Karlsruher Institut für Technologie

Faires Scheduling unter Beachtung von AVX-512-Frequenzeffekten

Bachelorarbeit
von

Philipp Machauer

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Mathias Gottschlag, M. Sc.

Bearbeitungszeit: 11. Mai 2020 – 10. September 2020

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde, sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 10. September 2020

Abstract

Intel introduced the *Advanced Vector Extensions (AVX)* to their processors for making complex calculations faster. Those instructions lead to a higher power usage, thus producing more heat on the processor. In order to prevent the power supply from being overloaded and the processor from overheating, the core frequency gets reduced. After some time past the last AVX instructions where executed the processor resets its frequency to the regular level.

When the processor detects the execution of an AVX instruction it starts a timer. Everytime another AVX instruction gets executed this timer is being reset and restarted. Once the timer expires there was no AVX code recently and therefore the frequency is set back to its normal level. Using this procedure one can be relatively sure that no AVX instructions will be executed in the near future and that a resetting of the frequency is reasonable.

The disadvantage of this technique is the reduction of the CPU frequency for processes executed directly after an process which used AVX instructions. The normally used *Completely Fair Scheduler (CFS)* in a Linux environment calculates the fairness only considering the execution time of a process. Therefore processes which got executed while the frequency was reduced are treated exactly like a process which ran under full speed. This leads to a disadvantage for those processes.

In order to face this problem this thesis detects those discriminated processes. According to the duration of execution under reduced frequency the measured execution time of this process gets reduced afterwards. This leads to a preferred schedule of these processes and therefore to a fairer share of execution time on the processor.

Implementing this solution was done by modifying the Linux kernel in a way that it can detect those processes. This was done by measuring the reduced runtime of the previous process on every call of the scheduler.

This solution showed a fairer distribution of execution time to processes that ran under reduced frequency. That resulted in a speedup of those processes upto 11 %. Under completely optimal and fair conditions a speedup of upto 25 % would be possible. This leads to the conclusion that this thesis provides a working proof of concept but further development has to be done in order to optimize the code and eliminate other occuring sideeffects.

Inhaltsverzeichnis

Kapitel 1

Einführung

Zur effizienteren Durchführung von Berechnungen existieren auf x86 Prozessoren *Advanced Vector Extensions (AVX)* [2, Vol. 1, Kapitel 5.13]. Die Problematik in der Nutzung dieser AVX-Befehle besteht darin, dass diese zu einem hohen Stromverbrauch und damit zu hoher Hitzeentwicklung am Prozessor führen. Um eine Überhitzung des Prozessors und eine Überlastung der Stromversorgung zu verhindern, wird während der Ausführung des AVX-Codes die Taktfrequenz reduziert. Sobald kein AVX-Code mehr auf dem Prozessor ausgeführt wird, setzt dieser die Frequenz wieder hoch, um mit regulärer Taktrate weiterzulaufen. [1]

Durch einen Timer wird erkannt, ob weiterhin AVX-Code ausgeführt wird. Der Prozessor startet diesen Timer, sobald AVX-Befehle ausgeführt werden. Weitere Ausführungen von AVX-Befehlen setzen diesen Timer zurück und starten diesen neu. Nach Ablauf des Timers wird die Frequenz wieder hoch gesetzt, da für die Dauer des Timers keine AVX-Befehle ausgeführt wurden. Hierdurch ist die Wahrscheinlichkeit niedrig, dass in Kürze wieder AVX-Befehle ausgeführt werden. [27] [1, Kapitel 2.2.3]

Das Problem hierzu liegt in folgendem Szenario: zunächst wird ein Prozess A ausgeführt. Dieser nutzt AVX-Befehle, daher setzt der Prozessor die Frequenz herunter. Wird nun ein Prozess B ausgewählt, der keine AVX-Befehle ausführt, muss dieser unter Umständen zunächst mit geringerer Frequenz laufen bis der Timer abgelaufen ist. Der im Linux Kernel meist verwendete *Completely Fair Scheduler (CFS)* versucht, wie der Name schon sagt, die Prozesse möglichst fair auf dem Prozessor auszuführen. Hierbei lässt er jedoch die Frequenz des Prozessors außer Acht. Dies führt zu der Situation, dass Prozesse benachteiligt werden, die nach AVX-Befehlen ausgeführt werden, da sie mit geringerer Taktrate (und damit langsamer) rechnen. Der CFS beruht jedoch allein auf der Zeit, die der Prozess auf dem Prozessor ist und kann daher auf diese verringerte Taktrate nicht reagieren. [28]

In dieser Arbeit wird der Ansatz behandelt, dass die Auswirkung der Frequenzreduzierung auf nachfolgende Prozesse berechnet werden soll. Anhand dessen wird die Ausführungszeit des benachteiligten Prozesses reduziert. Hierbei wird die Funktionsweise des CFS genutzt: da der CFS Fairness anhand der Ausführungszeit berechnet [3], wird dieser den benachteiligten Prozessen zukünftig mehr Zeit auf dem Prozessor einräumen. Hierdurch lassen sich Beschleunigungen von 11 % der benachteiligten Prozesse erzielen. Unter optimalen, und damit fairen, Bedingungen müssten diese 25 % schneller ausgeführt werden. Hiermit lässt sich zeigen, dass durch diesen Ansatz benachteiligte Prozesse ein faireres, wenn auch noch nicht optimales, Scheduling erfahren.

In der Praxis könnte ein solcher Ansatz zum Beispiel beim maschinellen Lernen zum Einsatz kommen. Zum Trainieren dieser Algorithmen kommen häufig AVX-512-Befehle zum Einsatz, um die Berechnungen effizienter durchführen zu können. Hierbei kommt es jedoch, unter anderem durch die Frequenzreduzierung bei der Ausführung von AVX-Code, zur Ausbremsung anderer Prozesse. Für solche Anwendungsfälle könnte ein angepasster Scheduler ein faireres Ausführen der Prozesse ermöglichen. [21]

Kapitel 2

Technische Hintergründe und verwandte Arbeiten

Um die Implementierung dieser Arbeit besser verstehen zu können, ist Wissen über den verwendeten Scheduler und angewandte Techniken notwendig. In diesem Kapitel werden diese Hintergründe genauer erläutert und bestehende Ansätze zu dieser Arbeit abgegrenzt.

2.1 CFS

Der *Completely Fair Scheduler (CFS)* ist der Scheduler, der meist im Linux Kernel verwendet wird [3]. Hierbei versucht dieser, wie der Name bereits impliziert, möglichst fair die Rechenzeit auf Prozesse zu verteilen. Hierzu teilt dieser die zur Verfügung stehende Ausführungszeit in Zeitabschnitte (*Timeslice*) ein. Diese Zeitabschnitte werden in $1/N$ Teile unterteilt, wobei N die Anzahl der laufenden Prozesse darstellt. Auf diese Teilstücke verteilt der CFS die Prozesse, wobei die Teilstücke, je nach der im Folgenden beschriebenen Gewichtung der Prozesse, unterschiedlich lang sein können. Durch dieses Verfahren ist eine faire Verteilung der Rechenleistung möglich. [4]

Durch Priorisieren von Prozessen kann diesen anteilhaft mehr Ausführungszeit pro Zeitabschnitt zugesprochen werden. Dies führt jedoch zu einer Reduzierung der Ausführungszeit der restlichen Prozesse. Erreicht werden kann diese Priorisierung durch das Setzen des `Nice`-Wertes, wobei ein höherer Wert eine niedrigere Priorität repräsentiert. Hierdurch werden die Prozesse unterschiedlich gewichtet und erhalten daher auch unterschiedlich viel Rechenzeit zugeschrieben. [3] [4]

Mit der Verwendung von `nice` ist jedoch nur eine statische Gewichtung der Prozesse möglich. Die Gewichtung erfolgt nicht in Abhängigkeit anderer Prozesse. Hierdurch kann dies nicht verwendet werden, um dynamisch auf den Einfluss anderer Prozesse zu reagieren. Da in dieser Arbeit jedoch die Gewichtung der Prozesse in Abhängigkeit des zuvor ausgeführten Prozesses vorgenommen wird, ist `nice` hierzu nicht verwendet worden. [6]

Um nachzuverfolgen welcher Prozess bereits wie viel Ausführungszeit erhalten hatte, addiert der Scheduler diese Zeit pro Prozess, durch Skalierung mit der Anzahl der aktiven Prozesse und unter Berücksichtigung der Gewichtung, in `vruntime` [3]. Nach jedem Scheduling-Vorgang wird der soeben ausgeführte Prozess anhand seiner `vruntime` in einen Rot-Schwarz-Baum einsortiert. Hierbei stehen die Prozesse mit der geringsten `vruntime` ganz links im Baum, wohingegen die Prozesse mit einer hohen `vruntime` rechts im Baum stehen. Als nächster auszuführender Prozess wird der am weitesten links stehende Prozess im Baum ausgewählt, sodass immer die Prozesse mit geringster Ausführungszeit ausgeführt werden können. [3] [4]

Um häufiges Hin- und Herwechseln zwischen zwei Prozessen mit ähnlicher `vruntime` zu vermeiden, muss ein Schwellwert überschritten werden, wodurch die Anzahl an Kontextwechseln reduziert wird [4] [8]. Dies führt auch zu einer Reduzierung des Overheads, den die Kontextwechsel erzeugen. Gleichzeitig führt die Nutzung dieses Schwellwerts jedoch auch zu der Situation, dass zwei zur gleichen Zeit gestartete Prozesse, trotz gleicher Priorisierung, eine unterschiedliche `vruntime` haben können. Dies ist darin begründet, dass durch den Schwellwert der eine Prozess dem anderen zunächst voraus eilt, bevor die Prozesse gewechselt werden und der andere die Zeit wieder aufholen kann. [4]

Der CFS berechnet Fairness nur unter Berücksichtigung der Ausführungszeit und gesetzter Prioritäten [3]. Es ist jedoch möglich faires Scheduling auch in Bezug auf den Energieverbrauch zu erzielen. Ansätze hierzu sind in Abschnitt 2.3 beschrieben.

2.2 Traps

Traps treten auf, wenn der Prozessor durch Ausführung eines Befehls in einen ungültigen Zustand kommt [11]. Dies kann zum Beispiel passieren, indem Befehle ausgeführt werden, die auf dem Prozessor deaktiviert wurden [7]. Dieser Fall wird in dieser Arbeit verwendet. Wie dies im Zusammenhang mit AVX steht, ist in Kapitel 2.4 genauer erläutert.

Tritt ein Fehler (also eine Trap) auf, so wird das aktuell ausgeführte Programm unterbrochen und der Kernel aufgerufen [11]. Hierbei werden im Kernel entsprechende Routinen aufgerufen, um diesen Fehler aufzulösen. Diese Abhandlungen

sind in der Datei `traps.c` pro Architektur abgelegt [15]. So können, wie in diesem Fall, bei der Verwendung eines deaktivierten Befehls, Flags gesetzt und im Anschluss die Ausführung dieser Befehle freigegeben werden.

Nachdem die Fehlerbehandlung abgeschlossen ist, wird wieder zurück zur Ausführung des Prozesses gewechselt. Im Gegensatz zu Abbrüchen ist dies bei Verwendung von Traps möglich. Hierdurch kann der Kernel den fehlerhaften Zustand auflösen und im Anschluss kann die Ausführung des Programms fortgesetzt werden. [11]

2.3 DVFS

Dynamic voltage and frequency scaling (DVFS) wird auf heutigen Prozessoren eingesetzt, um deren Energieverbrauch der aktuellen Nutzung anpassen zu können. Würden die Prozessoren immer auf maximaler Leistung laufen, obwohl die Leistung gar nicht abgerufen wird, würde dies zu einer hohen Wärmeentwicklung und einem hohen Stromverbrauch führen. Um einerseits die Überhitzung des Prozessors zu verhindern und andererseits nicht unnötig Strom zu verbrauchen, wird DVFS eingesetzt, um die Spannung und die damit verbundene Taktfrequenz des Prozessors zu steuern. Hierdurch kann dynamisch die Leistung des Prozessors an die benötigte Rechenleistung angepasst werden. [30] [9]

Da jeder Wechsel der Frequenz jedoch einen Overhead mit sich führt, wird mit Hilfe einer Verzögerung die aktuelle Frequenz etwas länger gehalten, um nicht direkt bei jeder Änderung der Belastung des Prozessors die Frequenz anpassen zu müssen. Dies hat eine glättende Wirkung und reduziert den Overhead durch zu häufiges Wechseln der Frequenz. Jedoch werden auch Prozesse mit geringerer Frequenz ausgeführt bis diese angepasst wird. [34]

Hier gibt es bereits Forschungsarbeit, die diese Verzögerung minimieren bzw. den Zeitpunkt für das Ändern der Frequenz intelligent auswählen will. Hierzu können Vorhersagen des Betriebssystems berechnet werden. Alternativ können Entwickler in deren Anwendungen Mechanismen implementieren, sodass diese dem Betriebssystem Hinweise geben können wie viel Rechenleistung die Anwendung in nächster Zeit benötigt. Hieraus könnte intelligent die Frequenz gesetzt werden, um Verzögerungen durch ungünstig gesetzte Frequenzen zu vermeiden. [30]

Dieser Ansatz versucht jedoch lediglich präventiv einzugreifen [30] und berücksichtigt keine entstandene Benachteiligung von Prozessen, um diese zukünftig auszugleichen.

2.3.1 Skalierung der Zeitabschnitte

Die Zeitabschnitte der Prozesse zu skalieren ist eine weitere Möglichkeit, um der geringeren Ausführungsfrequenz durch DVFS entgegen zu wirken. Hierbei wird bei jeder Änderung der Frequenz die Gewichtung und damit die Verteilung der Zeitabschnitte unter den aktiven Prozessen neu berechnet. Hierbei wird berücksichtigt, wie stark welcher Prozess durch die Frequenzänderung beeinträchtigt wird. Dies berücksichtigt auch, dass Prozesse, die viel mit dem Speicher interagieren, weniger von solchen Frequenzänderungen betroffen sind, da hier die Verzögerung durch den Speicher unabhängig der aktuellen Ausführungsfrequenz des Prozesses ist. [31]

Die Beeinträchtigungen durch DVFS können mit Hilfe dieses Ansatzes behoben werden, jedoch wird hierbei nicht unterschieden ob der Prozess selbst die Frequenzänderung verursacht hat oder ob dieser unverschuldet beeinträchtigt wurde [31]. Diese Arbeit prüft hingegen ob ein Prozess AVX-Code (vgl. Abschnitt 2.4) ausgeführt hat und hierdurch die Frequenzänderung selbst verursacht hat, oder ob ein Prozess unverschuldet verlangsamt ausgeführt wird.

2.3.2 Energieverbrauch der Prozesse

Der CFS berechnet die Fairness für Prozesse lediglich mit Hilfe deren Ausführungszeit und der festgelegten Priorität [3]. Es wird hierbei keine Rücksicht auf ausgeführte Befehle oder den Energieverbrauch genommen. Wie andere Arbeiten bereits gezeigt haben, ist es jedoch möglich, die Fairness auch mit Hilfe der genutzten Energie der Prozesse zu berechnen. Dies ermöglicht es, ressourcensparenden Prozessen mehr Rechenzeit zuzuweisen, da Prozesse, die einen hohen Energieverbrauch haben, in gleicher Zeit mehr Leistung erhalten haben. Dieser Ansatz kann auch bei mobilen System Anwendung finden, bei denen stark auf den Energieverbrauch geachtet werden muss und daher dies der limitierende Faktor ist. [36]

Problematisch an diesem Verfahren ist bisher die Echtzeiterkennung des tatsächlichen Energiebedarfs der Prozesse. Dies wäre notwendig, um faires Scheduling anhand des Bedarfs an Energie zu ermöglichen. In der beschriebenen Arbeit ist dieser Punkt jedoch noch offen. [36]

Teil dieser Arbeit ist es, dass anhand der verursachten Frequenzreduzierung erkannt wird, welche Prozesse besonders viel Energie verbrauchen.

2.3.3 Limitierter Energieverbrauch des Systems

Die Frequenzreduzierung durch DVFS wird auf Grund von Limitierungen im Energieverbrauch des Prozessors und dessen Hitzeentwicklung durchgeführt. Ein ähnlicher Ansatz besteht darin, dass der Energieverbrauch des gesamten Systems begrenzt ist. Dieser Fall findet zum Beispiel in Rechenzentren Anwendung, bei denen die zur Verfügung stehende Energie pro Serverrack begrenzt ist. Hierbei ist es wichtig, dass die Server die gesetzten Limits nicht überschreiten, um eine Überlastung der Versorgungsleitungen oder das Auslösen einer Sicherung zu verhindern. Unter diesen Bedingungen soll ebenfalls faires Scheduling möglich sein. Hierzu ist es nötig, die aktuelle Frequenz an die zur Verfügung stehende Leistung dynamisch anzupassen. Um bei diesen Schwankungen trotzdem ein faires Scheduling zu ermöglichen, wird entstehender Geschwindigkeitsverlust und -gewinn durch Änderungen der Frequenz bei betroffenen Prozessen in zukünftigen Ausführungen berücksichtigt. [32]

Das Problem dieses Ansatzes besteht darin, dass die Fairness nur in Bezug auf Frequenzreduzierungen sichergestellt wird, die durch das Erreichen der Energielimits verursacht wird. Hierbei wird nicht der Einfluss der Prozesse auf die Frequenz berücksichtigt. Damit verbundene Beeinträchtigungen anderer Prozesse können daher nicht behoben werden. [32]

In dieser Arbeit wird beschrieben, wie diese Frequenzänderung durch Prozesse und die damit verbundene Beeinträchtigung untereinander festgestellt werden kann, um im nächsten Schritt die Fairness wiederherzustellen.

2.3.4 Minimierung des Energieverbrauchs

Der Linux Kernel ist in der Lage bei asymmetrischen Prozessortopologien den Energieverbrauch durch intelligentes Zuteilen der Prozesse an die verschiedenen Kerne zu minimieren [17]. Hierzu wird errechnet, welcher Kern wie viel Energie aktuell verbraucht und um welchen Faktor dieser Verbrauch steigen würde, wenn der auszuführende Prozess auf diesem Kern gestartet wird. Im Anschluss wird der Kern ausgewählt, bei dem dies zur geringsten Steigerung des Gesamtverbrauchs des Systems führt. Dies ermöglicht es, den Prozessen die benötigte Rechenleistung zur Verfügung zu stellen, gleichzeitig jedoch auch den Verbrauch des Systems zu minimieren. [17] [18]

Dieses Scheduling-Verfahren findet jedoch keine Anwendung bei symmetrischen Prozessortopologien, da hier keine Einsparungen festgestellt werden konnten. Zudem bezieht sich dieser Ansatz nur auf den Energieverbrauch des Prozessors, auf den Einfluss der Prozesse untereinander wird kein Bezug genommen. [17]

Ziel dieser Arbeit ist es hier jedoch diesen Einfluss zwischen den Prozessen zu detektieren, um die Fairness bei Prozessen, die unverschuldet mit niedrigerer Frequenz ausgeführt werden, wiederherzustellen.

2.3.5 Zwischenfazit

Es wurde gezeigt, dass bereits mehrere Ansätze existieren, um den Energiebedarf bei der Ausführung zu minimieren oder um anhand des Energiebedarfs zu versuchen, die Rechenleistung fair unter den Prozessen aufzuteilen. All diese Ansätze berücksichtigen jedoch nicht ausgelöste Veränderungen der Frequenzen durch andere Prozesse oder ermöglichen es nicht bei hierdurch beeinflussten Prozessen wieder Fairness herzustellen. Diesen Problemen soll in dieser Arbeit begegnet werden.

2.4 AVX-512

AVX-Befehle benötigen bei deren Ausführung physikalisch bedingt sehr viel Energie, was zu einer erhöhten Wärmeentwicklung des Prozessors führt. Um diesen vor Schäden durch Überhitzung zu schützen, wird die anliegende Spannung, und hiermit die Frequenz des Prozessors, reduziert. Nach der Ausführung der AVX-Befehle kann die Spannung, und damit die Frequenz, wieder gesteigert werden. [1, Kapitel 2.2.3]

Hierbei ist zwischen AVX2- und AVX-512-Befehlen zu unterscheiden. Beide Arten führen zu einer Reduktion der Frequenz, letztere benötigen jedoch mehr Energie und setzen die Frequenz daher weiter herunter als AVX2-Befehle. Hierdurch entstehen verschiedene Frequenzniveaus auf denen der Prozessor ausgeführt werden kann: die reguläre Basisfrequenz, die zu AVX2-Befehlen zugehörige Basisfrequenz und die Basisfrequenz für AVX-512-Befehle. Diese werden als Power Level 0, 1 bzw. 2 bezeichnet und repräsentieren jeweils die Frequenzbereiche in denen die Taktfrequenz des Prozessors in Abhängigkeit der verwendeten Kerne liegen kann (vgl. Kapitel 4.1.1) [1, Kapitel 2.2.3]

Diese Arbeit bezieht sich ausschließlich auf die Effekte, die bei der Ausführung von AVX-512-Code auftreten. Zur einfachen Lesbarkeit wird im Folgenden von „AVX-Code“ bzw. „AVX-Befehlen“ gesprochen. Hierbei ist immer, soweit nicht anders angegeben, „AVX-512“ gemeint.

Um das Ende der Ausführung von AVX-Code zu erkennen, verwendet der Prozessor einen Timer von 670 µs [30][1]. Dieser wird bei der Ausführung eines

[1] Im Optimization Manual von Intel werden 2 ms genannt [1], Messungen haben jedoch 670 µs ergeben [30]

AVX-Befehls gestartet und bei weiterer Ausführung von AVX-Code zurückgesetzt [1]. Wenn der Timer abläuft, also kein AVX-Code mehr ausgeführt wurde, setzt der Prozessor die Frequenz wieder hoch. Aus diesem Verhalten folgt eine Benachteiligung für Prozesse die nach AVX-Code ausgeführt werden, da sie, bis zum Ablauf des Timers, mit geringerer Frequenz ausgeführt werden. Da der CFS die Fairness nur aus der Dauer auf dem Prozessor ableitet, werden diese Prozesse genauso behandelt wie welche, die bei regulärer Frequenz ausgeführt werden. [30]

Diese Problematik gibt es jedoch nicht nur bei der Ausführung von Prozessen nach AVX-Code. Bei der Verwendung von Hyperthreading führt die Ausführung von AVX-Code auf einem Hyperthread auch zu einer Verringerung der Frequenz auf dem anderen Hyperthread. So werden Prozesse mit geringerer Frequenz ausgeführt, obwohl zuvor kein AVX-Code auf diesem Hyperthread ausgeführt wurde. Die Problematik bleibt jedoch die gleiche: bis zum Ablauf des Timers werden auch diese Prozesse mit geringerer Taktfrequenz ausgeführt. [30] [29]

2.4.1 Dedizierte Kerne, manuelles Rescheduling

Um zuvor genanntem Fairness-Problem bei der Verwendung von AVX aus dem Weg zu gehen, wurde bereits folgender Ansatz entwickelt: Auf dem System werden die Prozessorkerne nicht mehr gleichberechtigt behandelt. Stattdessen werden sie in zwei Gruppen unterteilt: die, die AVX-Befehle ausführen dürfen und jene, die keine AVX-Befehle ausführen dürfen. Dieses Verfahren hat den Vorteil, dass Prozesse, die AVX-Code ausführen, sich nur gegenseitig verlangsamen. Prozesse die hingegen keine AVX-Befehle verwenden, können ohne Reduzierung der Ausführungsfrequenz auf den passenden Kernen ausgeführt werden. [28]

Konkret implementiert wird dieses Verfahren durch das Einführen eines neuen Systemaufrufs. Dieser soll von Prozessen, die AVX-Code ausführen wollen, aufgerufen werden. Das System führt diese Prozesse hierdurch auf einem Kern aus, der für deren Ausführung freigegeben ist. Hierdurch können die verbleibenden Prozesse ungebremst weiterhin auf dem aktuellen Kern ausgeführt werden. [28]

Das Problem dieses Ansatzes besteht vor allem in der Verwendung eines Systemaufrufs durch die betroffenen Prozesse. Das System muss darauf vertrauen, dass alle Prozesse bei der Ausführung von AVX-Befehlen zunächst diesen Systemaufruf ausführen. Falls Prozesse diesen Aufruf nicht ausführen, können diese auch AVX-Code auf dafür nicht freigegebenen Kernen ausführen und hierdurch wieder nachfolgende Prozesse verlangsamen. [28]

Zudem müssen sämtliche Anwendungen, die auf der Verwendung von AVX beruhen, so umgeschrieben werden, dass sie den entsprechenden Systemaufruf verwenden [28]. Dies führt zu zusätzlichem Entwicklungsaufwand und wird sich sicherlich nicht bei allen Anwendungen durchsetzen.

Weiterhin führt die Verwendung eines solchen Rescheduling-Vorgangs zusätzlichen Overhead ein, da der aktuelle Prozess unterbrochen, die Ausführung auf einen anderen Kern verschoben und im Anschluss ein neuer Prozess zur Ausführung auf dem Prozessor ausgewählt werden muss. [28]

2.4.2 Dedizierte Kerne, automatisches Rescheduling

Ein weiterer Ansatz hierbei ist es, ebenfalls dedizierte Kerne für die Ausführung von AVX-Code zu verwenden, jedoch das Zuteilen dieser Prozesse auf Prozessorkerne automatisch durchführen zu können. Hierbei verwendet man Traps: auf den Prozessoren, die keinen AVX-Code ausführen sollen, deaktiviert man diese Instruktionen. Beginnt nun ein Prozess auf einem solchen Kern mit Berechnungen durch AVX-Befehle, so wird die Trap ausgelöst. Hierbei wird in der dann aufgerufenen Routine der Prozess auf einen Kern, der für die Ausführung von AVX-Befehlen ausgewählt ist, verschoben. [27]

Der primäre Vorteil dieses Verfahrens gegenüber dem Vorgänger besteht darin, dass die Verantwortung des Schedulings nun nicht mehr bei den Anwendungen liegt. Es muss nicht mehr darauf vertraut werden, dass alle Anwendungen den oben beschrieben Systemaufruf unterstützen und auch die Zurverfügungstellung dieses Systemaufrufs entfällt. [27]

Dem entgegen stehen prinzipiell die sonstigen Nachteile wie beim vorherigen Ansatz: durch das Verschieben des Prozesses auf einen anderen Kern entsteht ein großer Overhead. Zudem muss die Anzahl der für AVX-Code freigegebenen Kerne clever gewählt werden, sodass keine Rechenleistung durch ein falsches Verhältnis zwischen AVX-Kernen und nicht-AVX-Kernen verschenkt wird. [27]

2.4.3 Zwischenfazit

Wie bereits beschrieben, ist die Verwendung von dedizierten Kernen nicht optimal geeignet, um den Einfluss von AVX-Befehlen auf nachfolgende Prozesse zu reduzieren bzw. zu vermeiden. Um dem Problem des Verschiebens der Prozesse und des dabei entstehenden Overheads zu beheben, wäre die gleichberechtigte Verwendung aller Kerne wünschenswert. Daher wird sich diese Arbeit damit befassen, einen solchen Lösungsansatz zu formulieren und zu evaluieren.

2.5 Performance Counter

Intel Prozessoren besitzen *Performance Counter*. Mit Hilfe dieser kann man verschiedene Metriken des Prozessors aufzeichnen. Hierzu stehen programmierbare und festgelegte Counter zur Verfügung. Letztere sind bereits fest konfiguriert und müssen nur noch aktiviert werden, um verwendet werden zu können. Bei den programmierbaren Countern kann man durch das Setzen spezieller Register konfigurieren, welche Ereignisse diese Counter zählen sollen. [1, Kapitel 2.2.3]

Mit Hilfe dieser Performance Counter lassen sich zum Beispiel die durchgeführten Zyklen des Prozessors aufzeichnen, sodass diese zur späteren Auswertung verwendet werden können. Weiterhin können auch die Zyklen pro Performance Level mitgezählt werden. [1, Kapitel 2.2.3]

2.5.1 Time-Stamp Counter

Ähnlich zu den Performance Counter besitzen Intel Prozessoren einen *Time-Stamp Counter (TSC)*. Dieser wird bei einem Reset des Prozessors auf 0 gesetzt und inkrementiert anschließend in Geschwindigkeit der Basisfrequenz des Prozessors. Hierbei ist zu beachten, dass der invariante TSC[2] konstant mit dieser Taktrate zählt, unabhängig der tatsächlichen Frequenz des Prozessors. Ist der variante TSC vorhanden, hängt dessen Taktrate mit der tatsächlichen Frequenz des Prozessors zusammen. [2, Vol. 3B, Kapitel 17.15]

Um einfache Zeitmessungen durchführen zu können, ist es notwendig, dass der invariante TSC unterstützt wird. Hierdurch kann, durch Differenzenbildung zweier Zeitpunkte und durch Skalierung dieser Differenz mit der Basisfrequenz des Prozessors, die verstrichene Zeit berechnet werden. Der für diese Arbeit zur Verfügung gestellte Prozessor (vgl. Abschnitt 4.1.1) unterstützt invarianten TSC[3]. Daher ist es möglich, diesen für die Zeitmessung zu verwenden (vgl. Abschnitt 3.1).

[2] Ob dieser aktiv ist, kann anhand der CPUID ermittelt werden [2, Vol. 3B, Kapitel 17.15]

[3] Validierbar durch die CPUID

Kapitel 3

Implementierung & Design

Das Ziel dieser Arbeit ist die Erkennung welche Prozesse durch die Ausführung von AVX-Code verlangsamt wurden. Hierzu wurden im vorherigen Kapitel die nötigen technischen Grundlagen beschrieben. In diesem Kapitel wird gezeigt, wie die Ausführung von AVX-Code und dessen Auswirkung auf nachfolgende Prozesse erkannt werden kann. Zudem wird erklärt, wie dieses Wissen genutzt werden kann, um bei beeinträchtigten Prozessen wieder Fairness herzustellen. Die konkrete Implementierung im Linux-Kernel wird im Anschluss noch kurz beschrieben.

3.1 Lösungsansatz

Das primäre Problem ist die Beeinträchtigung von Prozessen, die AVX-Code ausführen, auf andere Prozesse. Wie bereits gezeigt, existieren Ansätze dieses Problem zu beheben, indem AVX-Code ausschließlich auf dafür freigegebenen Kernen ausgeführt wird (vgl. Abschnitt 2.4). Diese Arbeit verfolgt einen Ansatz, der die Ausführung von AVX-Befehlen nicht auf spezifische Prozessorkerne beschränkt, sondern die Ausführung auf allen verfügbaren Kernen erlauben soll. Da das grundsätzliche Problem darin besteht, dass Prozesse nach AVX-Code durch die reduzierte Frequenz effektiv weniger Rechenzeit erhalten, soll dieser Ansatz diese reduzierte Zeit bei zukünftigen Schedulingvorgängen berücksichtigen. Hierzu macht man sich die Funktionsweise des CFS zu Nutzen.

3.1.1 Erkennung der Ausführung von AVX-Code

Zur Erkennung ob AVX-Befehle ausgeführt wurden, wird ein programmierbarer Performance Counter verwendet. Dieser wird so konfiguriert, dass er die Zyklen des Prozessors im Power Level 2 zählt. Nun wird bei jedem Aufruf des Schedulers

dieser Counter ausgelesen und mit dem vorherigen Wert verglichen. Falls sich der Wert nicht geändert hat heißt das, dass der Prozessor zwischen den Messungen nicht im Power Level 2 war und somit kein AVX-Code ausgeführt wurde. Ändert sich hingegen der Wert, so kann dies drei Auslöser haben: es wurde zwischen den Messungen AVX-Code ausgeführt, sodass der Prozessor die Frequenz reduziert hat. Alternativ wurde bereits in der vorherigen Phase AVX-Code ausgeführt und der Prozessor läuft aktuell noch mit geringerer Frequenz da der Timer noch nicht abgelaufen ist. Die dritte Möglichkeit besteht in einer Kombination aus den beiden vorherigen Fällen: der Prozessor läuft noch mit geringerer Frequenz bis der Timer abläuft, jedoch führt der aktuell laufende Prozess nun AVX-Code aus und bewirkt somit ein Zurücksetzen des Counters.

Die Problematik hierbei ist die Abgrenzung der ersten beiden Fälle zum letzten Fall. Nur so kann ermittelt werden, ob der soeben ausgeführte Prozess nur wegen eines noch nicht abgelaufenen Timers verlangsamt ausgeführt wurde oder ob er zusätzlich selbst AVX-Code ausgeführt hat. Dieser Entscheidungsprozess ist in Abbildung 3.1 dargestellt. In Abhängigkeit dessen, ob die Frequenz reduziert und AVX-Code ausgeführt wurde, kann ermittelt werden welche Prozesse durch zuvor ausgeführten AVX-Code beeinträchtigt wurden.

Diesem Problem kann man mit Traps begegnen. Hierbei kann auf dem Prozessorkern die Ausführung von AVX-Befehlen deaktiviert werden. Will nun ein Prozess auf eben diesem Kern Quellcode mit AVX-Befehlen ausführen, dann löst der Prozessor eine Trap aus. Dies lässt sich in `traps.c` [15] abfangen. Hierbei wird für den aktuell ausgeführten Prozess hinterlegt, dass AVX-Code ausgeführt wird. Im Anschluss wird die Ausführung von AVX-Befehlen wieder freigeschaltet und zur Ausführung zurückgekehrt.

Wie aus Abbildung 3.2 zu entnehmen ist, wird die Ausführung von AVX-Befehlen immer dann deaktiviert, wenn zuvor AVX-Befehle ausgeführt wurden. Hiermit lässt sich das Problem des zuvor beschriebenen dritten Szenarios unterbinden, da trotz des inkrementierenden Performance Counters nicht sichergestellt werden kann, ob ein Prozess AVX-Befehle ausführt oder dieser nur unter dem Einfluss des Timers leidet. Dieser Ablauf erlaubt folgende Rückschlüsse: ändert sich der Performance Counter und die Ausführung von AVX-Befehlen ist erlaubt, dann hat der soeben ausgeführte Prozess AVX-Code verwendet. Nun wird die Ausführung von AVX-Befehlen deaktiviert. Wenn sich beim nächsten Prozess weiterhin der Performance Counter ändert, kann durch das Auslösen der Trap erkannt werden, ob AVX-Code ausgeführt wurde oder lediglich der Timer noch nicht abgelaufen ist. Ändert sich der Performance Counter hingegen nicht mehr und ist die Ausführung von AVX-Befehlen untersagt, dann kann diese wieder freigegeben werden, da über die Änderung des Performance Counters die nächste Ausführung von AVX-Befehlen erkannt werden kann.

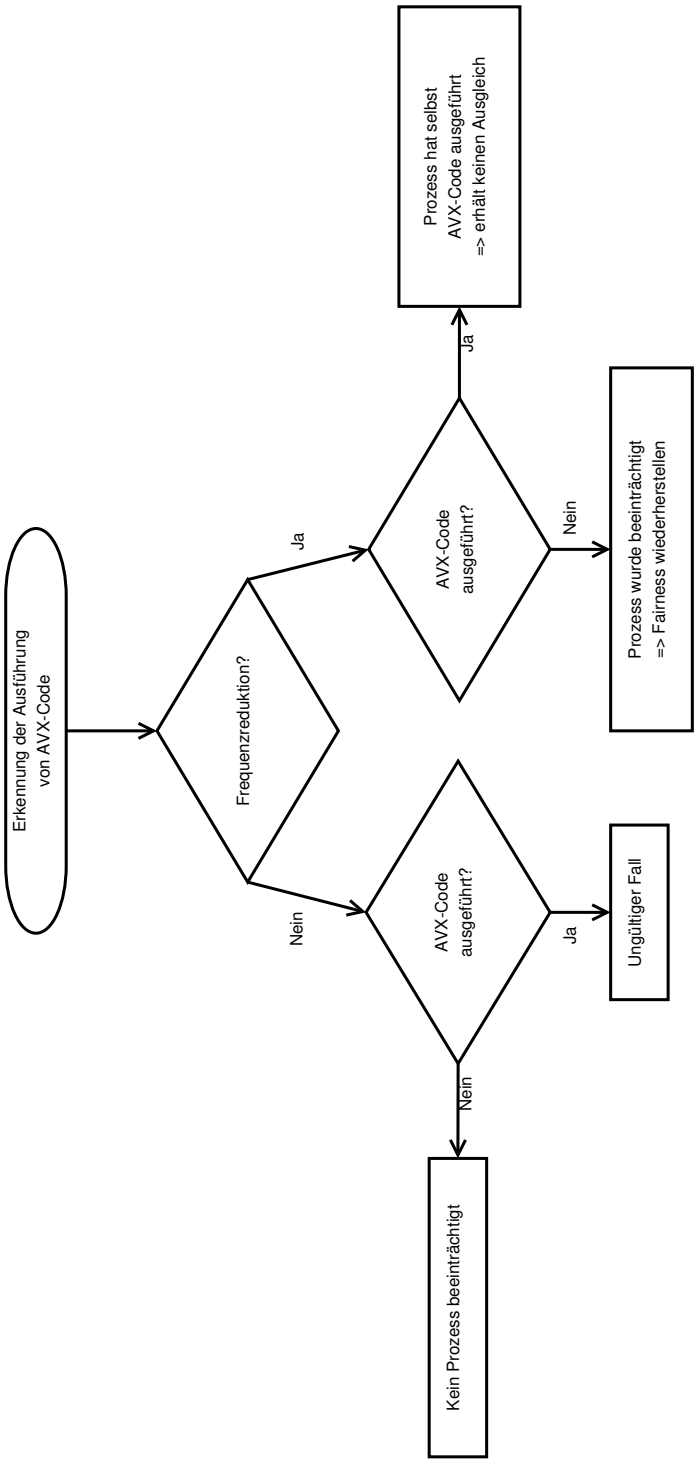

Abbildung 3.1: Aus der Kombination aus Frequenzreduzierung und ausgeführtem AVX-Code kann ermittelt werden, welcher Prozess durch zuvorigen AVX-Code beeinträchtigt wurde.

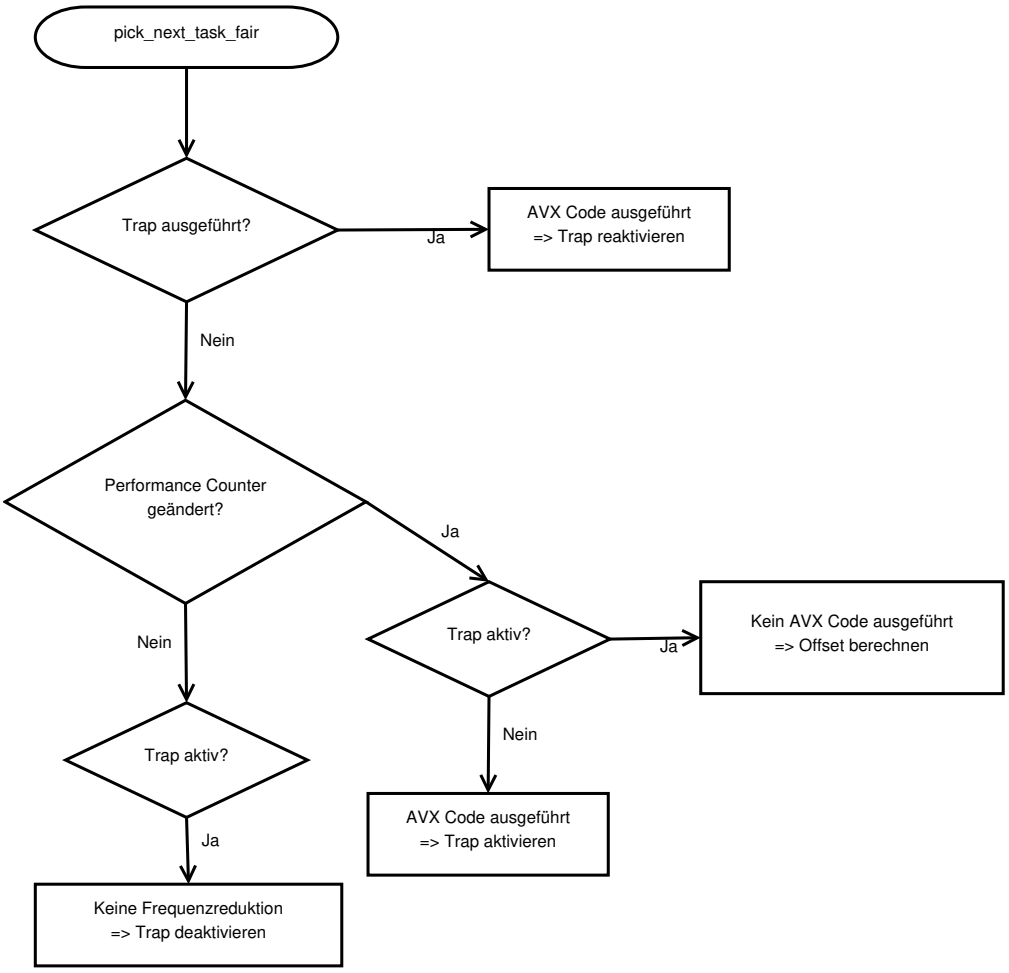

Abbildung 3.2: Durch Nutzung der Performance Counter und dem Auslösen von Traps kann in `pick_next_task_fair` erkannt werden, ob AVX-Code ausgeführt wurde oder ob einem langsamer ausgeführten Prozess Zeit gutgeschrieben werden muss.

3.1.2 Berechnung des Performance-Einflusses

Nachdem nun ein Konzept vorliegt, um die Ausführung von AVX-Code zuverlässig zu erkennen, kann dieses Wissen genutzt werden, um den Einfluss der Frequenzreduzierung zu berechnen. Hierzu muss zunächst die Frequenz bestimmt werden, mit der der betroffene Prozess ausgeführt wurde. Der ursprüngliche Ansatz war hierbei mit Hilfe der Performance Counter zu berechnen, mit welcher Frequenz der Prozessor während der Ausführung wie lange gelaufen ist. Der in Prozessoren verwendete Turbomodus [10] führt jedoch zu Schwankungen in der Taktfrequenz. Um dieses Problem zu vereinfachen, wurde der Turbomodus auf dem Testsystem deaktiviert.[1] Nun hätte mit Hilfe der Performance Counter und der damit verbundenen Basisfrequenzen die mittlere Ausführungsfrequenz bestimmt werden können. Hierzu wäre lediglich eine Verrechnung der zu den Performance Countern zugehörigen Basisfrequenzen notwendig gewesen. Messungen haben hier jedoch gezeigt, dass die Deaktivierung des Turbos sich, wie in Abbildung 3.3 dargestellt, ausschließlich auf den normalen Betriebsmodus des Prozessors bezieht und hier die entsprechende Basisfrequenz festsetzt. Bei der Ausführung von AVX2- und AVX-512-Befehlen wurden weiterhin die zugehörigen Turbofrequenzen genutzt. Durch diese schwankende Ausführungsfrequenz konnte die mittlere Frequenz nicht anhand der Performance Counter berechnet werden.

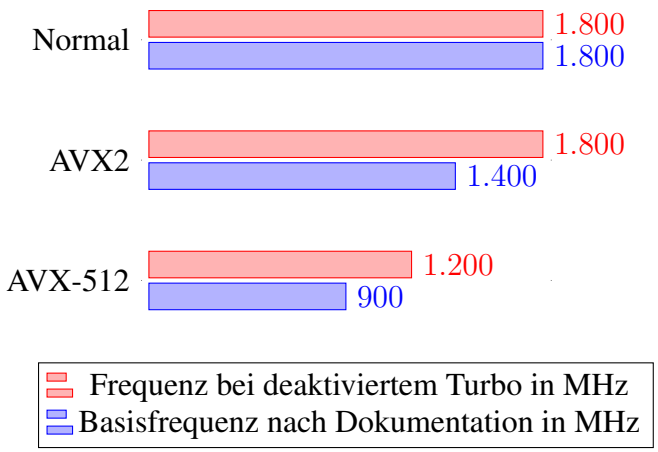

Abbildung 3.3: Vergleich der Basisfrequenz des Prozessors und der Frequenz bei deaktiviertem Turbo und Auslastung aller Kerne. Nur im normalen Modus entspricht die Frequenz bei deaktiviertem Turbo der Basisfrequenz.

[1] Diese Vereinfachung macht diesen Ansatz jedoch noch nicht praxistauglich (vgl. Kapitel 5.1)

Alternativ wurde nun mit einem Fixed Counter die Anzahl der ausgeführten Taktzyklen gezählt und mittels Auslesen von rdtsc[2] die verstrichene Zeit gemessen. Hierbei ist zu beachten, dass rdtsc immer Schritte mit der Basisfrequenz des Prozessors zählt, unabhängig von der aktuellen Ausführungsfrequenz. Um hieraus die verstrichene Zeit zu berechnen, muss dieser Wert noch mit der Basisfrequenz verrechnet werden. Im Anschluss kann durch Division die mittlere Frequenz während der Ausführungszeit des Prozesses berechnet werden:

$$freq_{average} = cycles * freq_{base}/cycles_{clock}$$

Hierbei sind $cycles$ die Anzahl an Zyklen, die durchgeführt wurden, $freq_{base}$ die Basisfrequenz des Prozessors in MHz (in diesem Fall 1.800 MHz, vgl. Abschnitt 4.1.1), $cycles_{clock}$ die Anzahl an Zyklen des TSC und $freq_{average}$ die resultierende Durchschnittsfrequenz in MHz.

Für Prozesse, die durch zuvorige Ausführung von AVX-Code beeinträchtigt wurden, kann nun mit Hilfe der mittleren Ausführungsfrequenz berechnet werden wie groß diese Beeinflussung war. Hierbei wird die ermittelte Frequenz mit der Basisfrequenz des Prozessors verglichen[3], um zu berechnen, um welchen Faktor der Prozess verlangsamt wurde. Wie im nächsten Abschnitt beschrieben, wird dieser Faktor verwendet, um die `vruntime` des Prozesses anzupassen.

3.2 Integration in den Linux-Kernel

Nachdem beschrieben wurde, wie die Beeinträchtigung von Prozessen durch die Frequenzreduzierung nach Ausführung von AVX-Befehlen festgestellt und Fairness wiederhergestellt werden soll, geht es in diesem Kapitel um die Umsetzung im Linux-Kernel. Hierbei spielen die Methoden `pick_next_task_fair` und `update_curr` eine zentrale Rolle [3] [14]. Die erste Methode wird aufgerufen, sobald beim Scheduling der als nächstes auszuführende Prozess ausgewählt werden soll [14]. An diesem Punkt setzt der entwickelte Code an, um die Performance Counter auszulesen und auszurechnen ob und wie viel Laufzeit der aktuell ausgeführte Prozess gutgeschrieben bekommen soll. Hierzu werden in der `cfs_rq` jeweils die aktuellen Werte der Counter abgespeichert, nachdem die Differenz zu den vorhergehenden Werten gebildet wurde.

Beim Aufruf von `update_curr` wird anhand der Priorität des Prozesses und dessen Ausführungszeit die `vruntime` berechnet [16]. Hierdurch kann der Prozess richtig in den Rot-Schwarz-Baum einsortiert werden, um bei zukünftigen

[2] rdtsc ist der zugehörige Befehl, um den in Abschnitt 2.5.1 beschriebenen TSC auszulesen

[3] Dies ist möglich, da durch das Deaktivieren des Turbos im normalen Betriebsmodus die Basisfrequenz des Prozessors gehalten wird.

Schedulingvorgängen fair ausgewählt zu werden [3]. Vor dieser Berechnung wird zunächst der zu eben berechnete Offset von der Ausführungszeit abgezogen. Hierdurch wird dem CFS suggeriert, dass der Prozess kürzer auf dem Prozessor ausgeführt wurde und daher zukünftig bevorzugt ausgeführt werden kann. Dies ermöglicht es, die Benachteiligung des Prozesses bei zukünftigen Ausführungen wieder auszugleichen.

Kapitel 4

Evaluierung

Anhand der zuvor vorgestellten Implementierung konnten nun Messungen durchgeführt werden, um die Funktionsweise des Schedulers zu evaluieren. Hierzu wird zunächst der Prozessor und die angewandte Methodik kurz erläutert. Im Anschluss folgen der konkrete Versuchsaufbau und die daraus resultierenden Messwerte.

4.1 System

Für die Durchführung dieser Arbeit stand ein System mit 48 GB Arbeitsspeicher, 256 GB SSD und einem Intel Xeon Silver 4108 (vgl. Abschnitt 4.1.1) zur Verfügung). Hierauf wurde Fedora 31 (Thirty One) installiert. Implementiert wurde dieser Prototyp im Linux Kernel 5.7.0 [13].

4.1.1 Prozessor

Wie eben beschrieben, stand für diese Arbeit ein Intel Xeon Silver 4108 der Skylake-Mikroarchitektur zur Verfügung. Dieser Prozessor besitzt 8 Kerne und läuft mit einer Basisfrequenz von 1.800 MHz. Bei Ausführung von AVX2 bzw. AVX-512 Befehlen reduziert sich diese Basisfrequenz auf 1.400 MHz bzw. 900 MHz. Die Turbofrequenzen in Abhängigkeit des verwendeten Modus und der Anzahl an Kernen sind der Tabelle 4.1 zu entnehmen. [5]

4.2 Methodik

Ziel dieser Arbeit ist ein faireres Scheduling für Prozesse zu realisieren, die nach AVX-Code ausgeführt werden. Da „Fairness“ jedoch keine messbare Metrik ist, muss eine andere genutzt werden. Hierbei führt man sich zunächst nochmals das

Modus	1 & 2	3 & 4	5 & 6	7 & 8
Normal	3.000 MHz	2.700 MHz	2.100 MHz	2.100 MHz
AVX2	2.900 MHz	2.300 MHz	1.800 MHz	1.800 MHz
AVX-512	1.800 MHz	1.500 MHz	1.200 MHz	1.200 MHz

Tabelle 4.1: Auflistung der Frequenzen in Abhängigkeit von Modus und Anzahl der Kerne des Intel Xeon Silver 4108 [5]

primäre Problem vor Augen: Der CFS versucht faires Scheduling zu gewährleisten, indem er die zur Verfügung stehende Rechenzeit möglichst gleich auf alle Prozesse verteilt. Hierbei berücksichtigt er jedoch nicht die reduzierte Taktrate, die durch die Ausführung von AVX-Befehlen erzwungen wird. Will man nun zeigen, dass ein Prozess, trotz AVX-Befehlen, ein faires Scheduling erhalten hat, so kann man die benötigte Ausführungszeit eines Referenzprozesses nutzen. Hierzu vergleicht man dessen Ausführungszeit, die bei Ausführung parallel zu einem AVX-Prozess ermittelt wird, mit der Ausführungszeit bei Ausführung neben einem Nicht-AVX-Prozess. Je fairer das Scheduling, desto näher muss hierbei die Ausführungszeit neben einem AVX-Prozess an der Ausführungszeit neben einem Nicht-AVX-Prozess liegen.

4.3 Versuchsaufbau

Da der Einfluss durch reduzierte Frequenz auf nachfolgende Prozesse nur 670 µs beträgt [30], muss häufig zwischen Prozessen mit und ohne AVX-Code hin- und hergewechselt werden. Dies wurde mit folgendem Aufbau erreicht: Auf einem Prozessorkern wurden vier Prozesse gestartet. Hiervon führten zwei AVX-Code aus und erzwangen durch Ping Pong IPC häufige Aufrufe des Schedulers. Die zwei verbleibenden Prozesse führten ebenfalls Ping Pong IPC durch, verwendeten jedoch keinen AVX-Code. Parallel wurden die verbleibenden Kerne komplett mit der Ausführung von AVX-Code ausgelastet, um die maximale Taktfrequenz auf 1.200 MHz herunter zu setzen.

Die Implementierung der Prozesse ohne AVX-Code wurde so gewählt, dass diese rund 970.000 Zyklen auf dem Prozessor ausgeführt werden, wovon in der Regel etwa 730.000 Zyklen noch im Power Level 2 durchgeführt werden bis der Timer die Frequenz wieder auf reguläres Niveau anhebt. Messungen haben gezeigt, dass diese Prozesse in Folge mit einer durchschnittlichen Frequenz von 1.200 MHz ausgeführt werden. Dies passt zu den Informationen aus dem Datenblatt (s. Tabelle 4.1), die besagen, dass bei Ausführung von AVX-512-Befehlen auf allen Kernen die maximale Frequenz 1.200 MHz beträgt, wohingegen die re-

guläre Frequenz 1.800 MHz betragen müsste [5]. Hieraus folgt, dass 75 % der Laufzeit um 33 % verlangsamt wurden. Hieraus lässt sich berechnen, dass unter optimalen Voraussetzungen der Prozess etwa 25 % früher die Ausführung beenden müsste. Dies würde ein faireres Scheduling darstellen.

Als Metrik wurde die Laufzeit einer der beiden Prozesse ohne AVX-Code gewählt. Welcher dieser Prozesse gewählt wird, ist hierbei egal, da sie via IPC synchronisiert sind und daher die gleiche Ausführungsdauer haben.

4.4 Messergebnisse

Bei Durchführung der Messungen wurden die in Tabelle 4.2 dargestellten Ergebnisse ermittelt. Im Vergleich zum regulären CFS wurde beim modifizierten CFS die Ausführungszeit des Nicht-AVX-Prozesses von 3 min 36 sec auf 3 min 13 sec verringert. Dies stellt eine Beschleunigung von etwa 11 % dar. Zudem wurde für eine Referenzmessung der modifizierte CFS dahingehend abgewandelt, dass dieser weiterhin alle Berechnungen durchführt, jedoch den berechneten Offset nicht auf `delta_exec` anwendet. Hierdurch lässt sich bestimmen, wie groß der Overhead des modifizierten CFS ist. Bei dieser Messung kam eine Ausführungszeit von 3 min 38 sec heraus. Dieses Ergebnis liegt sehr nah an der Laufzeit des regulären CFS (3 min 36 sec) und zeigt somit, dass der entstehende Overhead durch den modifizierten CFS nur sehr gering ausfällt.

Im Vergleich hierzu wurden in der virtuellen Maschine (VM), auf der die Entwicklung und damit die ersten Messungen durchgeführt wurden, Laufzeiten von 3 min 55 sec und 3 min 28 sec für den regulären bzw. den modifizierten CFS gemessen. Dies entspricht einer Beschleunigung um etwa 13 %. Die ingesamt langsamere Ausführung kann auf den Overhead durch die Verwendung der VM zurückgeführt werden. Die vergleichbaren Prozentwerte zeigen jedoch, dass auch in der VM der modifizierte Scheduler prinzipiell wie gewünscht funktioniert hat.

Genutzer CFS	**8 Kerne, VM**	**8 Kerne, Host**
Regulär	3 min 55 sec	3 min 36 sec
Modifiziert	3 min 28 sec	3 min 13 sec
Modifiziert (nur Overhead)	—	3 min 38 sec
Beschleunigung	13 %	11 %

Tabelle 4.2: Vergleich der Ausführungszeit von Nicht-AVX-Prozessen zwischen dem modifizierten und dem regulären CFS und die daraus resultierende Beschleunigung. Zudem ist der entstehende Overhead durch den modifizierten CFS aufgeführt.

4.5 Diskussion

Die Evaluation hat gezeigt, dass ein faireres Scheduling von Nicht-AVX-Prozessen unter dem Einfluss von AVX-Prozessen prinzipiell möglich ist. Weiterhin wurde deutlich, dass durch die Nutzung des modifizierten CFS nur ein sehr geringer Overhead zustande kommt. Dies ermöglicht es, bereits diesen Prototyp statt des regulären CFS zu nutzen ohne übermäßigen Performanceverlust zu erwarten.

Jedoch zeigten die Messungen auch, dass trotz des geringen Overhead, die Beschleunigung der Prozesse mit 11 % noch deutlich geringer als die theoretisch erwarteten 25 % ausfiel. Der Grund für dieses Verhalten ließ sich jedoch nicht abschließend bestimmen. Möglich wäre, dass die Verwendung der Durchschnittsfrequenz Ungenauigkeiten herbeiführt. Um dies zu evaluieren, müsste ein anderer Ansatz zur Bestimmung der Fairness gewählt werden. Dieser könnte dann mit dem bestehenden Prototypen verglichen werden.

Der Prototyp hat jedoch noch weitere Einschränkungen, die in den folgenden Abschnitten genauer erläutert werden. Wo möglich, wird auch ein Ansatz zur Lösung dieser Problematik genannt. Zuletzt werden noch weitere Beobachtungen, die im Laufe der Arbeit entstanden sind, erläutert.

4.5.1 Weitere Performanceeinflüsse

Für ein faires Scheduling von Prozessen müssen weitere Performanceeinflüsse als nur die benötigte Rechenzeit berücksichtigt werden [26]. Hierzu zählen zum Beispiel die Verwendung des Speichers und anderer Peripherie. Diesbezüglich existieren bereits Arbeiten, die für ein faires Scheduling solche Zugriffe berücksichtigen. Diese Ansätze berücksichtigen jedoch nicht die Beeinflussung auf Prozesse durch die Ausführung von AVX-Code. [33] [35]

Der in dieser Arbeit vorgestellte Prototyp bezieht sich jedoch ausschließlich auf die benötigte Rechenleistung der Prozesse. Um ein abschließendes Urteil zu fairem Scheduling in Bezug auf die gesamte Performance der Prozesse fällen zu können, ist weitere Entwicklungsarbeit nötig, die all diese Faktoren beim Scheduling berücksichtigt.

4.5.2 Hyperthreading

Die hier vorgestellte Lösung unterstützt noch nicht die Verwendung von Hyperthreading. Dies ist damit begründet, dass durch Ausführung von AVX-Code auf dem einen Hyperthread auch die Frequenz auf dem anderen Hyperthread herabgesetzt wird. Mit der aktuellen Implementierung würde die daraus resultierende Inkrementierung des Performance Counters jedoch dazu führen, dass der Scheduler einem Prozess fälschlicherweise die Ausführung von AVX-Code nachweist.

Eine Möglichkeit dies zu umgehen wäre die Reimplementierung des Schedulers ohne die Verwendung eines programmierbaren Performance Counters für Power Level 2, der nur durch die Verwendung von Traps die Ausführung von AVX-Code detektiert.

4.5.3 Turbomodus

Wie bereits in Abschnitt 3.1.2 beschrieben, ermöglicht dieser Prototyp keine Verwendung des Turbomodus. Dies macht den Prototypen noch nicht praxistauglich, da das Deaktivieren des Turbos für die meisten Anwendungsfälle keine akzeptable Lösung ist. Eine Möglichkeit dies zu beheben wäre, durch Nutzung der Performance Counter, die Berechnung der Frequenz, die bei Verwendung des Turbos möglich gewesen wäre.

4.5.4 Weitere Probleme

Durch die Nutzung der Performance Counter ergeben sich Probleme bei Verwendung der Anwendung *perf*. Perf nutzt, je nach verwendeten Parametern, selbst Performance Counter, um die gewünschten Metriken messen zu können [12]. Hierbei werden die Einstellungen der Performance Counter für perf überschrieben und auch nach Beendigung des Programms nicht wieder zurückgesetzt. Dies führt zu einem undefinierten Verhalten des Schedulers.

Auch die Nutzung von *htop* wird durch diesen Scheduler eingeschränkt. Htop nutzt die Ausführungszeit der Prozesse, um deren prozentualen Anteil an der Prozessorauslastung zu berechnen [20]. Da der Scheduler jedoch die gemessene Ausführungszeit der benachteiligten Prozesse reduziert, wird in htop eine falsche prozentuale Auslastung berechnet. Dies führt dazu, dass bei Aufsummierung der Prozente in htop eine geringere Systemauslastung suggeriert wird als in der Realität vorliegt. Bei der Verwendung von `nice` kommt es nicht zu diesem Fehlerfall, da `nice` nur die `vruntime` verändert, nicht die tatsächliche Ausführungszeit [22].

Die in diesem Kapitel aufgezeigten Probleme haben nur bedingten Einfluss auf die Praxistauglichkeit dieses Prototyps. Grundsätzlich ist es natürlich erstrebenswert die Nutzung von perf und htop zu ermöglichen, trotzdem kann, bei Beachtung der beschriebenen Einschränkungen, der Prototyp verwendet werden, um für ein faireres Scheduling bei der Nutzung von AVX-Prozessen zu sorgen. Um perf weiterhin vollständig nutzen zu können, wäre es notwendig, dass Einstellungen der Performance Counter nicht durch perf überschrieben werden, sondern zunächst überprüft wird welche Performance Counter frei wären. Alternativ könnten Performance Counter explizit für den Scheduler reserviert werden, sodass keine Beeinflussung anderer Anwendungen auftreten. Für die Verwendung von htop wäre es möglich den Prototyp so umzuschreiben, dass dieser nicht die tatsächliche

Ausführungsdauer manipuliert, sondern, ähnlich zu `nice`, nur an der `vruntime` Änderungen durchführt. Dies dürfte mit geringem Aufwand möglich sein.

4.5.5 Weitere Beobachtungen

Im Zuge dieser Arbeit konnten noch weitere Beobachtungen gemacht werden, die in den nächsten Abschnitten vorgestellt werden.

Zyklen zählen

Der verwendete Prozessor unterstützt sowohl die Messung der Ausführungszyklen pro Performance Level durch Performance Counter, als auch die Messung aller Taktzyklen durch einen Fixed Counter. Hierbei war eine Überlegung, dass die Summation der Performance Counter der drei Performance Level den Taktzyklen des Fixed Counter entsprechen müssten. Messungen zeigten, dass dies in der Tat zutrifft, sodass nicht die Verwendung aller 3 Performance Counter notwendig ist, um die Anzahl der Zyklen zu erhalten.

Zeitmessung

Die Messung von Zeit ist im Kernel sowohl mit *ktime* als auch rdtsc möglich. Hierzu muss jeweils vor und nach der Messung der Wert dieser Zähler abgespeichert werden, sodass im Anschluss die Differenz gebildet werden kann. Für die Nutzung von ktime sind in `linux/timekeeping.h` mehrere Funktionen angegeben, die es u.a. ermöglichen einen aktuellen Zeitstempel in ns zu erhalten [23]. Hierdurch erwies sich die Zeitmessung mit Hilfe von ktime als sehr einfach, da die berechnete Differenz beider Zeitstempel direkt die verstrichene Zeit in ns ergibt. Bei der Verwendung von rdtsc muss beachtet werden, dass dieser die Zyklen des Prozessors mit der Basisfrequenz zählt. Um hieraus die verstrichene Zeit zu berechnen, muss diese Differenz noch mit der Basisfrequenz des Prozessors verrechnet werden. Auch hier haben Messungen gezeigt, dass beide Verfahren zu ähnlichen Ergebnissen führen, sodass auch hier der verwendete Counter frei gewählt werden kann.

Durchschnittsfrequenz

Um die mittlere Frequenz des Prozessors zu ermitteln, wäre auch die Verwendung von `aperf` und `mperf` möglich gewesen. Hierbei handelt es sich um zwei Register, die mit einer Taktfrequenz inkrementiert werden. Hierbei inkrementiert `mperf` mit der maximalen Frequenz, während `aperf` jeweils mit der aktuellen Frequenz inkrementiert wird. Um hieraus die mittlere Frequenz während der Ausführung

zu erhalten, hätte wieder die Differenz der Register vor und nach der Messung berechnet werden müssen, um im Anschluss die mittlere Frequenz mit

$$freq_{average} = freq_{base} * \Delta aperf/\Delta mperf$$

auszurechnen. [25] [24]

Für die Entwicklung dieses Prototyps wurde viel zunächst in einer virtuellen Maschine (VM) getestet, bevor am Ende auf dem realen System gemessen wurde. Zu dieser Zeit war es nicht möglich die entsprechenden Register in der VM auszulesen. Es existiert zwar bereits ein erster Ansatz [19], um dies zu beheben, ohne weitere Vorarbeiten wäre dies jedoch nicht direkt integrierbar gewesen.

Kapitel 5

Fazit

Kernproblem war es, dass die Ausführung von AVX-Befehlen viel Strom benötigt und sich hierbei die CPU stark erwärmt. Daher wird bei der Ausführung solcher Befehle die Taktfrequenz des Prozessors reduziert. Diese Reduzierung führt jedoch zu einer Benachteiligung nachfolgender Prozesse. Hier gab es bereits Forschungsansätze, um das Problem zu beheben, indem Prozesse mit AVX-Code auf dediziert dafür vorgesehene Prozessoren verschoben wurden. Dies ermöglichte eine ungebremste Ausführung von Prozessen ohne AVX-Code, führte jedoch auch Overhead in das System ein.

Um diesem Problem zu begegnen, verfolgt diese Arbeit den Ansatz die angerechnete Ausführungszeit der benachteiligten Prozesse zu reduzieren, sodass der CFS diesen Prozessen zukünftig mehr Rechenzeit zuweist. Benachteiligt sind die Prozesse, die nach AVX-Code ausgeführt werden. Durch die Reduzierung der Frequenz bei AVX-Befehlen, können die danach ausgeführten Prozesse ebenfalls zunächst mit dieser geringeren Frequenz ausgeführt werden.

Hierzu wurde mit Hilfe von Performance Countern und dem Verwenden einer Trap die Ausführung von AVX-Code detektiert und mit Hilfe der benötigten Ausführungszeit die durchschnittliche Ausführungsfrequenz berechnet. Durch Vergleich mit der regulären Ausführungsfrequenz des Prozessors ließ sich hieraus ein Offset errechnen, der im Anschluss von der Ausführungszeit abgezogen wurde.

Prinzipiell wurde gezeigt, dass die Änderung der Ausführungszeit dazu genutzt werden kann, um das Schedulingverhalten des CFS so anzupassen, dass die Rechenzeit fairer auf die Prozesse verteilt wurde. Die hierbei erreichte Fairness war zwar noch nicht optimal, aber es wurden auch hier Punkte aufgezeigt an denen weitere Optimierungsarbeit ansetzen könnte. Weiterhin ermöglicht dieser Ansatz nicht die Verwendung des Turbomodus oder die Nutzung von Hyperthreading. Auch hier bedarf es weiterer Forschungsarbeit, um den Scheduler produktiv einsetzen zu können.

5.1 Weiterführende Arbeiten

Wie bereits in der Diskussion erwähnt, wäre eine Möglichkeit den Scheduler dahingehend abzuwandeln, dass er die Verwendung von Hyperthreading ermöglicht. Hier könnte man den Ansatz wählen, nur mit Hilfe von Traps die Ausführung von AVX-Code zu erkennen.

Des weiteren wäre eine Ausweitung des Schedulers auf die Verwendung von AVX2-Befehlen möglich. Hierzu könnte man mit Hilfe mehrerer programmierbarer Performance Counter ermitteln, in welchem Power Level sich der Prozessor wie lange befunden hat und dementsprechend einen Offset für betroffene Prozesse berechnen.

Die aktuelle Implementierung unterstützt auch nicht die Verwendung von Turbo im normalen Betriebsmodus des Prozessors, da hierdurch die Berechnung des Offsets durch die Durchschnittsfrequenz nicht mehr funktionieren würde. Daher ist dieser Prototyp auch noch nicht für den produktiven Einsatz geeignet, da die Deaktivierung des Turbos in der Praxis keine Option darstellt. Zur Behebung dieses Problems wäre ein Ansatz zur Ausführungszeit regelmäßig die aktuelle Frequenz des Prozessors abzufragen und anhand dessen zu berechnen, wann die Benachteiligung des ausgeführten Prozesses wie groß war. Hier ist jedoch eine offene Frage wie groß der Overhead einer solchen regelmäßigen Unterbrechung wäre und ob dies den Nutzen aufwiegen würde.

Literaturverzeichnis

[1] Intel® 64 and ia-32 architectures optimization reference manual. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html, 2018 (zugegriffen am 07.05.2020).

[2] Intel® 64 and ia-32 architectures software developer's manual combined. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html, 2018 (zugegriffen am 07.05.2020).

[3] Cfs scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html, (zugegriffen am 02.09.2020).

[4] Cfs: Completely fair process scheduling in linux. https://opensource.com/article/19/2/fair-scheduling-linux, (zugegriffen am 03.09.2020).

[5] Xeon silver 4108 - intel. https://en.wikichip.org/wiki/intel/xeon_silver/4108, (zugegriffen am 04.05.2020).

[6] nice(2) - linux manual page. https://man7.org/linux/man-pages/man2/nice.2.html, (zugegriffen am 04.09.2020).

[7] Interrupt 6 - invalid opcode exception. https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-212.html, (zugegriffen am 05.09.2020).

[8] How to change the length of time-slices used by the linux cpu scheduler? https://unix.stackexchange.com/questions/466722/how-to-change-the-length-of-time-slices-used-by-the-linux-cpu-scheduler, (zugegriffen am 07.09.2020).

[9] Dynamic voltage and frequency scaling (dvfs). https://semiengineering.com/knowledge_centers/low-power/techniques/dynamic-voltage-and-frequency-scaling/, (zugegriffen am 08.09.2020).

[10] Intel® turbo boost technology improves application performance. https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/intel-turbo-boost-technology.html, (zugegriffen am 08.09.2020).

[11] Interrupts and exceptions - understanding the linux kernel, second edition. https://www.oreilly.com/library/view/understanding-the-linux/0596002130/ch04s02.html, (zugegriffen am 08.09.2020).

[12] Perf wiki. https://perf.wiki.kernel.org/index.php/Main_Page, (zugegriffen am 08.09.2020).

[13] Quellcode des linux kernels in version 5.7.0. https://elixir.bootlin.com/linux/v5.7/source, (zugegriffen am 08.09.2020).

[14] Quellcode zu pick_next_task_fair. https://elixir.bootlin.com/linux/v5.7/source/kernel/sched/fair.c#L6957, (zugegriffen am 08.09.2020).

[15] Quellcode zu traps. https://elixir.bootlin.com/linux/v5.7/source/arch/x86/kernel/traps.c#L277, (zugegriffen am 08.09.2020).

[16] Quellcode zu update_curr. https://elixir.bootlin.com/linux/v5.7/source/kernel/sched/fair.c#L845, (zugegriffen am 08.09.2020).

[17] Energy aware scheduling. https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html, (zugegriffen am 09.09.2020).

[18] Energy aware scheduling (eas) progress update. https://www.linaro.org/blog/energy-aware-scheduling-eas-progress-update/, (zugegriffen am 09.09.2020).

[19] Patch support aperf/mperf registers. https://lkml.org/lkml/2020/6/8/182, (zugegriffen am 09.09.2020).

[20] Quellcode von htop. `https://github.com/htop-dev/htop/blob/cd55cfd6d263a89bbaf401481ea6bd6a5dd8f110/linux/LinuxProcessList.c#L837`, (zugegriffen am 09.09.2020).

[21] Many uses for core scheduling. `https://lwn.net/Articles/799454/`, (zugegriffen am 10.09.2020).

[22] Quellcode zu `update_curr` bei berücksichtigung von nice. `https://elixir.bootlin.com/linux/v5.7/source/kernel/sched/fair.c#L866`, (zugegriffen am 10.09.2020).

[23] Quellcode zur verwendung von ktime. `https://elixir.bootlin.com/linux/v5.7/source/include/linux/timekeeping.h`, (zugegriffen am 10.09.2020).

[24] Ubuntu manpage: cpufreq-aperf. `https://manpages.ubuntu.com/manpages/bionic/man1/cpufreq-aperf.1.html`, (zugegriffen am 10.09.2020).

[25] x86: Calculate mhz using aperf/mperf for cpuinfo and scaling_cur_freq. `https://patchwork.kernel.org/patch/8765251/`, (zugegriffen am 10.09.2020).

[26] Saeid Barati and Hank Hoffmann. Providing fairness in heterogeneous multicores with a predictive, adaptive scheduler. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

[27] Peter Brantsch. Core specialization for avx-512 using fault-and-migrate, 8. Juli 2019.

[28] Mathias Gottschlag and Frank Bellosa. Mechanism to mitigate avx-induced frequency reduction. Technical report, KIT, 2018.

[29] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. Automatic core specialization for avx-512 applications. In *Proceedings of the 13th ACM International Systems and Storage Conference*, 2020.

[30] Mathias Gottschlag, Yussuf Khalil, and Frank Bellosa. Dim silicon and the case for improved dvfs policies. Technical report, KIT, 2020.

[31] Gangyong Jia, Xuhong Gao, Xi Li, Chao Wang, and Xuehai Zhou. Dts: Using dynamic time-slice scaling to address the os problem incurred by dvfs. Technical report, University of Science and Technology of China, 2012.

[32] Yanpei Liu, Guilherme Cox, Qingyuan Deng, Stark C. Draper, and Ricardo Bianchini. Fastcap: An efficient and fair algorithm for power capping in many-core systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software*, 2016.

[33] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007.

[34] Sangyoung Park, Jaehyun Park, Donghwa Shin, Yanzhi Wang, Qing Xie, Massoud Pedram, and Naehyuck Chang. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume: 32 , Issue: 5 , May 2013)*, 2013.

[35] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[36] J. Wei, R. Ren, E. Juarez, and F. Pescador. A linux implementation of the energy-based fair queuing scheduling algorithm for battery-limited mobile systems. In *IEEE Transactions on Consumer Electronics (Volume: 60, Issue: 2, May 2014)*, 2014.