Performanz

Max Binder

Hauptseminar - Multicore-Programmierung - WS 2010
Lehrstuhl für Programmierung

10. Februar 2011

Inhaltsverzeichnis

1 Einleitung

1.1 Vorwort

Die beeindruckende Entwicklung der Leistungsfähigkeit von Prozessoren in der Vergangenheit befindet sich seit Kurzem in einem Stadium, wo alte Ansätze zum Erreichen höherer Taktfrequenzen nur noch sehr begrenzt umsetzbar sind. Hauptgrund dafür ist die enorme Hitzeentwicklung auf den Prozessorchips.
Natürlich ist aber auch weiterhin eine Leistungssteigerung bei Computern erwünscht. Deshalb wird derzeit die Strategie verfolgt, mehrere Prozessorkerne auf einem Chip unterzubringen und die anfallenden Berechnungen auf die einzelnen Kerne aufzuteilen.
Dieser Umstand eröffnete neue Bereiche in der Forschung und hat wissenschaftliche Gebiete in den Fokus gerückt, die früher weniger intensiv verfolgt wurden. Ein Aspekt, der aus dieser Entwicklung hervorging und der auch generell in jüngster Vergangenheit zunehmend an Wichtigkeit gewinnt, ist der Stromverbrauch von Rechnern; unmittelbar davon ist auch die Wärmeentwicklung abhängig. Allerdings ist dies nicht nur der Hauptgrund für die Stagnation beim Design schnellerer Prozessor-Hardware, sondern gewinnt auch wegen der klimatischen Entwicklung auf unserem Planeten in jüngster Vergangenheit zunehmend an Aufmerksamkeit. Man versucht also jetzt, die Software effizienter zu gestalten um die verfügbare Hardware besser auszunutzen.
Aus diesem Grund wird im Folgenden das Thema “Performanz” näher betrachtet und mit Beispielen veranschaulicht.

1.2 Definition

Unter dem Oberbegriff Performanz fasst man vier Teilbereiche zusammen, welche die Leistungsfähigkeit eines Datenverarbeitungssystems charakterisieren. Stets dreht es sich dabei um den Anspruch auf Ressourcen des Systems.

- Zeit
- Energie
- Prozessor
- Speicher

1.3 Performanz im Laufe der Zeit

In der Vergangenheit gingen Überlegungen zur Optimierung der Performanz stets in die Richtung, dass man der Hardware - genauer gesagt dem Prozessor - genau soviel Energie zuführt, dass die anstehenden Berechnungen in einem gegebenen Zeitintervall noch durchgeführt werden können.
Weil die Hitzeentwicklung direkt von der Energieaufnahme abhängig ist, wurde somit als positiver Nebeneffekt auch die Energie, die zur Kühlung aufgewendet werden musste, auf ein Minimum reduziert.
Zuerst versuchte man die Energieaufnahme der Hardware ideal zu regeln. Die Software allerdings fand bei diesen Entwicklungen sehr wenig Beachtung.
Damals ergab sich die Geschwindigkeit eines Systems hauptsächlich aus der Energie, die man dem System zuführte. In den Anfängen des Power-Management waren die Computer-Systeme noch wesentlich kleiner und weniger komplex als heute und man praktizierte lediglich die Drosselung einzelner Komponenten des Rechners. So gab es beispielsweise in fast allen Rechnern nur einen Prozessorkern dessen Taktrate man regulieren konnte, und nur eine Festplatte, deren Drehgeschwindigkeit man je nach Gebrauch bremsen bzw. beschleunigen konnte.
Diese Regulierung der Hardware im binären Schema war zwar simpel aber doch sehr effektiv.
Bei modernen Rechnern allerdings hat sich bei der Hardware vieles verändert. Sogar in kleine Geräte sind Multicore-Prozessoren verbaut und die Aufnahme der Energie muss auf ein Minimum reduziert werden, damit der Betrieb auch mit Akkus längere Zeit möglich ist.[9]
Aus diesem Grund wird jetzt zunehmend auch Software genauer inspiziert und gewissenhafter konzipiert, denn auch wenn Software nicht unmittelbar selbst Energie verbraucht, so legt sie doch fest, welche Systemteile des Rechners zu welchem Maße beansprucht werden.
Diese Entwicklungen haben sich mit dem Durchbruch der Multicore-Rechner im Jahr 2005 und dem Erscheinen der ersten Smartphones, die seit dem Jahr 2007 stetig wachsende Popularität erfahren, noch verstärkt.

2 Analyse und Visualisierung von zeitlicher Performanz

2.1 Begriffe

- Response Time
  "Response Time" — zu deutsch Antwortverzögerung — ist die Zeit, die ein System benötigt, um vorliegende Tasks auszuführen. Die Maßeinheit dafür ist also Zeiteinheit pro Task.

- Throughput
  "Throughput" — zu deutsch Datendurchsatz — gibt die Datenmenge an, die pro Zeiteinheit verarbeitet werden kann.

2.2 Gegenüberstellung von Response Time und Throughput

Auf den ersten Blick möchte man denken, dass diese beiden Maße eine reziproke Verwandtschaft verbindet. Dies ist aber im Allgemeinen nicht der Fall, wie im folgenden Beispiel klar wird:

Man nehme an, es wurde ein Datendurchsatz von 1000 Tasks pro Sekunde gemessen. Daraus ergibt sich nicht zwangsläufig, dass die durchschnittliche Response-Time 1ms pro Task ist, denn es wäre denkbar, dass das System, welches diesen Datendurchsatz erzeugt hat, 1000 parallele, unabhängige Service-Kanäle hat, die die Anfrage erwartet und bearbeitet haben. Somit kann die Ausführung jedes einzelnen Tasks genau 1 Sekunde gedauert haben. Folglich weiß man also nur, dass die durchschnittliche Response-Time zwischen 0 und 1 Sekunde pro Task liegt.

Ebenso ist es nicht möglich, aus einer gegebenen durchschnittlichen Response-Time den Throughput genau zu bestimmen. Gründe dafür sind der schwer einschätzbare CPU-Scheduler und die serialisierten Ressourcen im System, wie z.B. exklusiver Schreibzugriff auf Speicherelemente.
Diese Faktoren führen dazu, dass man Response-Time und Throughput separat messen muss, wenn man sie kennen möchte. Welches dieser beiden Maße die höhere Priorität besitzt, ist abhängig von der aktuellen Situation. Für den Betreiber eines Systems ist es aber im Allgemeinen wichtig, dass beide Parameter gewisse Benchmarks einhalten.

2.3 Prozentuale Abwägungen

Um zeitliche Abschätzungen zu formulieren, kann man unterschiedliche Darstellungen benutzen. Beispielsweise kann man die Formulierung: *"In 99% oder mehr Fällen liegt die Response-Time bei 1 Sekunde oder weniger."* wählen. Genauso ist es auch möglich, zu sagen: *"Die durchschnittliche Response-Time liegt bei 1 Sekunde oder weniger."*.
Der grundsätzliche Unterschied dieser beiden Aussagen ist, dass bei der ersten Sichtweise zusätzlich zu der Obergrenze der Response-Time eine geringe Varianz der Messwerte vorausgesetzt wird, sodass sie eine präzisere Einschätzung

der Performanz bietet.
Ein weiterer Grund, warum die erste Formulierung zu bevorzugen ist, ist die Tatsache, dass sie am besten zu der menschlichen Wahrnehmung passt, wie folgendes Beispiel veranschaulicht:

Abbildung 1: Gütevergleich von Antwortverzögerungen

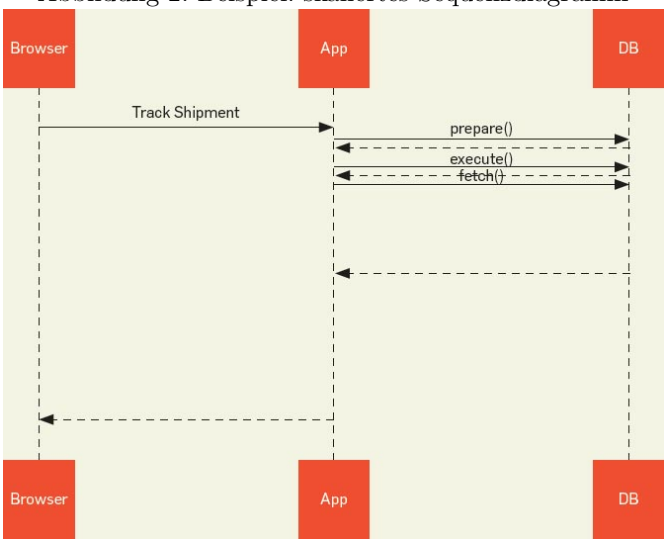
	List A	List B
1	.924	.796
2	.928	.798
3	.954	.802
4	.957	.823
5	.961	.919
6	.965	.977
7	.972	1.076
8	.979	1.216
9	.987	1.273
10	1.373	1.320

Man stelle sich vor, ein Task habe eine Response-Time Toleranz von bis zu 1 Sekunde. Die Werte für die Dauer von zehn separaten Abläufen dieses Tasks sind jeweils in Liste A und Liste B in der linken Tabelle gegeben. Durchschnittlich dauert der Task also in beiden Fällen genau 1 Sekunde, aber trotzdem haben sie ein Merkmal, das sie grundlegend voneinander unterscheidet: die Varianz.
In Liste A war die Response-Time in 90% der Fälle unter 1 Sekunde, während in Liste B dies nur in 60% der Fälle zutrifft.
Anders formuliert bedeutet dies, dass in Liste B 40% der Taskausführungen von der Geschwindigkeit her nicht den Ansprüchen des Benutzers genügten, während dies im Beispiel von Liste A nur in 10% der Fälle auftrat.

2.4 Sequenzdiagramme

Sequenzdiagramme sind in UML formulierte Interaktionsabläufe, die eine zeitliche Abfolge berücksichtigen. Man kann solchen Graphen besondere Aussagekraft verleihen, indem man die Abstände zwischen den request-Kanten und den zugehörigen response-Kanten so skaliert, dass sie die verbrauchte Zeit widerspiegeln. Somit ist auf den ersten Blick absehbar, wie sich die Response-Time im Laufe der Abarbeitung des Tasks entwickelt hat.
Außerdem ist diese Art der Darstellung sehr gut geeignet um parallel ablaufende Threads zu visualisieren.[4]
Im nachfolgendem Beispiel wird die Interaktion zwischen einem Browser, eines Application-Servers und einer Datenbank dargestellt.

Abbildung 2: Beispiel: skaliertes Sequenzdiagramm



2.5 Taskprofile

Für Tasks mit einer großen Anzahl an Aufrufen ist die Darstellung mit Sequqenzdiagrammen jedoch ungeeignet, weil die Übersicht verloren geht, sobald die Zahl der request- bzw response-Kanten zu groß wird. Folglich benötigt man für solche Tasks eine praktischere Art der Darstellung, bei der man mehrere Aufrufe zusammengefasst wiedergeben kann.
Ein Mittel um selbst solche Daten sinnvoll darzustellen, ist das Taskprofil. Dies ist eine tabellarische Aufschlüsselung der Response-Time eines einzelnen Tasks, die (normalerweise) absteigend geordnet ist.
Das Taskprofil zeigt, wo der Quellcode wieviel Zeit verbraucht, sodass man auch Rückschlüsse ziehen kann, wo die Zeit nicht verbraucht wurde, was manchmal eine noch wertvollere Information darstellt.
Bedenkt man nun, dass die Steigerung der Performanz proportional ist zu der Inanspruchnahme eines Systemteils, lässt sich also sehr gut beurteilen, welche Änderungen die höchste Beschleunigung des Systems bewirken können.

Abbildung 3: Beispiel: Taskprofil

	Function Call	R (sec)	Calls
1	`DB: fetch()`	1,748.229	322,968
2	`App: await _ db _ netIO()`	338.470	322,968
3	`DB: execute()`	152.654	39,142
4	`DB: prepare()`	97.855	39,142
5	Other	58.147	89,422
6	`App: render _ graph()`	48.274	7
7	`App: tabularize()`	23.481	4
8	`App: read()`	0.890	2
	Total	**2,468.000**	

Natürlich ist es nicht immer sinnvoll zu versuchen, das Taskprofil von oben her abzuarbeiten ohne sich über den Aufwand für eine Verbesserung Gedanken gemacht zu haben. Nach einer gewissenhaften Abschätzung der Kosten und der Nutzen kann man das Taskprofil jedoch um ein paar Angaben, wie z.B. die erwartete Performanz-Steigerung und die anfallenden Kosten erweitern und hat schließlich aussagekräftige Hinweise erarbeitet, wie eine Agenda zur Leistungssteigerung aussehen kann.[8]

Abbildung 4: Beispiel: erweitertes Taskprofil

	Potential improvement % and cost of investment	R (sec)	R (%)
1	34.5% super expensive	1,748.229	70.8%
2	12.3% dirt cheap	338.470	13.7%
3	Impossible to improve	152.654	6.2%
4	4.0% dirt cheap	97.855	4.0%
5	0.1% super expensive	58.147	2.4%
6	1.6% dirt cheap	48.274	2.0%
7	Impossible to improve	23.481	1.0%
8	0.0% dirt cheap	0.890	0.0%
	Total	**2,468.000**	

3 Performanz und Energie

Auch der exakt dimensionierte Verbrauch von Energie zur Abarbeitung eines Tasks in einer gegebenen Zeitspanne gehört zur optimalen Performanz eines Systems. Hier gibt es meherere Möglichkeiten, den Energieverbrauch angemessen zu drosseln. Natürlich steht eine verminderte Energiezufuhr in das System direkt im Widerspruch zum Erreichen der schnellstmöglichen Geschwindigkeit, allerdings ist die minimale Dauer einer Taskausführung auch nicht in jedem Fall zwingend erforderlich, sodass sich ein Zeitintervall ergibt, welches es optimal auszufüllen gilt.
Im Folgenden werden Techniken vorgestellt, mit denen dies bewerkstelligt werden kann.

3.1 Power-Down-Mechanismen

Hierbei handelt es sich um das meist verbreitete Werkzeug zur Verminderung des Energieverbrauchs, mit der nahezu jeder täglich Umgang hat. Die wohl bekanntesten Beispiele sind das Ausschalten des Monitors, wenn der Rechner einige Zeit nicht benutzt wurde, der Wechsel von Laptops in den Standby-Modus oder die Verringerung der Display-Helligkeit, sobald ein Netbook nicht mehr an eine Stromquelle angeschlossen ist.
Für gewöhnlich greifen solche Power-Down-Transitionen nachdem ein gewisser Schwellwert überschritten wurde, demnach ist die Qualität solcher Mechanismen unmittelbar abhängig von den gewählten Schwellwerten.
Primäres Ziel ist es also, Strategien zu finden, die diese Schwellwerte errechnen und dabei stets eine beweisbar gute Performanz — gemessen an der optimal erreichbaren Performanz — liefern.

3.2 Power-Management

Für gewöhnlich befindet sich ein Gerät in einem von mehreren Energizuständen, wie beispielsweise “Aktiv”, “Standby”, “Sleep”, “Aus”. Jeder dieser Zustände hat einen gewissen Energiebedarf, wobei der Energieaufwand zum Wechseln von einem höherwertigen Energiezustand in einen niedrigwertigeren quasi vernachlässigbar gering ist. Andersherum ist der Energieverbrauch jedoch durchaus signifikant, sodass die Abfolge von “aktiv”- und “idle”-Phasen möglichst sinnvoll gestaltet werden sollte. Man benötigt also einen Algorithmus, der entscheiden kann, wann zu welchen Zuständen gewechselt werden muss um die Energieaufnahme des Systems zu minimieren.
Da das System zu keiner Zeit die zukünftigen Ereignisse vorhersehen kann handelt es sich hierbei um ein sogenanntes “online-Problem”, welches von Irani et al. erforscht wurde.[5]
Zur Bestimmung der Güte eines Algorithmus, der versucht dieses Problem zu lösen verwendet man *“competitive analysis”*. Hierbei wird der Online- Algorithmus verglichen mit einem optimalen Algorithmus, der sämtliche zukünftigen Ereignisse im System kennt und somit die bestmögliche Abfolge von Energiezuständen berechnen kann.
Sollte der Online-Algorithmus für jede mögliche Eingabe maximal c-mal soviel Energie verbrauchen wie der optimale Algorithmus, so wird er *“c-competitive”*

genannt.
Eine Solche Analyse ist äußerst aussagekräftig, denn sie liefert eine starke worst-case Abschätzung des Energieverbrauchs unter allen vorstellbaren Eingaben, sogar wenn absichtlich versucht wird das System zu stören.

3.3 Systeme mit mehreren Energiezuständen

Die meisten modernen Geräte mit mehreren Energizuständen sind nach dem offenen Industrie-Standard des "Advanced Configuration and Power Interface", kurz ACPI, spezifiziert.[1]
Dieser Standard erschien erstmals im Dezember 1996 und wurde ursprünglich entwickelt von Intel, Microsoft, und Toshiba - später auch HP und Phoenix. Er zielt darauf ab, bestehende Standards für Leistungsaufnahme und Konfigurationen zu vereinheitlichen und zu verbessern, sodass Power-Management erst plattformübergreifend möglich wurde.
Ein Beispiel dafür sind die dort definierten Prozessorzustände:

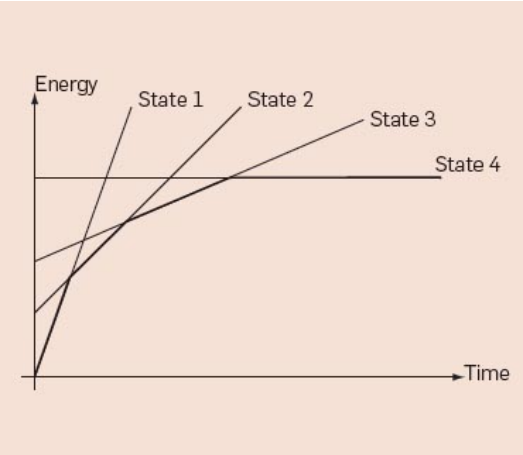
- C0: rechnender Zustand
- C1: Der Prozessor führt aktuell keine Instruktionen aus, kann aber nahezu ohne Zeitaufwand in den Zustand C0 wechseln.
- C2: Die Verwaltung der Software wird weiterhin aufrechterhalten, wobei der Wechsel in einen höheren Zustand eine geringe Wake-Up Zeit benötigt. (optional)
- C3: In diesem Zustand muss der Prozessor seinen Cache nicht kohärent halten und befindet sich in einem absoluten Ruhezustand. Er kann nach einer Wake-Up Zeit in einen höheren Zustand wechseln. (optional)

Man stelle sich nun ein System vor, das l unterschiedliche Leistungslevel $s_1, ..., s_l$ hat. r_i sei die jeweils zugehörige Leistungsaufnahme von s_i, wobei gelten soll $r_1 > ... > r_l$. Folglich ist s_1 der "Aktiv"-Zustand, der in der ACPI dem Zustand C0 entspricht.
Weiterhin sei β_i, die Energie, die benötigt wird, um eine Transition von s_i nach s_1 durchzuführen. Die Kosten für so einen Zustandswechsel liegen für β_1 bei 0 und steigen mit dem Index an. Außerdem sind die Kosten für den Wechsel zu einer niedrigeren Leistungsaufnahme vernachlässigbar gering – liegen also bei 0.
Das Ziel ist es nun die Zustandsübergänge so zu organisieren, dass der gesamte Energieverbrauch eines Zeitintervalls minimal wird. Unter der Annahme, dass die Transitions-Energien additiv sind, betragen also die Kosten für den Wechsel von s_j nach s_i, mit $i < j$, $\beta_j - \beta_i$.
Um eine optimale offline-Strategie "OPT" für eine Idle-Phase der Länge T zu finden, geht man folgendermaßen vor:
Dabei finden maximal zwei Transitionen statt: Eine zu Beginn und eine am Ende der Phase. Die Leistungsaufnahme von OPT ist also $r_i * T + \beta_i$, wobei OPT das Energieoptimum für eine Phase nach folgender Formel errechnet:

$$OPT(T) = min\{r_i * T + \beta_i\}, \text{ für } 1 \leq i \leq l$$

Während der Idle-Phase selbst macht es keinen Sinn, von dem errechneten Zustand s_i in einen niedrigewertigeren s_j zu wechseln, weil am Ende wieder die Kosten β_j anfallen um das System anzufeuern.
Ebenso ist eine Transition von s_i in ein höherwertiges Leistungslevel s_h ineffizient, weil $r_i \leq r_h$ ist und zum Schluss der Idle-Phase die Energie $\beta_i - \beta_h$, die man gewinnen würde, sowieso aufgewendet werden muss um nach s_1 zu wechseln.
Folgende Grafik veranschaulicht die optimalen Kosten bei fortlaufender Zeit T. Die bestmögliche Sequenz von Zustandstransitionen erhält man, indem man stets der Gerade folgt, die der X-Achse am nächsten ist. Dieses Vorgehen wurde erforscht von Irani et al. und wird "Lower-Envelope" genannt.

Abbildung 5: Optimale Energiekosten in einem System mit vier Leistungsstufen



Diese Ergebnisse kann man nutzen, um einem Online-Algorithmus Stützstellen t_i zu geben, an denen er seine Zustandstransitionen orientieren kann. Sämtliche Zeitpunkte t_i errechnen sich dabei aus den Schnittpunkten der Geradengleichungen $f_{i-1}(t)$ und $f_i(t)$, also durch Lösen der Formel $r_{i-1} * t + \beta_{i-1} = r_i * t + \beta_i$, wobei Zustände, deren Funktionen nicht auftreten können, nicht betrachtet werden.

3.4 Dynamic Speed Scaling

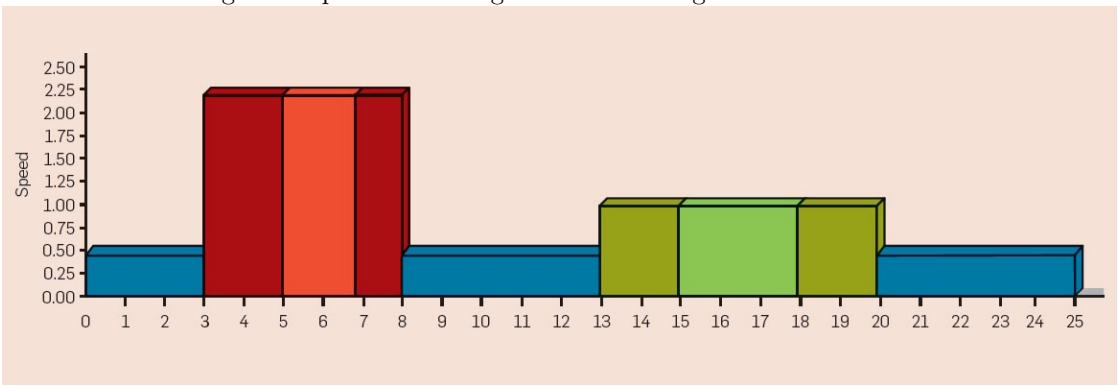
Nachdem nun vorgestellt wurde, wie man ein System mit mehreren Leistungsleveln modellieren kann, und welche Strategien man zur Berechnung der optimalen Prozessorzustände heranziehen kann, findet das Ganze Anwendung im "Dynamic Speed Scaling". Beispiele wo diese Technik angewendet wird ist SpeedStep von Intel oder auch PowerNow von AMD.
Hier ist das erklärte Ziel, die Geschwindigkeit eines Prozessors stets dynamisch zu regeln um einen minimalen Energieverbrauch zu erreichen, während die Qualität eines Dienstes nicht zu stark eingeschränkt werden soll.
Ein Scheduler muss also zu jeder Zeit entscheiden können, welcher Task mit welcher Geschwindigkeit ausgeführt werden soll. Zunächst werden Scheduling-Probleme mit festen Deadlines betrachtet:
Sei n die Anzahl der Jobs $J_1, ..., J_n$, die auf einem Prozessor mit unterschiedlichen Geschwindigkeitsstufen berechnet werden sollen. Jeder dieser Jobs J_i hat eine zugehörige release-time r_i, eine deadline d_i und eine Berechnungsumfang von w_i. Das Zeitintervall in dem der Job erledigt werden muss ist also markiert durch r_i und d_i. Falls J_i mit konstanter Geschwindigkeit s abgearbeitet wird, dauert die Berechnung $\frac{w_i}{s}$ Zeiteinheiten. Preemption, also das Unterbrechen und spätere Weiterausführen von Jobs ist erlaubt.
Der YDS-Algorithmus, der benannt ist nach seinen Erfindern Yao, Demers und Shenker, liefert einen guten Ansatz, wie man solche Scheduling Probleme lösen kann. (10)
Er läuft in Iterationsschritten ab, in denen zunächst die Berechnung des Zeitintervalls $I = [t, t']$ mit maximaler Dichte Δ_i erfolgt. Die Laufzeit zur Berechnung dieses Intervalls beträgt $O(n^2 * logn)$, wobei aber noch weitere Optimierungen vorgenommen werden können, wenn die Zeitintervalle der Jobs in einer Baumstruktur abgespeichert werden.
Zur Berechnung der Dichte wird folgende Formel verwendet:

$$\Delta_i = \frac{1}{|I|} * \sum_{J_i \in S_I} w_i$$

Sobald ein solches Zeitintervall gefunden ist, wird ein partieller Schedule nach der Earliest-Deadline-First-Strategie erstellt. Die EDF-Strategie bevorzugt zu jedem Zeitpunkt denjenigen Job mit der geringsten Deadline und bringt ihn zur Ausführung. Danach beginnt der YDS-Algorithmus mit der nächsten Iteration. Wenn kein weiteres Intervall gefunden werden kann terminiert der Algorithmus.
Das nachfolgende Beispiel veranschaulicht diesen Algorithmus anhand eines Falles mit fünf Jobs:

- blau $J_1 = (0, 25, 9)$
- rot $J_2 = (3, 8, 7)$
- orange $J_3 = (5, 7, 4)$
- dunkelgrün $J_4 = (13, 20, 4)$
- hellgrün $J_5 = (15, 18, 3)$

Abbildung 6: Beispiel: Scheduling mittels YDS-Algorithmus



Hier laufen drei Iterationen des YDS-Algorithmus ab:

1. Berrechnung von I = [3, 8] als Zeitintervall mit der höchsten Dichte $\Delta_I = 11$.
   Menge der zu erledigenden Jobs S = $\{J_2, J_3\}$.
   Preemption von J_2 aufgrund der EDF-Strategie bei Sekunde 5.

2. Berrechnung von I = [13, 20] als Zeitintervall mit der höchsten Dichte $\Delta_I = 7$.
   Menge der zu erledigenden Jobs S = $\{J_4, J_5\}$.
   Preemption von J_4 aufgrund der EDF-Strategie bei Sekunde 15.

3. J_1 bleibt als einziger Job und wird auf die übrigen Zeitslots aufgeteilt.

3.5 Aktuelle Forschungen

Die bisher betrachteten Optimierungsmethoden betreffen ausschließlich Single-core-Systeme, die Erforschung der Performanz in Multicore-Rechnern befindet sich noch in den Anfängen. Albers et al.[3] haben begonnen, deadline-basiertes Scheduling auf Systemen mit m identischen, parallelen Prozessoren zu untersuchen. Ihr Ziel ist es dabei den Gesamtenergieverbrauch zu minimieren.
Dabei bewiesen sie zunächst, dass die Komplexität zur Berechnung von optimalen Schedules als offline-Problem, bereits für jobs einheitlicher Größe NP-hart ist. Eine effiziente Berechnung der Lösung existiert also nicht.
Im nächsten Schritt entwickeln sie jetzt offline-Algorithmen mit polynomialer Laufzeit, die für jedwede Eingabe einen Energieverbrauch, der maximal um einen konstanten Faktor vom absoluten Optimum abweicht, aufweist.
Außerdem arbeiten Lam et al.[6] an der Entwicklung von deadline-basiertem Scheduling-Strategien auf Systemen mit zwei geschwindigkeitsbeschränkten Prozessoren. Sie haben dafür eine Methode gefunden, die konstant competitive ist bezüglich des maximalen Datendurchsatzes und der minimalen Energieaufnahme.

4 Fazit und Ausblick

Dieser Einblick in das Themengebiet Performanz, mit speziellem Fokus auf den zeitlichen Aspekt und das Leistungsmanagement von Prozessoren, mit dem korrelierenden Energieverbrauch, macht deutlich, dass es sich hierbei um ein komplexes Feature für Software und Hardware handelt.
Ein System, das sich durch besonders hohe Performanz auszeichnet, muss von Grund auf so designed werden, und trotzdem ist es schwierig vorherzusagen, wie performant das System in seiner Gesamtheit arbeiten wird. Performanz erfordert zusätzlichen Quellcode bzw. Hardwareaufwand und leicht zu ermittelnde Maßeinheiten für alle performanz-relevanten Größen, sodass eine nachträgliche Optimierung möglich wird.
Besonders im zweiten Teil dieser Arbeit, wo es um die Abwägung von Prozessorgeschwindigkeit und Energieverbrauch geht, wird deutlich, dass in diesem Forschungsbereich noch viel Potenzial liegt.
Ein zukünftig relevantes Gebiet wird hier der Einbezug der Verzögerungszeiten, die entstehen, wenn ein System von einem Sleep-Zustand in den Akiven-Modus übergeht.
Desweiteren müssen die derzeitigen Speed-Scaling-Strategien speziell auf die im Vormarsch befindlichen Multicore-Systeme eingehen.
Ein anderer Ansatzpunkt für künftige Forschungen wird es sein, die bereits bekannten Algorithmen in diesem Bereich auf ihre Laufzeiten hin zu untersuchen. Interessant ist dabei vor allem, wieviel Overhead an Energie es kostet, diese Algorithmen unter realistischen Bedingungen laufen zu lassen.

Literatur

[1] ACPI *Description of ACPI power management architecture http://www.microsoft.com/whdc/system/pnppwr/powermgmt/default.mspx*

[2] S. Albers. *Energy-Efficient Algorithms*, *Communication of the ACM*, Volume 53, May 2010

[3] S. Albers, F. Müller, S. Schmelzer *Speed scaling on parallel processors*, *In Proceedings of the 19th ACM Symposium on Parallelism in ALgorithms and Architectures*, 2007

[4] General Electric Company *What is Six Sigma? The roadmap to customer impact*, *http://www.ge.com/sixsigma/SixSigma.pdf*

[5] S. Irani, S.K. Shukla, R.K. Gupta *Online Strategies for dynamic Power management in systems with multiple power-saving states*, *Communication of the ACM*, 2003

[6] T.-W. Lam, L.-K. Lee, I.K.-K. To, P.W.H. Wong *Energy efficient deadline scheduling in two processor systems*, *In Proceedings of the 18th International Symposium on Algorithms and Computation*, 2007

[7] C. Millsap. *Thinking Clearly About Performance, Part1*, *Communication of the ACM*, Volume 53, September 2010

[8] C. Millsap *On the importance of diagnosis before resolving*, *http://carymillsap.blogspot.com/2009/09/on-importance-of-diagnosis-before.html*, 2009

[9] P. Ranganathan *Recipe for Efficiency: Principles of Power-Aware Computing*, *Communication of the ACM*, Volume 53, April 2010

[10] F.F. Yao, A.J. Demers, S.A. Shenker *scheduling model for reduced CPU energy*, *Proceedings of the 36th IEEE Symposium on Foundation of Computer Science*, 1995