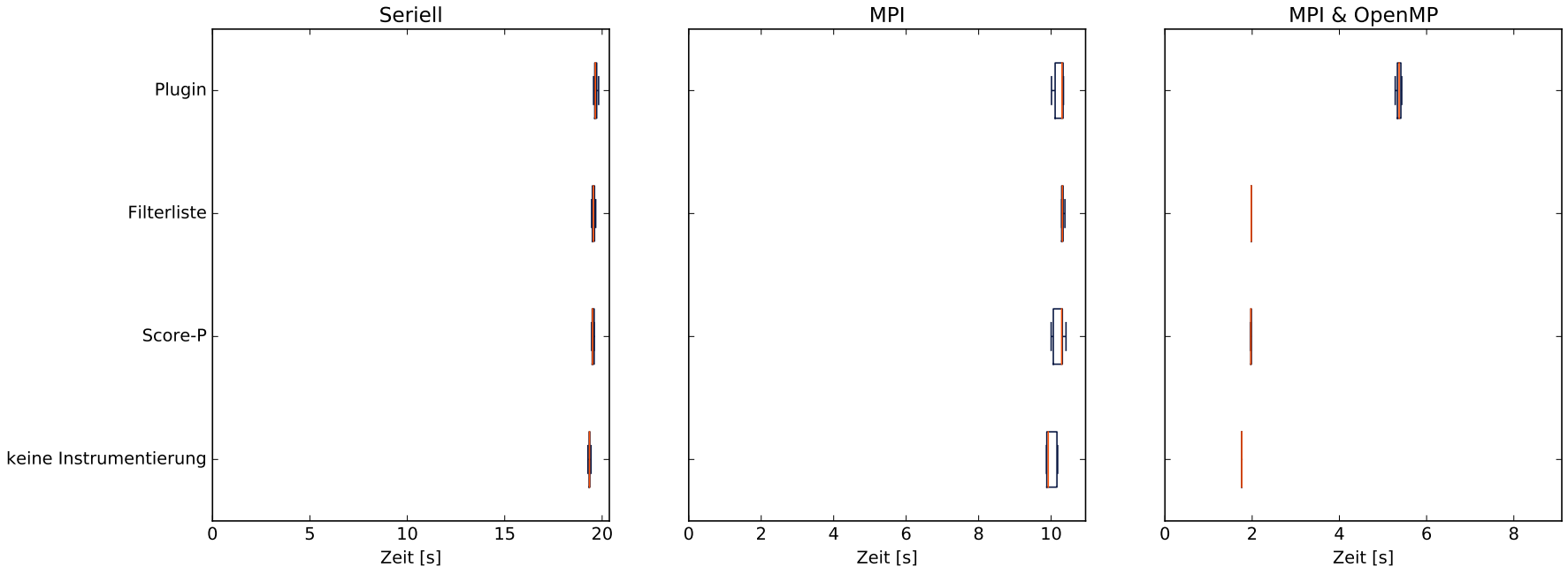
	Seriell	*MPI*	*MPI & OpenMP*
Score-P	$127,37\,\%$	$135,01\,\%$	$133,21\,\%$
Filterliste	$11,64\,\%$	$8,11\,\%$	$11,07\,\%$
Plugin	$4,90\,\%$	$2,80\,\%$	$78,23\,\%$

Ergebnisse – Ausführungszeiten – *NPB bt-mz A Intel*



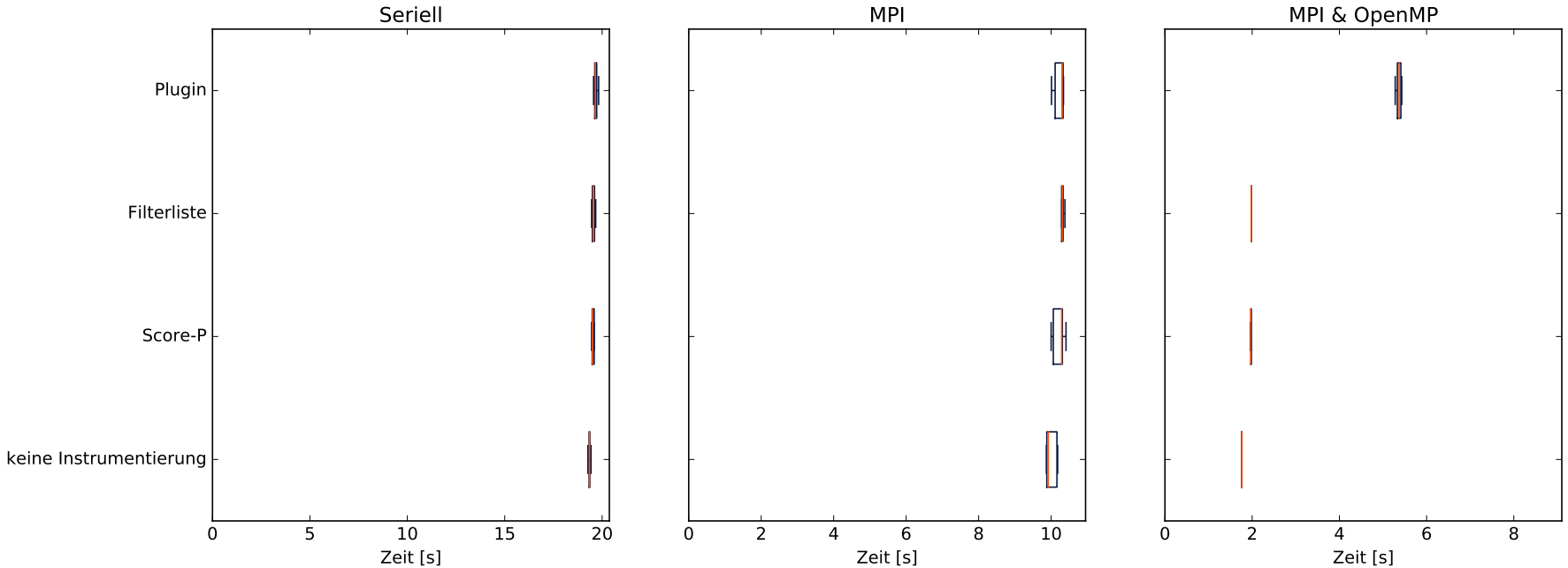
	Seriell	*MPI*	*MPI & OpenMP*
Score-P	$127,37\,\%$	$135,01\,\%$	$133,21\,\%$
Filterliste	$11,64\,\%$	$8,11\,\%$	$11,07\,\%$
Plugin	$4,90\,\%$	$2,80\,\%$	$78,23\,\%$

Ergebnisse – Ausführungszeiten – *NPB sp-mz* A *Intel*



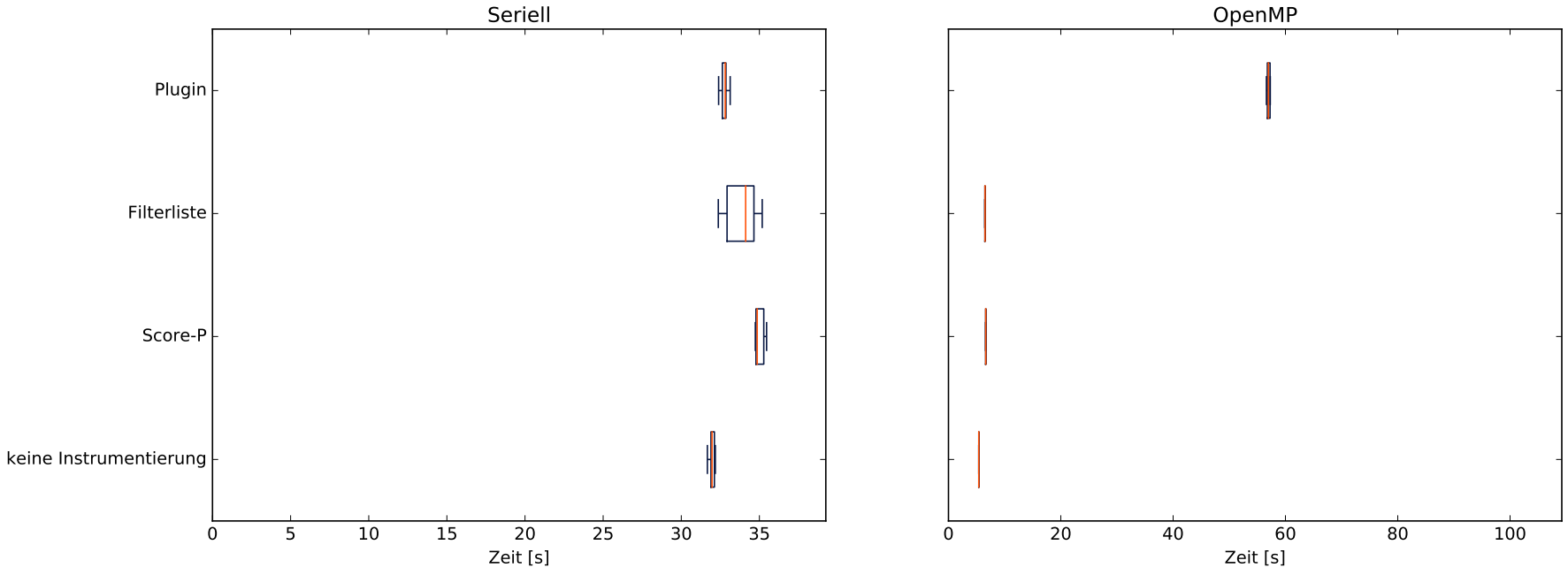
	Seriell	*MPI*	*MPI & OpenMP*
Score-P	$0,83\,\%$	$3,73\,\%$	$11,93\,\%$
Filterliste	$1,03\,\%$	$3,93\,\%$	$12,50\,\%$
Plugin	$1,55\,\%$	$3,83\,\%$	$205,11\,\%$

Ergebnisse – Ausführungszeiten – *NPB sp-mz* A *Intel*



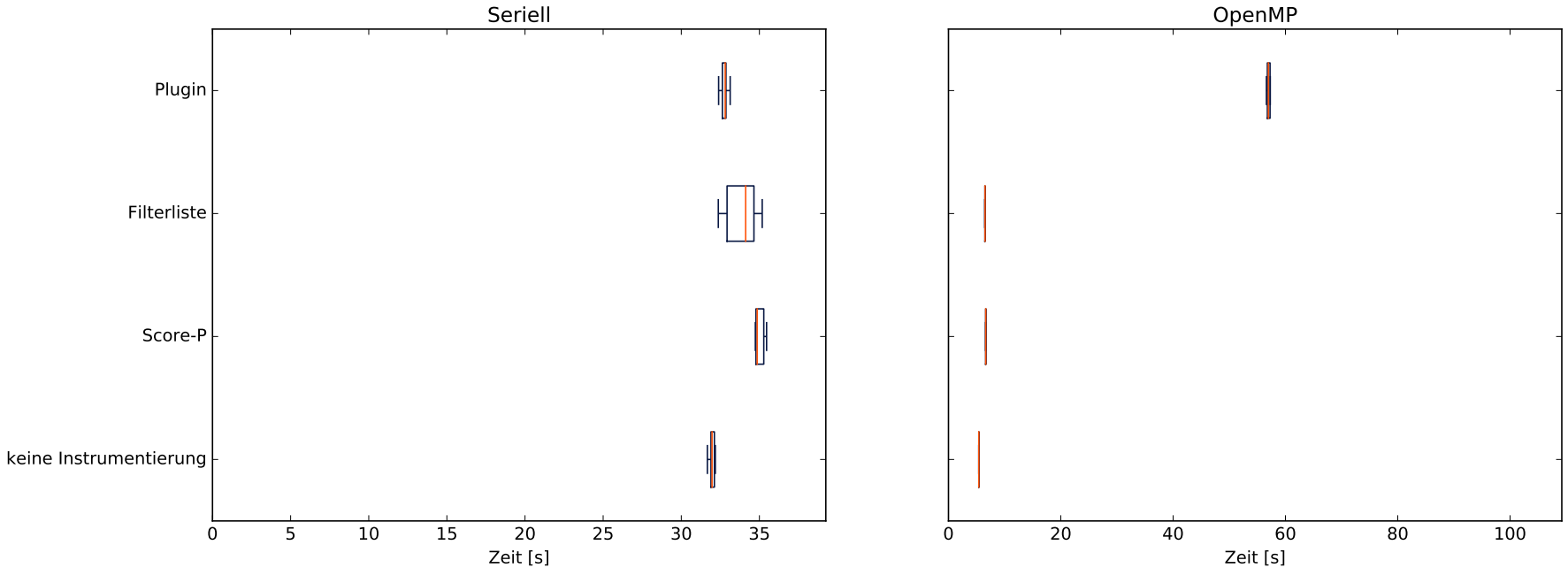
	Seriell	*MPI*	*MPI & OpenMP*
Score-P	$0,83\,\%$	$3,73\,\%$	$11,93\,\%$
Filterliste	$1,03\,\%$	$3,93\,\%$	$12,50\,\%$
Plugin	$1,55\,\%$	$3,83\,\%$	$205,11\,\%$

Ergebnisse – Ausführungszeiten – *LULESH GCC*



	Seriell	*OpenMP*
Score-P	$9,04\,\%$	$22,96\,\%$
Filterliste	$6,73\,\%$	$20,56\,\%$
Plugin	$2,63\,\%$	$955,00\,\%$

Ergebnisse – Ausführungszeiten – *LULESH GCC*



	Seriell	*OpenMP*
Score-P	$9,04\,\%$	$22,96\,\%$
Filterliste	$6,73\,\%$	$20,56\,\%$
Plugin	$2,63\,\%$	$955,00\,\%$

Ergebnisse – Trace – *LULESH* seriell

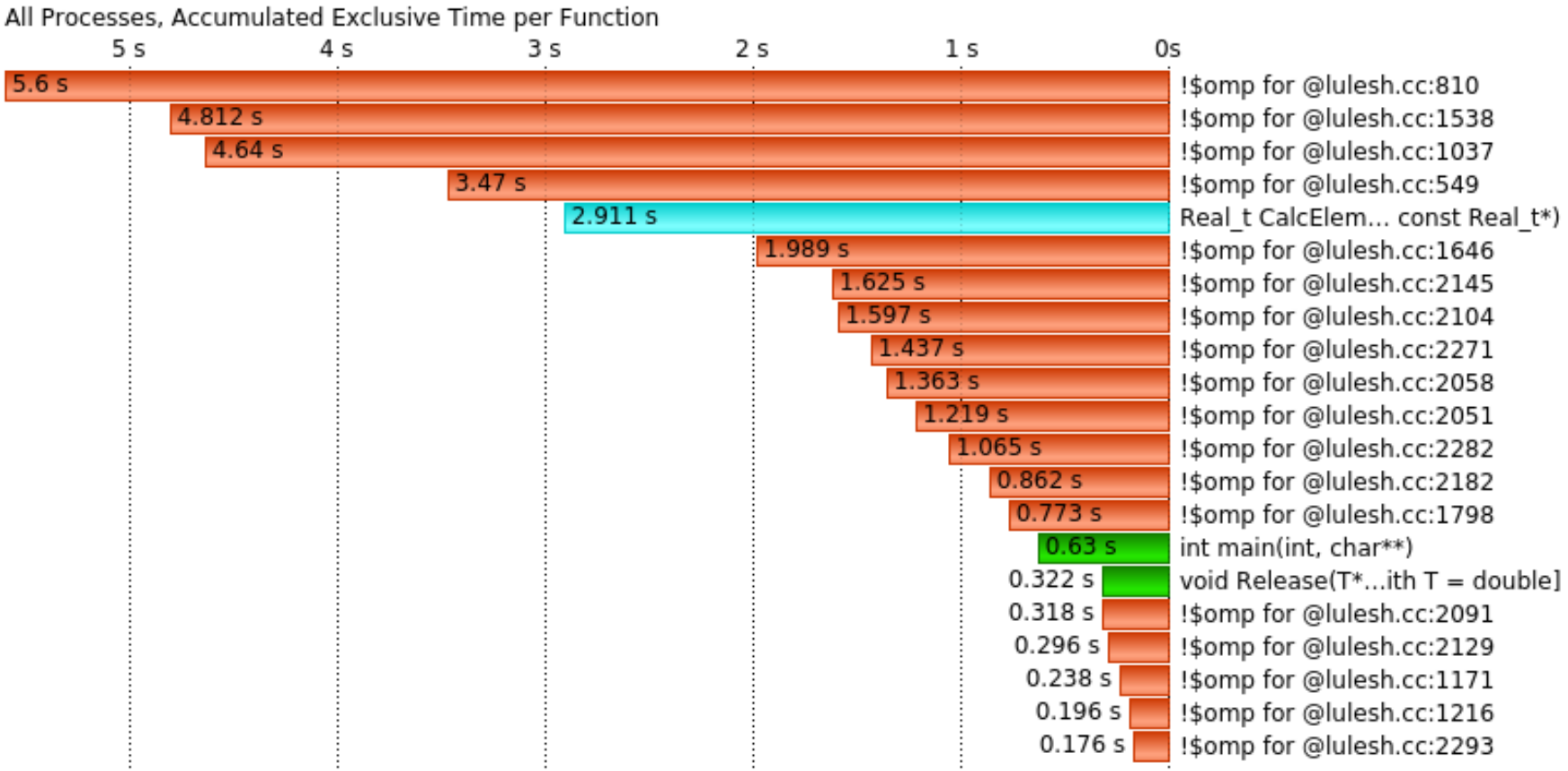


Abbildung: Akkumulierte Laufzeiten der Funktionen des *LULESH*-Benchmarks bei vollständiger Instrumentierung.

Ergebnisse – Trace – *LULESH* seriell

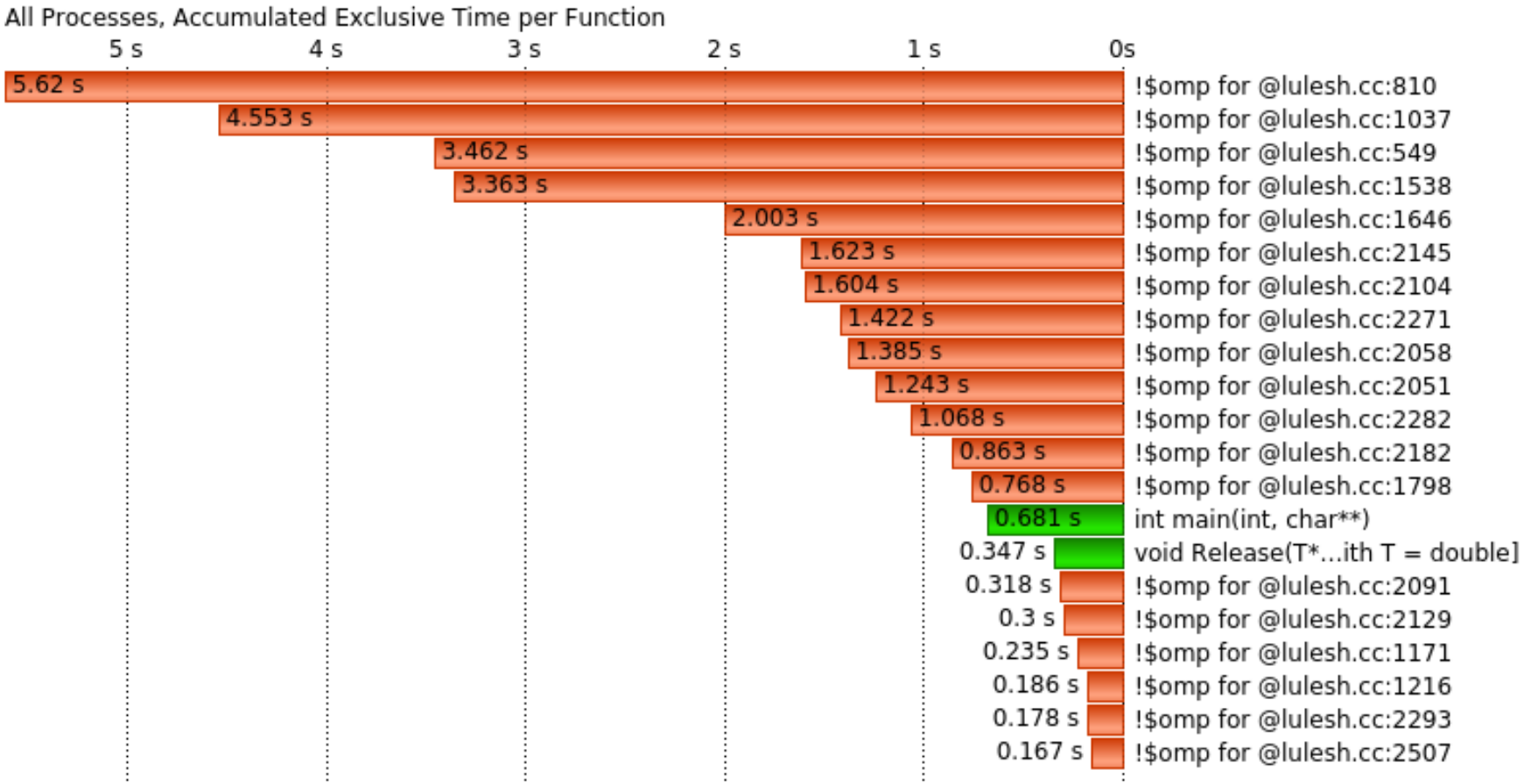


Abbildung: Akkumulierte Laufzeiten der Funktionen des *LULESH*-Benchmarks unter Nutzung des Plugins.

→ Reduktion der Datenmenge von ca. 646 *MB* auf ca. 67 *MB*

Ergebnisse – Trace – *LULESH* seriell

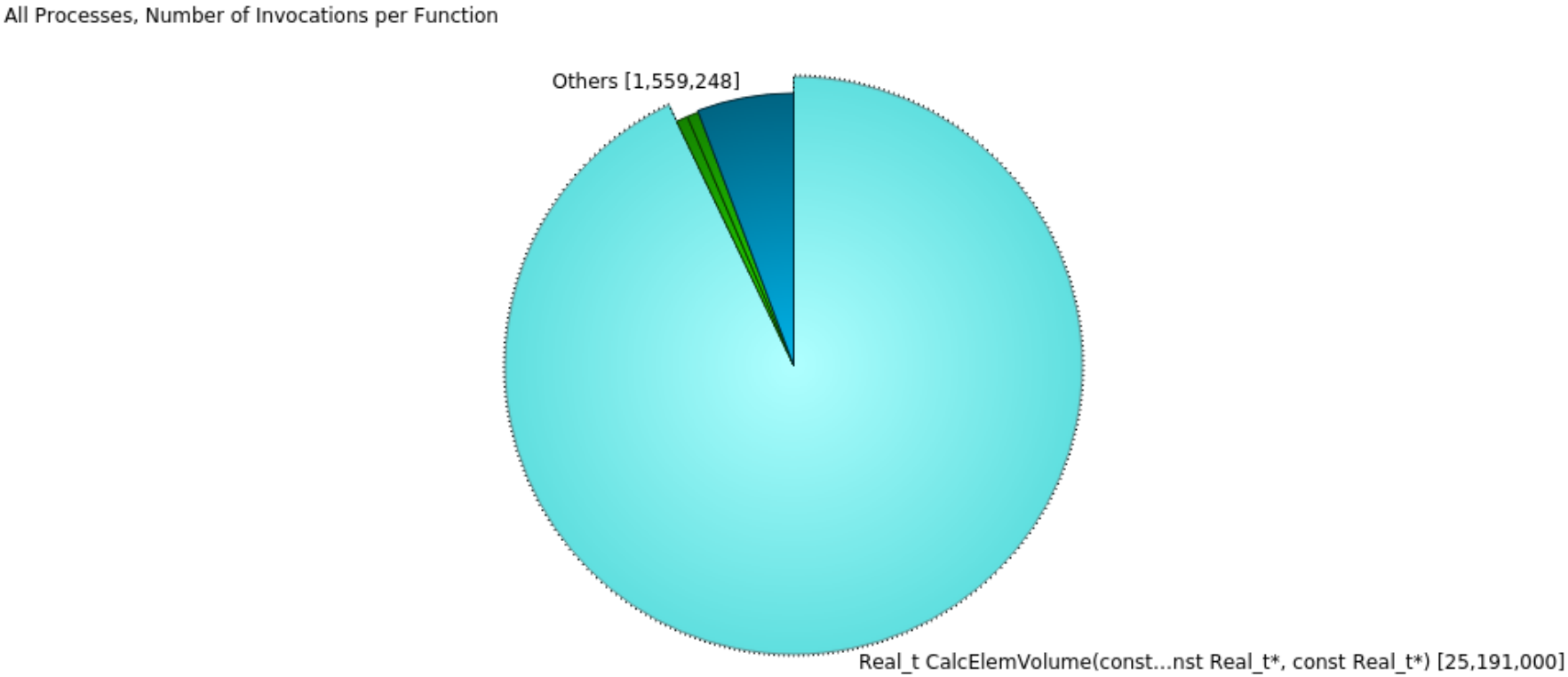


Abbildung: Aufrufanzahl der Funktionen des *LULESH*-Benchmarks.

Gliederung

Diskussion – Speicherzugriffe und Synchronisierung

- Synchronisierung zwischen Threads benötigt viele Speicherzugriffe
- Deutlich in den Ergebnissen sichtbar
- Untersuchung mit *perf* bestätigt dies
  - ca. 600 Mio. zu 1450 Mio. *LLC-loads*
  - ca. 250 Mio. zu 750 Mio. *LLC-stores*

Tabelle: Lesende Zugriffe auf den *last level cache* (*LLC-loads*).

Anteil	Herkunft	Symbol
$10,34\,\%$	`libdynamic_filtering_plugin.so`	`on_team_end`
$5,80\,\%$	`lulesh2.0`	`IntegrateStressForElems`
$5,09\,\%$	`lulesh2.0`	`EvalEOSForElems`
$4,09\,\%$	`lulesh2.0`	`CalcFBHourglass…`
$3,92\,\%$	`lulesh2.0`	`metric_perf_open`
⋮	⋮	⋮

Diskussion – Funktionsaufrufe

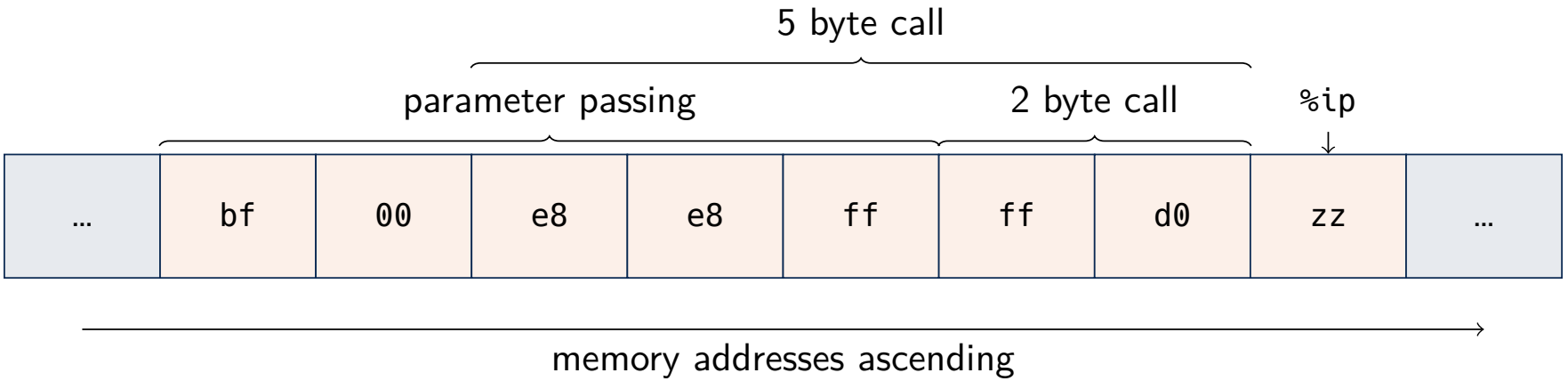


Abbildung: Verdeutlichung des Problems der variablen Sprungbefehlslänge.

- *CISC*-Befehlssätze haben variable Befehlslänge
- Befehlslänge nur aus Kontext bestimmbar
- Kontext beim Lesen in umgekehrter Reihenfolge nicht vorhanden

Diskussion

Weitere Herausforderungen:

- Mehrere Rücksprünge aus einer Funktion
  - Werden vom genutzten Algorithmus nicht gefunden
  - Führen zu invalidem Zustand in *Score-P*
    $\rightarrow$ Messung wird abgebrochen
- Weitere Instrumentierungsarten
  - Bibliotheksinstrumentierung
  - Nutzerinstrumentierung
  - ...

Gliederung

Fazit & Ausblick

Bisherige Ergebnisse zeigen Potential des Ansatzes:

- Prinzipiell gute Filterwirkung
- Kein händisches Erstellen der Filterliste notwendig
- Rudimentäre Informationen bleiben trotz Filtern erhalten
- Auch komplexere Metriken möglich

Dennoch notwendige Verbesserungen:

- Unterstützung verschiedener Instrumentierungsarten
- Erkennen aller Aufruftypen
- Erkennen aller Rücksprünge
- **Reduktion des Overheads**

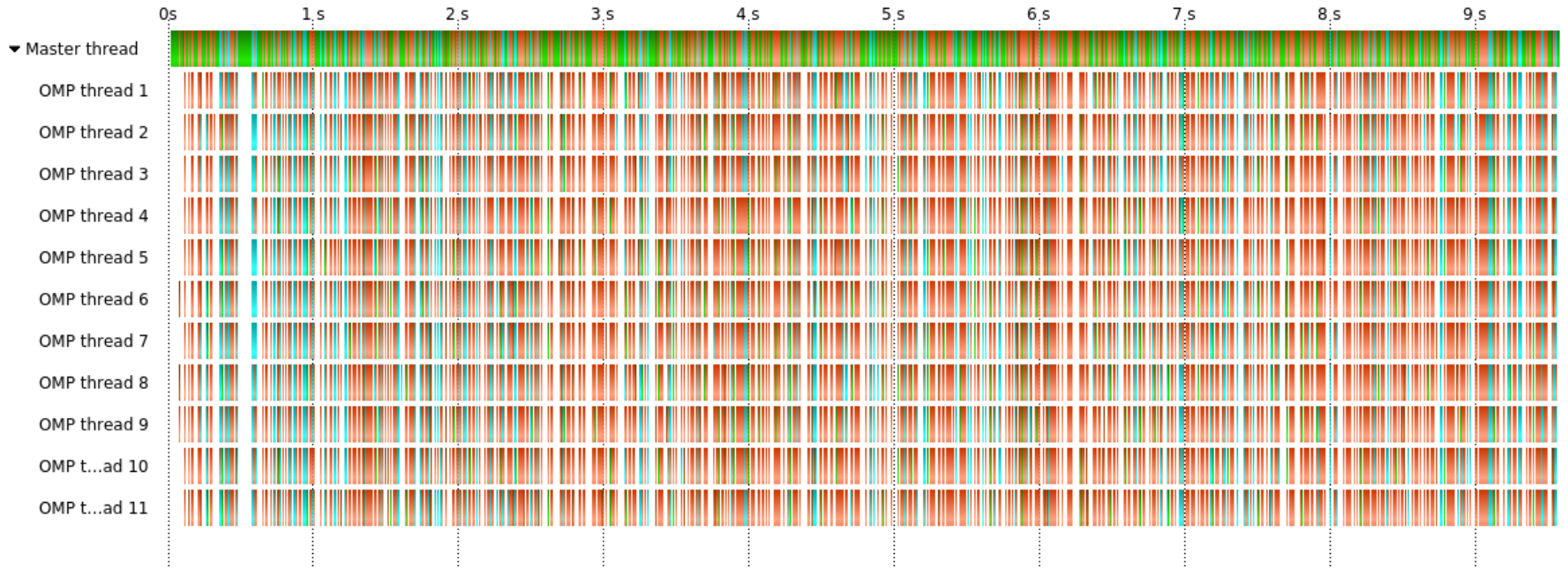
→ Integration der Funktionalität in *Score-P* möglich

Genutzte Software

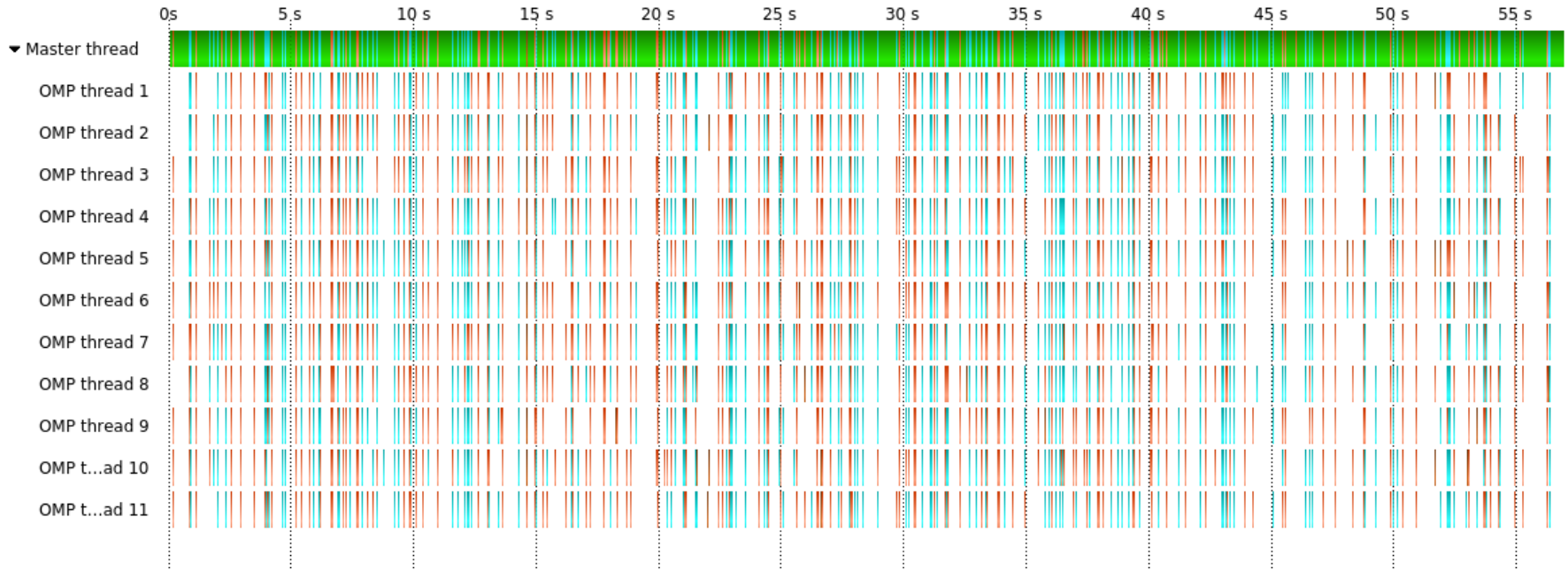
- Plugin:
  `https://github.com/Ferruck/scorep_substrates_dynamic_filtering`
- *Score-P*: `http://www.vi-hps.org/projects/score-p/`
- *NAS Parallel Benchmarks*:
  `http://www.nas.nasa.gov/publications/npb.html`
- *LULESH*: `https://codesign.llnl.gov/lulesh.php`
- *uthash*: `https://troydhanson.github.io/uthash/`
- *Vampir*: `https://www.vampir.eu/`

Anhang

Trace – *LULESH* parallel

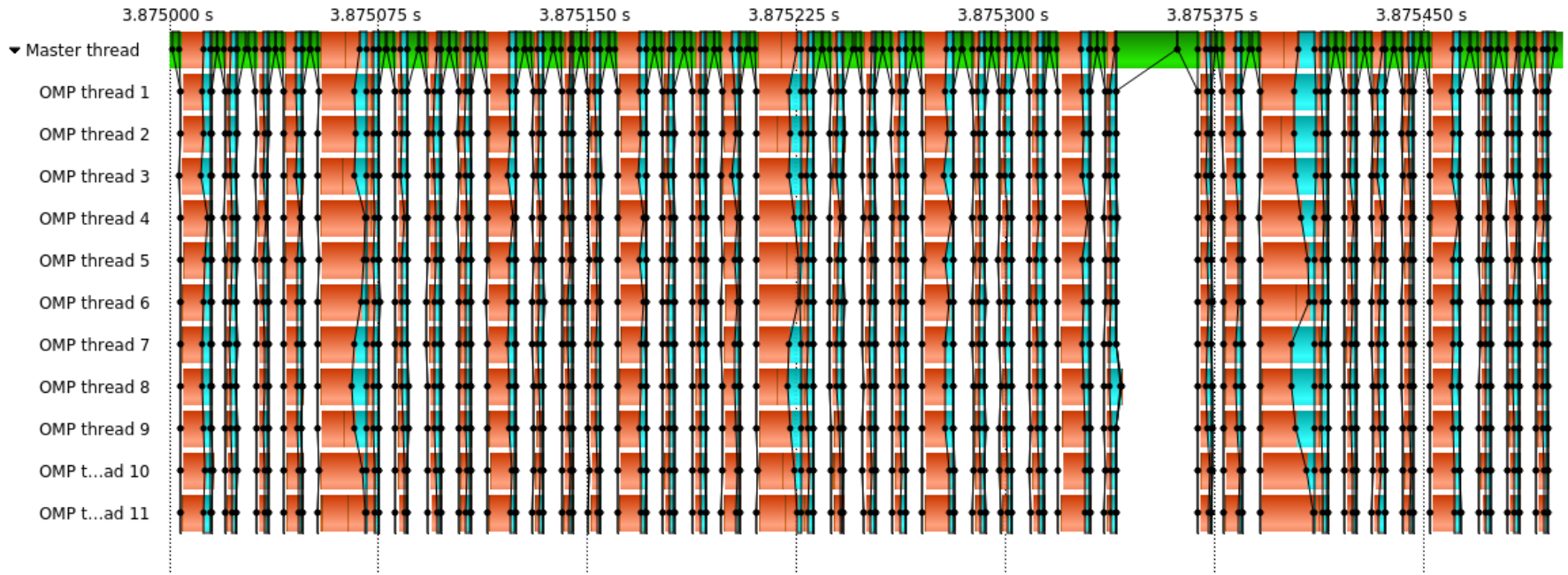


Trace – *LULESH* parallel

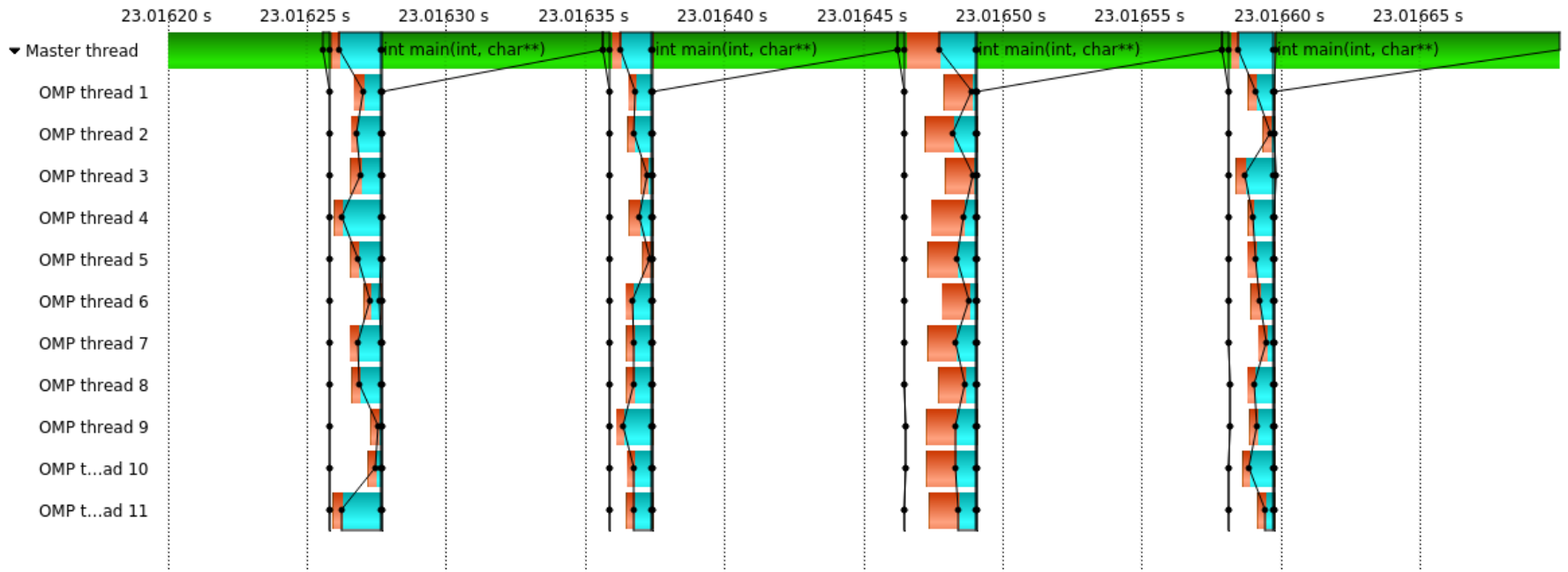


→ Reduktion der Datenmenge von ca. 1,2 *GB* auf ca. 590 *MB*

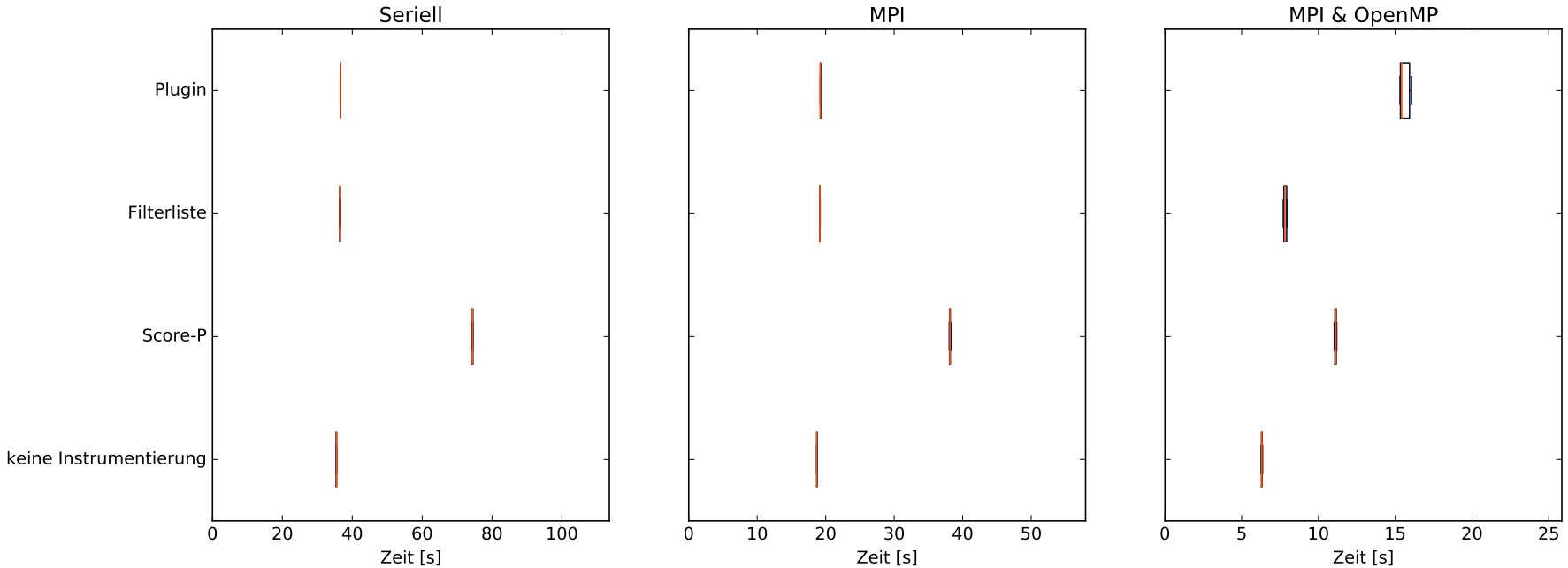
Trace – *LULESH* parallel



Trace – *LULESH* parallel

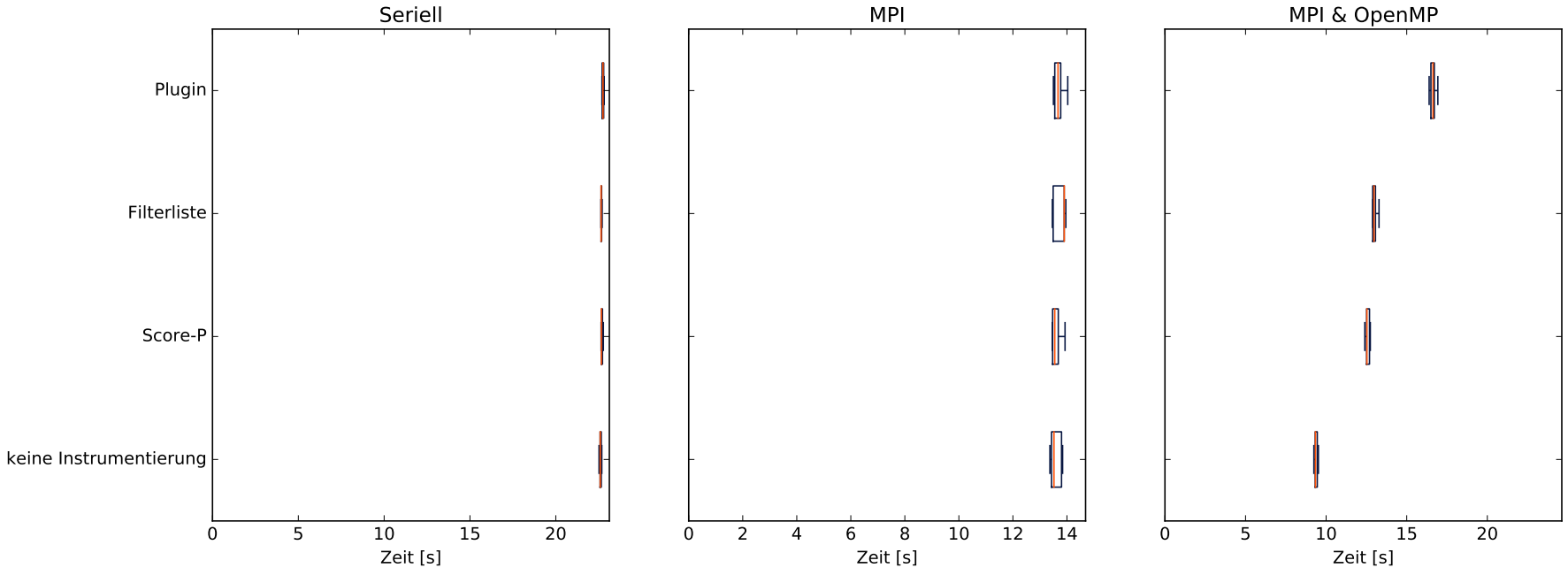


Ausführungszeiten – *NPB bt-mz A GCC*



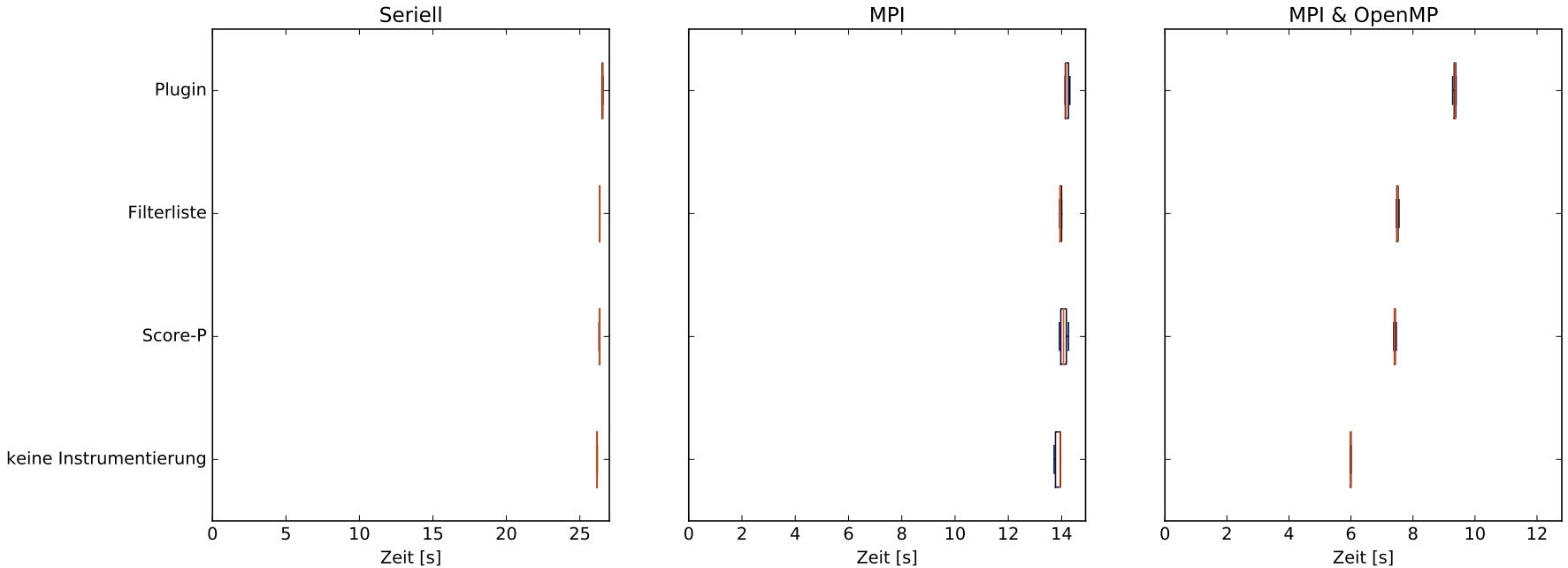
	Seriell	*MPI*	*MPI & OpenMP*
Score-P	$109,84\,\%$	$103,85\,\%$	$76,39\,\%$
Filterliste	$2,85\,\%$	$2,35\,\%$	$23,93\,\%$
Plugin	$3,21\,\%$	$2,73\,\%$	$144,37\,\%$

Ausführungszeiten – *NPB sp-mz A GCC*



	Seriell	*MPI*	*MPI & OpenMP*
Score-P	$0,31\,\%$	$0,22\,\%$	$34,15\,\%$
Filterliste	$0,22\,\%$	$2,81\,\%$	$38,87\,\%$
Plugin	$0,71\,\%$	$1,11\,\%$	$78,05\,\%$

Ausführungszeiten – *NPB lu-mz A GCC*



	Seriell	*MPI*	*MPI & OpenMP*
Score-P	$0,31\,\%$	$0,22\,\%$	$34,15\,\%$
Filterliste	$0,22\,\%$	$2,81\,\%$	$38,87\,\%$
Plugin	$0,71\,\%$	$1,11\,\%$	$78,05\,\%$

Filterlisten – *NPB bt-mz*

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

Filterlisten – *NPB sp-mz*

```
1  SCOREP_REGION_NAMES_BEGIN
2  EXCLUDE exact_solution
3  SCOREP_REGION_NAMES_END
```

Filterlisten – *LULESH*

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