Universität Potsdam

Green Computing

Sommersemester 2024

Finales Projekt: Power Capping durch RAPL

Prof. Dr. Bettina Schnor

Hanna Kretz

Matrikelnummer: 824063

14.09.2024

1 Einleitung

Das Projekt behandelt „Power Capping durch RAPL“ und ist die Prüfungsleistung der Vorlesung Green Computing bei Prof. Dr. Bettina Schnor im Sommersemester 2024. Das Projekt wurde von Hanna Kretz in einem Zeitraum von sechs Wochen durchgeführt. Betreut wurde ich durch Prof. Dr. Bettina Schnor und Max Lübke. Die Forschungsfrage des Projektes lautet: „Wie kurz kann eine Belastung auf die CPU dauern ($t <= 1ms$), dass RAPL es erkennt und aufzeichnet?“.
Der Zeitplan richtet sich nach der üblichen Struktur einer wissenschaftlichen Arbeit:

- Einarbeitung/ Theorie
- Entwurf des Experiments (inklusive Quellcode-Implementierung)
- Durchführung
- Auswertung und Diskussion

Für jeden dieser Arbeitsschritte wurden ein bis zwei Wochen eingeplant.

2 Theorie

Zur Einarbeitung wurden zunächst die Grundlagen der Programmiersprache C wiederholt.
Anschließend begann die Literaturrecherche und inhaltliche Einarbeitung. Alle Quellen sind im Literaturverzeichnis zu finden: [1], [2], [3], [4].

2.1 EMA - J. Spazier (2024)

Teil der Vorlesung Green Computing war die Veranstaltung „EMA“ [4], ein Vortrag von Johannes Spazier. Dort wird EMA (= Energy Measurement for Applications) vorgestellt. Man kann mit EMA Energiemessungen auf Anwendungsebene durchführen. Auch RAPL (= Running Average Power Limit Energy Reporting) wird als Plugin bei EMA vorgestellt. Demnach ist RAPL eine programmgesteuerte Messung des Energieverbrauchs von Intel Prozessoren. Die Software wird von Intel zur Verfügung gestellt, seit der Sandy-Bridge-Architektur (2011).
Die RAPL-Messung liefert akkumulierte Messwerte des Energieverbrauchs in Mikrojoule (µJ). Außerdem wird von EMA die Dauer der Messung aufgezeichnet und in Mikrosekunden (µs) ausgegeben. Die RAPL-Messung findet 1000 mal pro Sekunde statt, also liegt die Messfrequenz bei $f = 1ms$. Intel begründet diese grobe Messfreuquenz mit Datensicherheit, weil so sichergestellt werden kann, dass mit Hilfe der RAPL-Messung keine Daten ausgelesen oder Kryptisierungen entschlüsselt werden können. Bei der Messung wird nicht zwischen den einzelnen Cores der CPU unterschieden sondern stattdessen alle Cores zusammengefasst (siehe Abbildung 1: Powerplane 0).

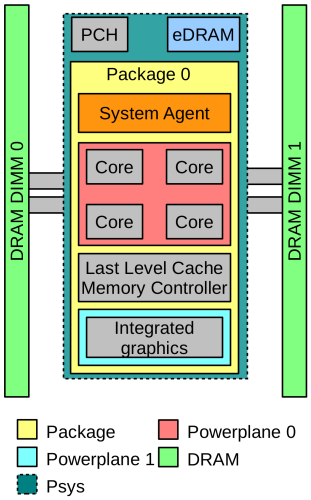


Abbildung 1: CPU structure

2.2 Measuring Energy consumption for Short Code Pathos using RAPL - Hähnel et al. (2012)

Im Paper von Hähnel et al. (2012) [2] beschäftigen sich die Autoren mit dem Thema „Measuring Energy consumption for Short Code Pathos using RAPL“. Dieses Thema ähnelt dem dieser Projektarbeit, denn die Forschungsfrage von Hähnel et al. (2012) lautet: „Ist RAPL akkurat genug um Kurzzeit-Messungen durchzuführen?“. Die Autoren entschieden sich dafür hauptsächlich Messlängen von $t > 1ms$ zu untersuchen. Ihre Schlussfolgerung und Antwort auf die Forschungsfrage ist zusammengefasst, dass RAPL nicht geeignet ist Kurzzeitmessungen durchzuführen, weil die Messfrequenz von RAPL zu gering ist für solche genauen Messungen. Die Lösung dafür, die sie im Paper vorstellen, ist eine eigene Implementierung namens "HAECER", ein Framework für Kurzzeit-Energiemessungen.

Die Abbildung 2 ist aus dem Paper von Hähnel et al. (2012) [2], das eine der Ungenauigkeiten von RAPL zeigt. Wie von Hähnel et al. (2012) getestet und analysiert ist der Abstand der RAPL-Messungen nicht immer exakt $1ms$ lang. Wären alle Punkte in der Abbildung auf einer Linie bei $cycles = 0$, wäre die Angabe der Messfrequenz exakt. Da jedoch eine deutliche Streuung und Ausreißer zu sehen sind, kann man nicht von einer so regelmäßigen Messung von RAPL jede Millisekunde ausgehen. Detailierter wird die Abbildung im Paper [2] analysiert. Wichtig für dieses Projekt ist vor allem zu wissen, dass die Messung nicht exakt jede Millisekunde stattfindet.

Eine weitere Abbildung, die weitere Probleme von RAPL verdeutlicht ist in Abbildung 3 zu sehen. Die gestrichelten und beschrifteten senkrechten Linien stellen hier die idealisierten RAPL-Messungen im Abstand von je einer Millisekunde dar. Zunächst wird der obere Teil der Abbildung betrachtet. Das Beispielprogramm „foo“ kann zu jeder möglichen Zeit zwischen den RAPL-Messungen gestartet werden. Dies

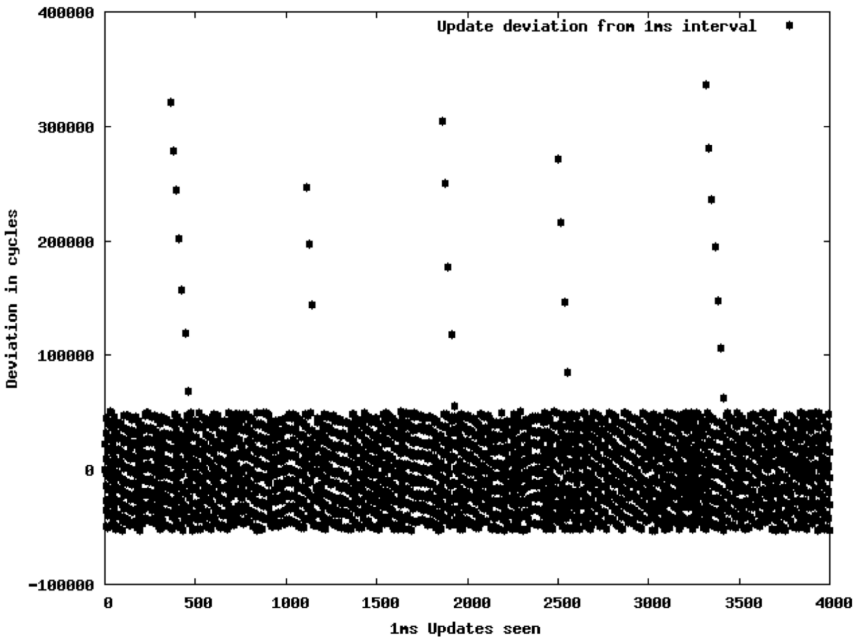


Abbildung 2: Distribution of updates relative to the specified 1*ms* interval

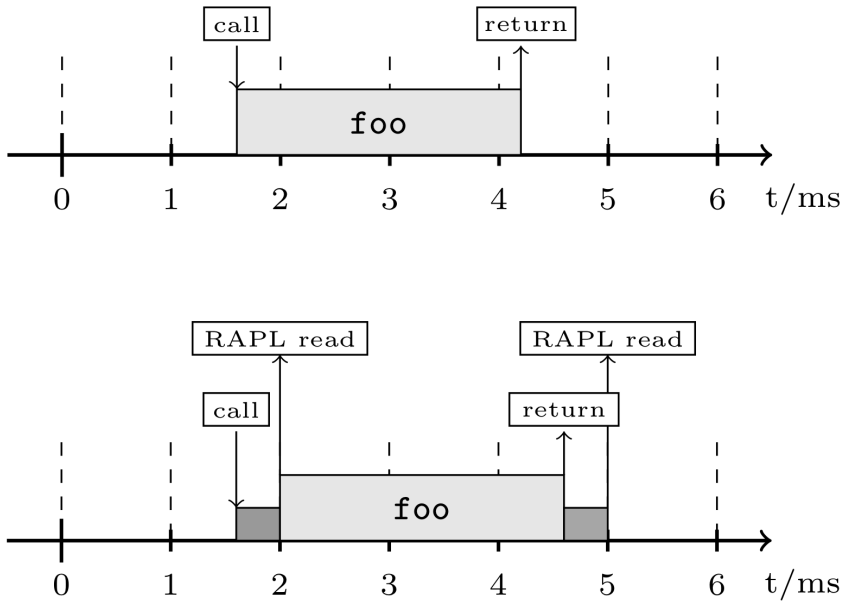


Abbildung 3: Adjusting measurements to accurately measure short-term energy consumption

hat jedoch zur Folge, dass zum Zeitpunkt der RAPL-Messung nur eine Bestandsaufnahme mitten während der Programmausführung gemessen werden kann. Verloren geht dabei wieviel Energie das Programm zwischen den Messungen verbraucht hat. Das selbe Problem besteht am Ende des Programms. Somit ist RAPL für eine solche Messung des Energieverbrauchs nicht geeignet. Besonders problematisch ist es, wenn ein Programm kürzer ist als der Messabstand von RAPL ($t < 1ms$) und das gesamte Programm zwischen den Messungen ausgeführt wird.
In ihrer eigenen Implementation HAECER haben Hähnel et al. (2012) dieses Problem gelöst, indem zum Aufruf des Programms ein Delay stattfindet bis RAPL die Energie misst, zu diesem Zeitpunkt startet das Programm. Nach Beendung des Programms gibt es wieder einen Delay, bis erneut ein Energiemesswert von RAPL aufgezeichnet wird. Durch diese Methode wird garantiert, dass es zwei Messwerte gibt (einen vor dem Programm und einen danach), die von einander abgezogen werden können um die vom Programm verbrauchte Energie zu berechnen.

3 Entwurf des Experimentes

Zum Entwurf des Experiments gehörte zunächst die genaue Definition des Ziels in Form der zu Beginn genannten Forschungsfrage.

3.1 Versuchsaufbau

Anschließend konnte der Versuchsaufbau geplant werden. Es wurde ein Programm in C geschrieben, das Lastszenarien auf der CPU verursacht und die Energie dabei aufzeichnet. Benötigt dafür wurden CPU-intensive Berechnungen, die plötzlich starten und stoppen. Dafür wurde der Zufallszahlen-Generator (Random-Number-Generator, RNG) $rand()$ gewählt. Die Intervalllängen, in denen diese Berechnungen stattfanden wurden kontrolliert und variiert. Zwischen den Berechnungen fanden Pausen (sleeptimes) statt. Es wurden elf verschiedene Intervalllängen gewählt, sie sind in Tabelle 1 aufgelistet. Jede Messung einer Intervalllänge wurde 10000 mal wiederholt um ein möglichst stabiles und belastbares Ergebnis zu erlangen.

3.2 Messmethode

Zur Messung wurde EMA verwendet, welche die RAPL-Messung überwacht und aufgezeichnet hat. Die Messung stand in der gleichen Messfrequenz statt wie es RAPL vorgibt: Ein Messwert pro Millisekunde. Die Messwerte wurden über die Länge aller 10000 Wiederholungen akkumuliert und pro Device (Package-0, core, uncore) ein Gesamtwert ausgegeben. Die Datenausgabe erfolgte im CSV-Dateiformat.

3.3 Quellcode

Zu Beginn des Programms werden Bibliotheken eingebunden (Abbildung 4). Dazu gehört $EMA.h$, zwei Standardbibliotheken von C und zwei Bibliotheken für die Kon-

Messung	Intervalllänge t [ms]	Name der Region
1	0.0	v0
2	0.1	v1
3	0.2	v2
4	0.3	v3
5	0.4	v4
6	0.5	v5
7	0.6	v6
8	0.7	v7
9	0.8	v8
10	0.9	v9
11	1.0	v10

Tabelle 1: Messdurchläufe

trollierung der Intervalllängen und sleep-times. Außerdem werden zwei Funktionsprototypen genannt.

```
1  #include <EMA.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h> // For usleep or sleep functions
5  #include <sys/time.h> // For time measurement
6
7  long long get_current_time_in_microseconds();
8  void do_work(int dur);
```

Abbildung 4: Header

Nun zu den Funktionen, welche in Abbildung 5 zu sehen sind. Eine der Funktionen ist für die Bestimmung der aktuellen Zeit in Mikrosekunden (µs) zuständig. Hierfür wird die Bibliothek *sys/time.h* und die Funktion *gettimeofday*() genutzt. Die Einzelwerte der Struktur "tv"müssen in Mikrosekunden umgerechnet und addiert werden, bevor die Funktion *get_current_time_in_microseconds*() das Ergebnis zurückgibt.
Die andere Funktion *do_work*() ist dafür zuständig Arbeit der CPU zu verursachen. Die Länge dieser Arbeit soll der in Mikrosekunden übergebenen Zeitangabe *dur* entsprechen. Der Zufallszahlen-Generator wird immer mit dem Startwert 1024 gestartet: So berechnen alle Lastszenarien die selben Werte, das Experiment hat so weniger Störvariablen und die Reliabilität wird erhöht. Anschließend wird mit Hilfe der zuvor erklärten Funktion die aktuelle Zeit gemessen und in einer Long Long Variable gespeichert. Die eigentliche Arbeit wird in einer do-while-Schleife ausgeführt, die solange läuft wie die Differenz der aktuellen Zeit und der Startzeit kleiner als die gewünschte Dauer *dur* ist. In der Schleife wird der Zufallszahlen-Generator ausgeführt und die

```
150  // FUNKTIONS
151
152  long long get_current_time_in_microseconds() {
153      struct timeval tv;
154      gettimeofday(&tv, NULL);
155      return (int)(tv.tv_sec * 1000000 + tv.tv_usec);
156  }
157
158  void do_work(int dur){
159      srand(1024);
160      long long t_before = get_current_time_in_microseconds();
161      long long t_new;
162      do{
163          rand();
164          t_new = get_current_time_in_microseconds();
165      } while(t_new - t_before < dur);
166  }
```

Abbildung 5: Functions

aktuelle Zeit gemessen. Nach jedem Schleifendurchlauf wird geprüft ob die angegebene Zeit der Intervalllänge um ist. Sobald dies der Fall ist, wird die Funktion ohne Rückgabewert beendet.

```
12  int main(int argc, char **argv)
13  {
14      int err = EMA_init(NULL); // Initialize EMA
15
16      if( err )       // Check for errors. If `err` is 0 then no error occurred.
17          return 1;
18
19      Filter *filter = EMA_filter_exclude_plugin("NVML"); // Create a filter to disable the NVML plugin.
20
21      // Initialize a region.
22      EMA_REGION_DECLARE(region_v0); // declares a region handle that holds the measurement data
23      EMA_REGION_DECLARE(region_v1);
24      EMA_REGION_DECLARE(region_v2);
25      EMA_REGION_DECLARE(region_v3);
26      EMA_REGION_DECLARE(region_v4);
27      EMA_REGION_DECLARE(region_v5);
28      EMA_REGION_DECLARE(region_v6);
29      EMA_REGION_DECLARE(region_v7);
30      EMA_REGION_DECLARE(region_v8);
31      EMA_REGION_DECLARE(region_v9);
32      EMA_REGION_DECLARE(region_v10);
```

Abbildung 6: Main 1/4

In der *main*()-Funktion (Abbildung 6) wird zunächst EMA initialisiert. Es wird ein Filter für den NVML- Plugin definiert und anschließend elf Regionen für die elf Messdurchläufe deklariert. Diese müssen anschließend erneut mit dem Filter und den einzelnen Namen definiert werden (Abbildung 7). Die Zuordnung der Namen für die Regions zu den verschiedenen Intervalllängen sind in der Tabelle 1 aufgelistet.
Danach werden zwei Variablen definiert. *Block_size* legt fest, dass jede einzelne Messung der 10000 Wiederholungen genau eine Millisekunde, also 1000 Mikrosekunden lang ist. Auch die Anzahl der Wiederholungen (10000) wird in *replication_num* festgelegt.

```
34      // EMA_REGION_DEFINE` defines the region by setting a name
35      EMA_REGION_DEFINE_WITH_FILTER(&region_v0, "V0", filter); //is an extended version that takes an optional filter argument
36      EMA_REGION_DEFINE_WITH_FILTER(&region_v1, "V1", filter);
37      EMA_REGION_DEFINE_WITH_FILTER(&region_v2, "V2", filter);
38      EMA_REGION_DEFINE_WITH_FILTER(&region_v3, "V3", filter);
39      EMA_REGION_DEFINE_WITH_FILTER(&region_v4, "V4", filter);
40      EMA_REGION_DEFINE_WITH_FILTER(&region_v5, "V5", filter);
41      EMA_REGION_DEFINE_WITH_FILTER(&region_v6, "V6", filter);
42      EMA_REGION_DEFINE_WITH_FILTER(&region_v7, "V7", filter);
43      EMA_REGION_DEFINE_WITH_FILTER(&region_v8, "V8", filter);
44      EMA_REGION_DEFINE_WITH_FILTER(&region_v9, "V9", filter);
45      EMA_REGION_DEFINE_WITH_FILTER(&region_v10, "V10", filter);
46      // Alternatively: `EMA_REGION_DEFINE(&region, "region");`
47
48
49      int block_size = 1000; // 1ms
50      int replication_num = 10000; // block_size * replication_num = 10 s
```

Abbildung 7: Main 2/4

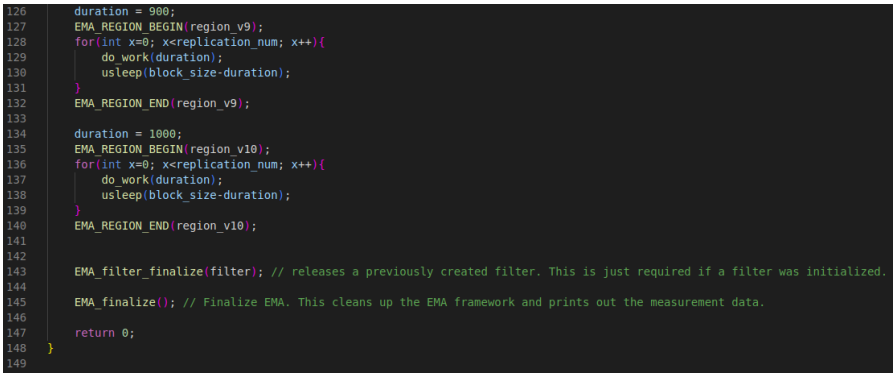
```
54      int duration = 0;
55      EMA_REGION_BEGIN(region_v0); //starts a measurement and stores the data in the region handle.
56      for(int x=0; x<replication_num; x++){
57          do_work(duration);
58          usleep(block_size-duration);
59      }
60      EMA_REGION_END(region_v0); //stops a measurement and stores the data in the region handle.
61
62      duration = 100;
63      EMA_REGION_BEGIN(region_v1);
64      for(int x=0; x<replication_num; x++){
65          do_work(duration);
66          usleep(block_size-duration);
67      }
68      EMA_REGION_END(region_v1);
```

Abbildung 8: Main 3/4

Vor jeder der elf Messungen wird die Intervalllänge in der Variable *duration* definiert (Abbildung 8). Anschließend wird mit EMA die Messung gestartet. In der for-Schleife werden 10000 Messungen durchlaufen, die je aus einem Teil Arbeit der Länge *duration* und einem Teil Pause (sleep) bestehen, sodass jeder Schleifendurchlauf der Dauer $block\_size = 1000s$ entspricht. Nach der Schleife wird die EMA-Messung beendet und ein akkumulierter Wert pro Region gespeichert.
Nach den 11 Messungen mit unterschiedlichen Intervalllängen wird zunächst der Filter beendet und anschließend auch EMA finalisiert (Abbildung 9). Dies beinhaltet auch die Datenspeicherung und -ausgabe in Form einer CSV-Datei.
Dann ist das Programm beendet.

3.4 Erwartung

Bei einer CPU-Belastung von $t = 0ms$ wird eine verbrauchte Energie von $E = 0J$ erwartet. Da das nicht ganz realistisch ist, wird erwartet, dass der Wert bei einer CPU-Belastung von $t = 0ms$ den kleinsten Energiemesswert hat. Bei kurzen (bspw. $t = 0.1ms$ oder $t = 0.2ms$ Belastung) sollte die verbrauchte Energie verglichen mit längeren Belastungen gering sein. Der Energieverbrauchswert sollte mit steigender Belastungsdauer kontinuierlich und linear steigen bis er das Maximum bei einer Belastungslänge von $t = 1ms$ erreicht.



```
    duration = 900;
    EMA_REGION_BEGIN(region_v9);
    for(int x=0; x<replication_num; x++){
        do_work(duration);
        usleep(block_size-duration);
    }
    EMA_REGION_END(region_v9);

    duration = 1000;
    EMA_REGION_BEGIN(region_v10);
    for(int x=0; x<replication_num; x++){
        do_work(duration);
        usleep(block_size-duration);
    }
    EMA_REGION_END(region_v10);


    EMA_filter_finalize(filter); // releases a previously created filter. This is just required if a filter was initialized.

    EMA_finalize(); // Finalize EMA. This cleans up the EMA framework and prints out the measurement data.

    return 0;
}

```

Abbildung 9: Main 4/4

Die Dauer der Belastung entspricht auch dem prozentualen Anteil der Belastungszeit in einem Schleifendurchlauf. Also zum Beispiel eine Belastung von $t = 0.1ms$ mit anschließender Pause / sleeptime von $t = 0.9ms$ entspricht 10 % zeitliche Belastung der CPU. Dies wiederum entspricht der Wahrscheinlichkeit, dass RAPL bzw. EMA zu diesem Zeitpunkt misst. Bei einer hohen Wiederholrate von 10000 sollte diese Wahrscheinlichkeit in den Ergebnissen erkennbar sein. Der Messwert bei $t = 0.5ms$ Belastung sollte also beispielsweise 50 % des Wertes von $t = 1ms$ Belastung entsprechen. Da ein Schleifendurchlauf der for-Schleife in $main()$ 1 ms lang sein sollte (laut $block\_size = 1000s$) und dieser 10000 Mal wiederholt wird, sollte der Messwert der Zeit von EMA 10 Sekunden entsprechen.

4 Durchführung

Es wurde zunächst ein Testlauf auf dem privaten Rechner durchgeführt. Anschließend wurde die eigentliche Messung mit naaice1 gemacht. Die technischen Daten der beiden Geräte sind in Tabelle 2 zu sehen.
Zur Kompilierung des Programms wurde der Befehl ausgeführt, der in Abbildung 10 erläutert wird.
In Abbildung 11 sind beispielhafte Rohdaten zu sehen, die CSV-Datei wird tabellarisch angezeigt. Wichtig zur Interpretation der Daten sind besonders die Spalten B, G, H und I. In Spalte B ist die Region angegeben, in die die Messwerte während den Messungen kategorisiert wurden. Die Namen geben Hinweise auf die Belastungslänge, siehe Tabelle 1. In Spalte G ist der Bestandteil der CPU dokumentiert für den die Messwerte gelten. Der in Abbildung 1 gekennzeichnete gelbe Teil Package 0 ist einer der Bestandteile, für die es Messwerte gibt. Ein weiterer Teil, der alle Cores zusammenfasst, wird „core“ genannt. Außerdem gibt es einen „uncore“, welcher nicht Teil der Auswertung sein wird. In Spalte H steht die gemessene akkumulierte Energie in Mikrojoule (µJ) und in Spalte I die gemessene Zeitlänge der Messungen in Mikrosekunden (µs).

PC	technische Daten
privat	ThinkPad-T450
	Intel® Core™ i5-5300U CPU @ 2.30GHz × 4
naaice1	Dell Precision 3660 Tower
	CPU: 1x Intel Core i9-12900 CPU @ 2.40GHz, 8C+8c/24T
	Memory: 128GiB DIMM DDR5 Synchronous Registered (Buffered) 3600 MHz (0,3 ns)
	Network:
	- NetXtreme BCM5720 2-port Gigabit Ethernet
	- NVIDIA MCX623106AN-CDAT 2-port [ConnectX-6 Dx]

Tabelle 2: Technical data of the computers

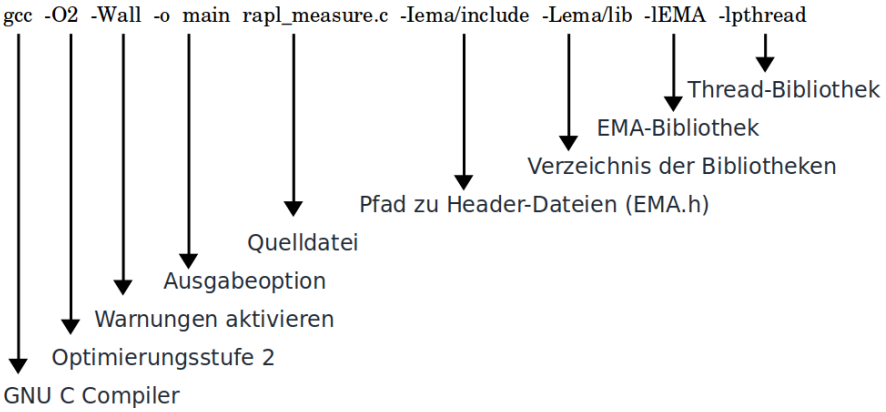


Abbildung 10: compiler command

	A	B	C	D	E	F	G	H	I
1	thread	region_idf	file	line	function	visits	device_name	energy	time
2	0	v0	rapl_measure.c	51	main	1	CPU-0.package-0	122131889	10602552
3	0	v0	rapl_measure.c	51	main	1	CPU-0.core	17888627	10602605
4	0	v0	rapl_measure.c	51	main	1	CPU-0.uncore	1892	10602687
5	0	v1	rapl_measure.c	52	main	1	CPU-0.package-0	121547480	10589681
6	0	v1	rapl_measure.c	52	main	1	CPU-0.core	17634598	10589796
7	0	v1	rapl_measure.c	52	main	1	CPU-0.uncore	305	10589790
8	0	v2	rapl_measure.c	53	main	1	CPU-0.package-0	121352778	10588615
9	0	v2	rapl_measure.c	53	main	1	CPU-0.core	17307389	10588690
10	0	v2	rapl_measure.c	53	main	1	CPU-0.uncore	1343	10588688
11	0	v3	rapl_measure.c	54	main	1	CPU-0.package-0	121606073	10588523
12	0	v3	rapl_measure.c	54	main	1	CPU-0.core	17665605	10588595
13	0	v3	rapl_measure.c	54	main	1	CPU-0.uncore	1831	10588595
14	0	v4	rapl_measure.c	55	main	1	CPU-0.package-0	121645196	10588619
15	0	v4	rapl_measure.c	55	main	1	CPU-0.core	17548356	10588690
16	0	v4	rapl_measure.c	55	main	1	CPU-0.uncore	1709	10588703
17	0	v5	rapl_measure.c	56	main	1	CPU-0.package-0	121929498	10588609

Abbildung 11: raw data

5 Auswertung und Diskussion

5.1 Zeitmesswerte

Die gemessene Zeit der 11 Messungen war im Mittel $t = 10,5672s$ mit einer Standardabweichung von $s = 0,039s$. Die Messkurve ist in Abbildung 12 zu sehen, man beachte die Beschriftung der y-Achse. Man sieht, dass alle 11 Werte deutlich über den erwarteten 10 Sekunden liegen und diese Abweichung kaum schwankt. Dies ist auch an der geringen Standardabweichung ersichtlich. Zwischen v7 und v8 gibt es eine Abfall um etwa 0,8 s. Der Grund für diese Veränderungen ist nicht ersichtlich. Die Messwerte weichen alle um etwa 0,5 ms von der Erwartung ab.
Auch bei der Betrachtung jedes Schleifendurchlaufs und der Einzelmessungen ist die Abweichung zu sehen, denn hier entspricht der Mittelwert $t = 1056s$ mit einer Standardabweichung von $s = 3,98s$. Es gibt somit eine regelmäßige zeitliche Verzögerung der Einzeldurchläufe um mindestens 52 µs. Als Ursache dafür werden verschiedene Punkte erkannt: Zum Beispiel die Funktionsaufrufe von *do_work*() und *get_current_time_in_microseconds*(). Verschiedene Teile des Codes benötigen Zeit, die bei der Erwartung nicht beachtet wurde.

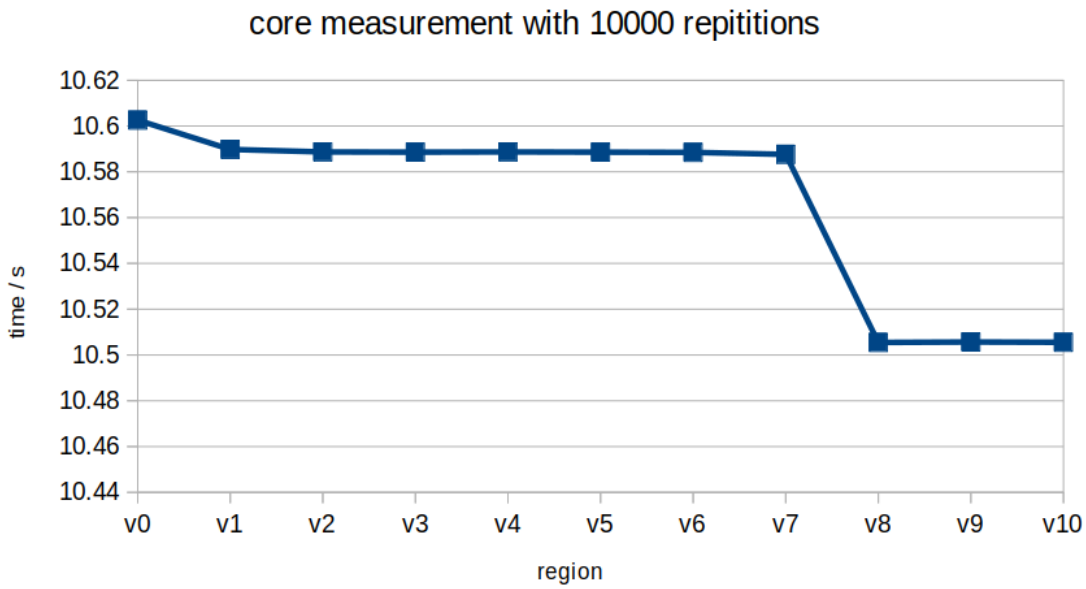


Abbildung 12: time measurement

5.2 Energiemesswerte

5.2.1 Einzelmessung

Bei einer einzelnen Messung ohne die 10000 Wiederholungen (Abbildung 13) ist zu sehen, dass die Energiemesswerte stark schwanken. Sowohl bei den Werten von package-

0 als auch von core ist keine Regelmäßigkeit zu erkennen. Erstaunlich ist außerdem, dass der Energiewert bei v0 weder $E = 0J$ ist noch kommt er den 0 J Nahe. Viele weiter Werte liegen in einem ähnlichen Bereich wie der von v0, sogar der letzte Wert bei v10 liegt bei ungefähr $E = 10mJ$. Es gibt zwei Werte (v3 und v7), welche deutlich höher als alle anderen liegen. Vermutlich hat RAPL bei diesen Messungen zur Zeit der Belastung gemessen, das war jedoch Zufall.

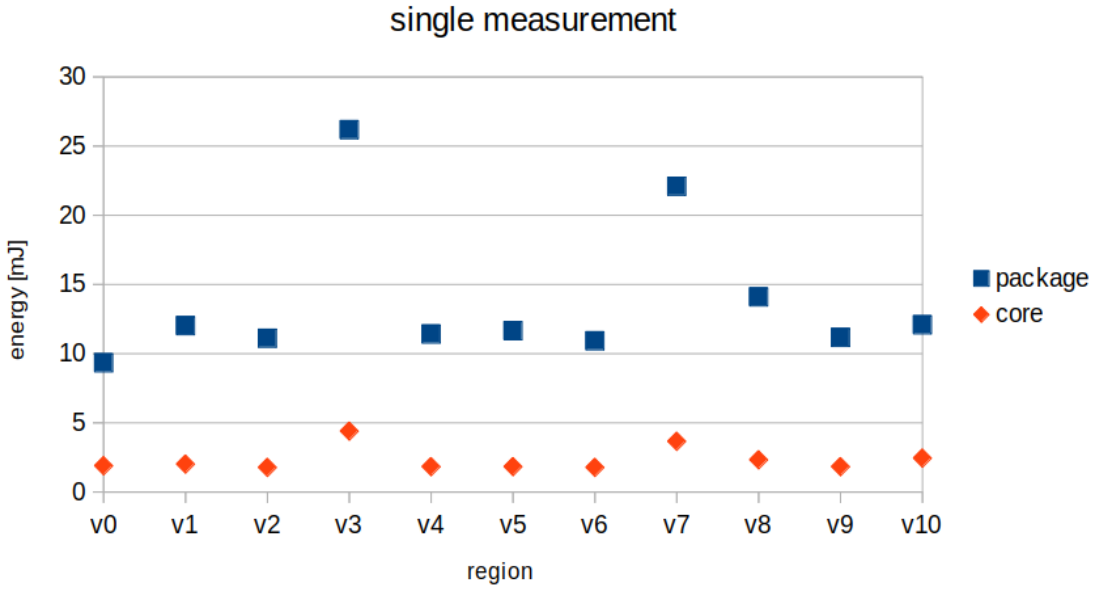


Abbildung 13: single energy measurement

5.3 Messwerte mit Mittelung über 10.000 Wiederholungen

Bei der Messung mit 10000 Wiederholungen (Abbildung 14) sind die Ergebnisse viel weniger gestreut und es gibt eine Regelmäßigkeit zu erkennen. Auf den ersten Blick ist sofort zu sehen, dass die Ergebnisse zweigeteilt sind.
Im ersten Teil von v0 bis v5 liegen alle Punkte auf einer waagrechten Geraden, scheinen sich nicht zu unterscheiden. Es ist keine Steigung zu erkennen. Dies trifft auf beide betrachteten Teile der CPU (package-0 und core) zu. Betrachtet man die Messwerte v0 - v6 genauer (Abbildung 15 und Abbildung 16), erkennt man sowohl bei package-0 als auch bei core eine zufällige Streuung. Bei package-0 ist der Wert bei v0 sogar der Höchste dieser 6 Werte. Die Streuungen sind so gering, dass man sie bei der Abbildung mit allen Messwerten (Abbildung 14) nicht erkennt.
Im zweiten Teil von v6 bis v10 steigen die Werte signifikant, es ist eine eindeutige Steigung zu erkennen. Das Maximum wird bei v10 erreicht. Dieses Verhalten, dass erst ab v6 eine Veränderung des Energiewertes erkannt wird, ist unerwartet und hat keinen ersichtlichen Grund. Es zeigt jedoch, dass vor allem unter einer Belastungslänge von

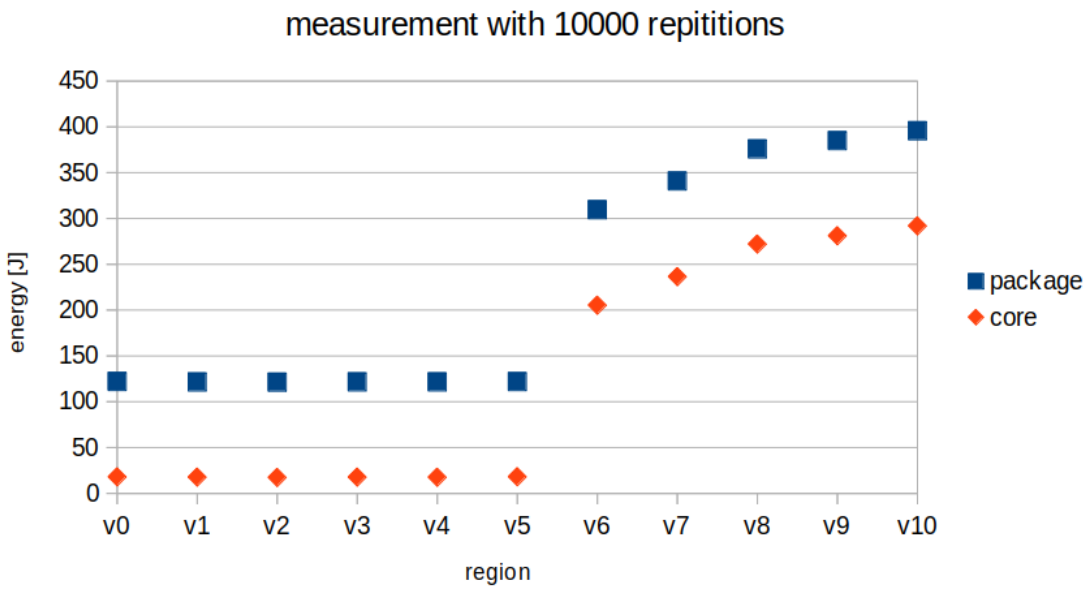


Abbildung 14: energy measurement

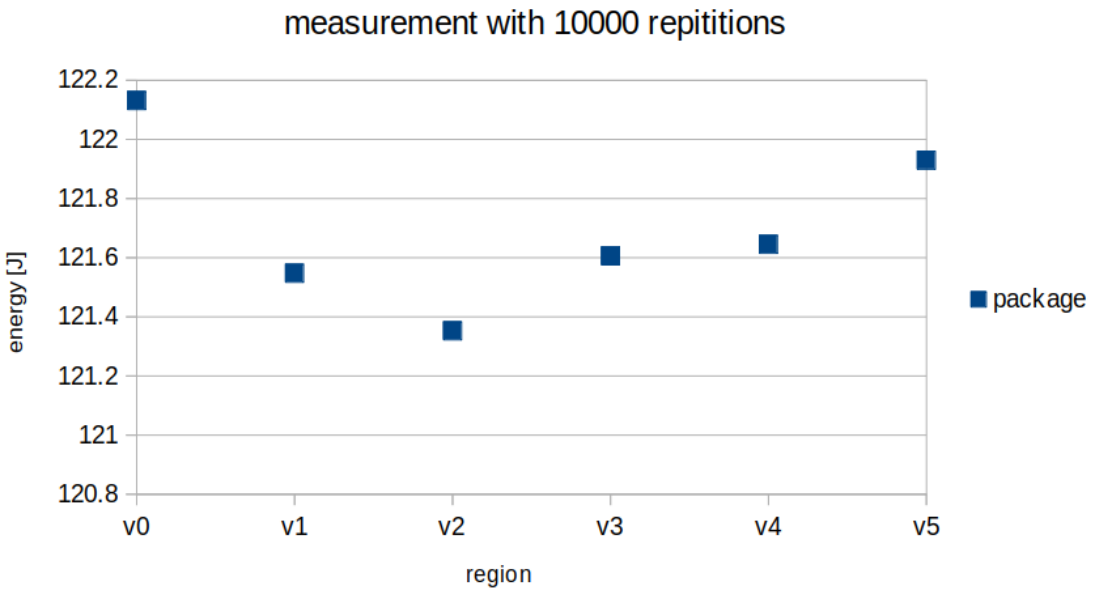


Abbildung 15: energy measurement: v0-v5 package-0

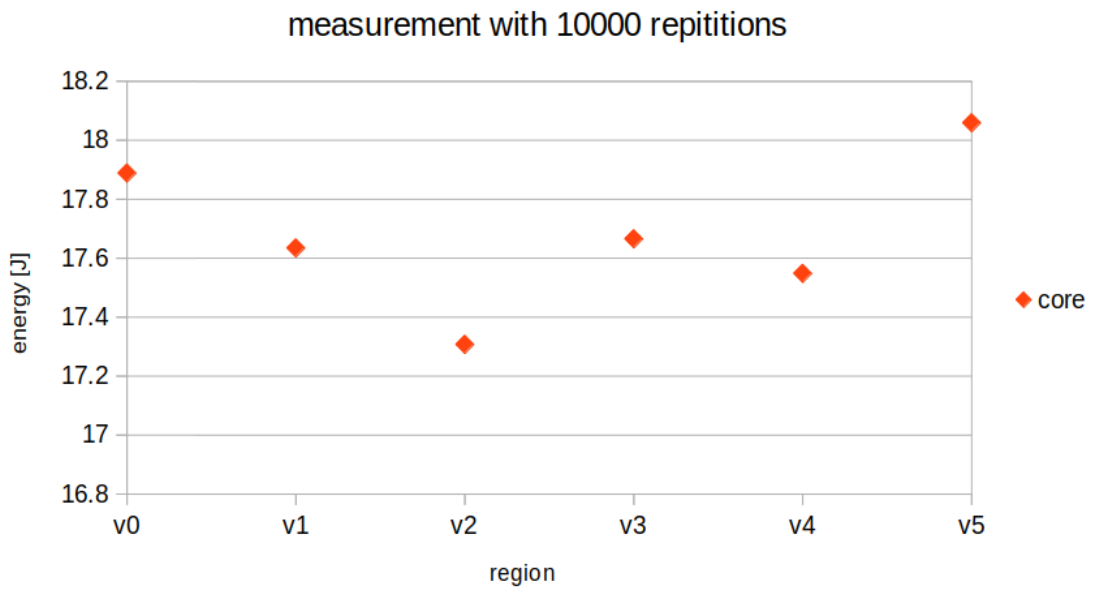


Abbildung 16: energy measurement: v0-v5 core

	package-0	core
Min. E [J]	121,352	17,307
Max. E [J]	395,768	291,844
Mean E [J]	230,653	126,620
Standard deviation [J]	127,179	127,187
Mean v0-v5 E [J]	121,702	17,68

Tabelle 3: Analysis values

$t = 0.6ms$ die Messung von RAPL dem Zufall überlassen ist. Ab $t = 0.6ms$ steigt nicht nur die Wahrscheinlichkeit weiter, dass RAPL zum Zeitpunkt der Belastung misst, sondern es steigt auch die Belastbarkeit der Messwerte.
In Tabelle 3 sind einige Analysewerte aufgelistet. Die Werte sind jedoch nicht besonders aussagekräftig.

5.4 Beantwortung der Forschungsfrage

Wie kurz kann eine Belastung auf die CPU dauern ($t <= 1ms$), dass RAPL es erkennt und aufzeichnet? Ab einer Belastungsdauer von $t = 0.6ms$ scheint RAPL recht verlässlich einen Energieverbrauch zu messen.

5.5 Fehleranalyse und Reflexion

Die Messwerte dieses Versuchs sind durch die hohe Wiederholrate belastbar. Störvariablen können jedoch nicht ausgeschlossen werden. Die drei Gütekriterien Objektivität, Reliabilität und Validität wurden so wie es möglich war erfüllt.
Die Messung mit EMA, wie sie hier durchgeführt wurde, ermöglicht keine detaillierte Interpretation. Detailliertere Datensätz wie beispielsweise ein Messwertverlauf der Energie (pro Intervalllänge) oder Einzelwerte statt akkumulierte Werte würden deutlich mehr Interpretationsspielraum und mehr Hinweise zum tatsächlichen Verhalten von RAPL ermöglichen. Bei einer weiterführenden Untersuchung wäre eine solche Verwendung von EMA sinnvoll.
In Einzelmessungen sind die RAPL-Werte bei Kurzmessungen wie in diesem Projekt nicht belastbar.
Zur Reflexion der Projektarbeit ist anzumerken, dass die Forschungsfrage nicht als erstes sondern erst kurz vor der Durchführung festgelegt wurde. Dies entspricht leider nicht guter wissenschaftlicher Arbeit. Außerdem gab es Verzögerungen durch Probleme bei der Arbeit mit dem externen Rechner naaice1. Dies lag hauptsächlich an wenig Erfahrung, die durch diese Arbeit zu Teilen aufgeholt werden konnte. Die genannten Punkte führten dazu, dass der zuvor festgelegte Zeitplan nicht eingehalten wurde.

6 Zusammenfassung

In diesem Projekt wurde ein Versuch gemacht, der untersucht wie genau die Messung mit RAPL ist und wie kurz eine CPU-Belastung sein kann, dass RAPL es erkennt und aufzeichnet. Ab einer Belastungsdauer von $t = 0.6ms$ scheint RAPL recht verlässlich einen Energieverbrauch zu messen. Mit dieser Angabe sollte man dennoch vorsichtig umgehen, denn gerade bei Einzelmessungen sind die RAPL- Werte nicht belastbar. Trotz möglicher Störvariablen sind die Messergebnisse dieses Projekts belastbar.

Literatur

[1] Running average power limit energy reporting, cve-2020-8694, cve-2020-8695 / intel-sa-00389, 2020. Available: `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00389.html`.

[2] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using rapl. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[3] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018.

[4] Johannes Spazier. Präsentation "09-ema", 2024. Green Computing.