Parallelisierung von Echtzeitanwendungen

Ausarbeitung im Rahmen des KvBK Seminars

Lukas Lehnert
Friedrich-Alexander Universität Erlangen-Nürnberg

Zusammenfassung

Angesichts der Wirtschaftlichkeit von Mehrkernprozessoren und deren zunehmenden Popularität sollte man sich bei Echtzeitanwendungen den Herausforderungen bei der Parallelisierung stellen. Auf unteilbare Betriebsmittel darf nicht gleichzeitig zugegriffen werden, kritische Abschnitte müssen geschützt und das System muss sicherheitskritische Aufgaben mit hoher Priorität in jedem Fall rechtzeitig erledigen. Insbesondere bei sicherheitskritischen Aufgaben dürfen Hardwareanomalien, die u.a. durch die Parallelisierung auftreten können, nie dazu führen, dass Deadlines der jeweiligen Aufgabe verletzt werden.

1. EINFÜHRUNG

Im Gegensatz zu sequentiellen oder pseudoparallelen Abläufen können Mehrkernprozessoren nicht immer voll ausgelastet werden, wenn noch unbearbeitete Arbeitsaufträge vorliegen. Eine theoretische Obergrenze definiert der sogenannte Speedup [7, S. 2-3]. Der erhöhte Verwaltungsaufwand durch Synchronisation, Kontextwechsel und Hardwareanomalien wurde dabei nicht berücksichtigt. Dennoch will man aus wirtschaftlichen Gründen auf Mehrkernprozessoren setzten, da nach der Regel von Pollack dem Leistungszuwachs in einem Prozessor physikalische Grenzen gesetzt sind. So fällt bei einem quadratischen Zuwachs der Komplexität der Schaltung der Leistungszuwachs nur linear aus [9].

Während eine Terminverletzung bei Echtzeitanwendungen mit weichen und festen Echtzeitanforderungen tolerierbar ist, führt eine Terminverletzung bei harten Echtzeitanforderungen zwingend zu einer Ausnahmebehandlung [7, S. 13].

Um Rechtzeitigkeit bei parallelen Anwendungen auch unter harten Echtzeitanforderungen garantieren zu können, wird man mit dem folgenden Problemen konfrontiert. Gemeinsame unteilbare Betriebsmittel müssen verwaltet, kritische Abschnitte müssen synchronisiert und die Folgen der Synchronisationsmechanismen auf das Echtzeitsystem dürfen nicht außer Acht gelassen werden.

Einführend wird die Umsetzung von Parallelisierung in Programmiersprachen und dazugehörigen externen Bibliotheken untersucht. Als kapitelübergreifendes Beispiel wird verglichen, wie ein periodischer Arbeitsauftrag realisiert wird. Im Anschluss gehe ich auf grundlegende Konzepte der Synchronisation und Ablaufplanung am Beispiel von Ada ein.

2. SPRACHKONSTRUKTE VS. BETRIEBSSYSTEMSCHNITTSTELLE

Nun werden die weit verbreiteten Programmiersprachen Java und C/C++ betrachtet.
Die Sprache Java eignet sich durch die Java Virtual Machine nicht für Echtzeitprogrammierung. Dazu gehören zum Beispiel das dynamische Laden von Klassen und Garbage Collection- eine automatische Speicherbereinigung nicht mehr benötigter Speicherbereiche. Beides ist problematisch, weil es in Java zur Laufzeit zu nicht deterministischen, beliebig langen Verzögerungen führen kann [8]. Um trotzdem Rechtzeitigkeit zu gewährleisten wurde Real-Time Java entwickelt, das als Programmierschnittstelle Mechanismen zur Echtzeitprogrammierung anbietet. Somit wird auch Java affinen Programmieren ein leichter Einstieg in Echtzeitprogrammierung ermöglicht.
C/C++ ist durch die sehr starke Hardwarenähe als Programmiersprache auch unter harten Echtzeitanforderungen verwendbar. Allerdings gibt es in der Sprache bzw. deren Standardbibliothek keine Sprachkonstrukte oder Methoden zur Parallelisierung und Ablaufplanung in Echtzeitanwendungen. Dafür muss man auf externe Bibliotheken zurückgreifen. Diese Arbeit beschränkt sich dabei auf die opensource Laufzeitumgebung eCos.
Bei beiden Programmiersprachen werden notwendige Mechanismen zur Parallelisierung in Echtzeitsystemen nicht direkt in der Sprache berücksichtigt. Ada enthält derartige Mechanismen.
Im Folgenden ist es das Ziel, mehrere periodische Arbeitsaufträge zu realisieren.

2.1 Fäden

Um mehrere Arbeitsaufträge gleichzeitig bearbeiten zu können, benötigt man mehrere Fäden. Unterschiede gibt es bei der Erstellung und dem Start der Fäden. Dabei kann es notwendig sein, Fäden explizit starten zu müssen. Ansonsten übernimmt der Compiler diese Aufgabe. Ressourcen wie den Stack oder Verwaltungsdaten explizit anlegen zu müssen ist mehr Programmieraufwand und fehleranfälliger. In Ada wird ein Faden durch das `task` Schlüsselwort nativ in der Sprache erstellt. In eCos oder Real-Time Java sind dafür keine Schlüsselwörter reserviert. [1, S.22-29], [6, S.37-39], [7, S.107-112]. Tabelle 1 veranschaulicht den Vergleich.

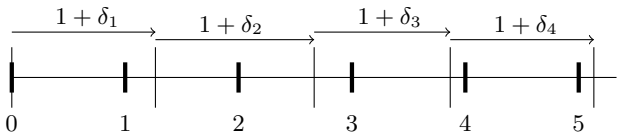
2.2 Periodische Abläufe

Um periodisch Arbeitsaufträge zu bearbeiten, kann man die Ausführung eines Arbeitsauftrags eine bestimmte Zeit verzögern. Dazu unterscheidet man zwischen relativer und absoluter Verzögerung. Relative Verzögerung wartet ab dem Zeitpunkt des Aufrufs eine bestimmte Zeit. Absolute Verzö-

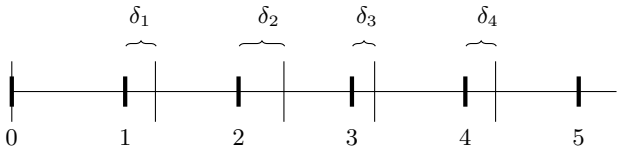
	Speicher explizit	expliziter Start
Real-Time Java		x
eCos	x	x
Ada		

Tabelle 1: Es wird der Aufwand verglichen, der mit der Erstellung von Fäden einhergeht. Da in Ada der Speicher an den `task` gebunden ist, ist eine explizite Anforderung von Speicher nicht erforderlich. Der `task` wird nach der Deklaration implizit gestartet.

gerung wartet auf einen bestimmten Zeitpunkt in Abhängigkeit vom Systemstart.
Sowohl bei absoluter als auch bei relativer Verzögerung entsteht durch die Ablaufplanung und durch einen Kontextwechsel zusätzlich eine kleine Verzögerung. Dieser Fehler ist nicht vermeidbar. Bei relativer Verzögerung tritt der Effekt auf, dass sich der Fehler auf die nächste Verzögerung auswirkt, während bei absoluter Verzögerung der Fehler davon unabhängig ist [7, S. 176-180]. Wie sich solch ein kleiner Fehler kumulieren kann zeigt Abbildung 1.



a) Phasenverschiebung bei relativer Verzögerung



b) Phasenverschiebung bei absoluter Verzögerung

Abbildung 1: Bei relativer Verzögerung kumulieren sich die Fehler. Bei absoluter Verzögerung ist der Fehler lokal und wirkt sich nicht auf die folgenden Phasen aus. [7, S. 301]

Um den kumulativen Fehler zu vermeiden, will man auf absolute Verzögerungen zurückgreifen.
eCos bietet zwar mit `cyg_thread_delay` eine relative Verzögerung an. Eine explizite Funktion zur absoluten Verzögerung wird allerdings nicht explizit angegeben [6, S.45-46]. Stattdessen muss mit Signalen vom Taktgeber gearbeitet werden. Dies bedeutet auch hier wieder einen Mehraufwand bei der Programmierung.
In Real-Time Java kann auf eine Klasse für absolute bzw. relative Zeit zurückgegriffen werden, auf die mit `sleep` gewartet wird [1, S.25-26].
Ada hat hier den Vorteil, dass mit den `delay` bzw. `delay until` Schlüsselwörtern, die nativ von der Sprache unterstützt werden, die Modellierung periodischer Aufgaben mit sehr wenig Programmieraufwand realisiert werden kann [7, S. 276].

3. RESSOURCE SPEICHER

Der in Kapitel 2 beschriebene periodische Ablauf erfordert die Verwaltung von Speicher, der dynamisch zur Laufzeit oder vor der Laufzeit statisch allokiert wird.

3.1 Statische Speicherverwaltung

Bei der statischen Allokation ist der Speicherbedarf zur Compilezeit bekannt. Um Arbeitsaufträge getrennt voneinander betrachten zu können empfiehlt es sich, für jeden Faden Verwaltungsdaten und einen Stack statisch anzulegen. In Tabelle 1 ist erkennbar, dass es bei Ada keiner expliziten Anforderung von Speicher für einen Faden bedarf. Das heißt der Compiler legt einen Standardwert für die Stackgröße fest. Da dieser Wert in einigen Fällen nicht mit dem Platzbedarf übereinstimmt, kann in Ada mittels `pragma Storage_Size` die Größe eines Stacks für einen Faden angepasst werden [7, S. 276]. Das untermalt die enge Verbindung von Funktionalität und Sprachkonstrukten in Ada. Da der Compiler direkt die Ressourcen verwaltet, kann man bei einer Analyse diese Erkenntnisse nutzen und es erleichtert dem Programmierer den Verwaltungsaufwand.

3.2 dynamische Speicherverwaltung

Die dynamische Allokation von Speicher erfolgt entweder auf dem Stack oder dem Heap. Bei der dynamischen Erweiterung des Stacks kann es zu Überläufen kommen. Diese statisch festzustellen ist nicht immer möglich.

Eine Allokation auf dem Heap erfolgt durch eine Referenz auf einen ungenutzten Speicherbereich, der global definiert ist. Die globale Nutzung erzeugt weitere inhärente Interferenzen wie zum Beispiel externen Verschnitt. Die Möglichkeit in Ada, separate Speicherpools zu definieren, gibt dem Programmierer etwas mehr Kontrolle über die Allokation auf dem Heap [7, S. 62]. In Ada und in vielen weiteren Programmiersprachen muss nicht mehr benötigter Speicher auf dem Heap explizit freigegeben werden.

4. ANALYSIERBARKEIT

Die Parallelisierung von Echtzeitanwendungen bietet einige Schwierigkeiten. Dabei gilt es, Rechtzeitigkeit und korrekte Funktionalität verifizieren zu können. Ada ist hierfür unter Anderem wegen der starken Typsicherheit geeignet. Zum Beispiel muss der Typ einer Referenz von Anfang an festgelegt sein und kann zur Laufzeit nicht mehr beliebig konvertiert werden [7, S. 57-62].
Für eine Zielarchitektur braucht man einen Compiler und eine Laufzeitumgebung, die kompatibel mit dem Ada 2005 Standard ist. Wenn diese nicht vorhanden sind oder mehrere andere Programmiersprachen verwendet werden, dann kann man stattdessen auf den Ada-POSIX Standard ausweichen. Da viele POSIX-Konstrukte optional sind, erreicht der Ada 2005 Standard eine höhere Portabilität [7, S.314-329].

4.1 Ravenscar Profil

Das Ravenscar Profil ist eine im Ada 2005 Standard in Annex D. definierte Einschränkung der Sprachkonstrukte und Ada-Bibliotheken, um Ada-Code in Analysewerkzeugen besser verifizieren zu können. So wird z.B. die relative Verzögerung wegen den in Kapitel 3.2 beschriebenen Problemen nicht zugelassen.

Das Ravenscar-Profil ist darauf ausgelegt, sequentielle Anwendungen kaum einzuschränken. Bei parallelen Anwendungen mit speziellen Anforderungen können diese allerdings nicht immer erfüllt werden. Zum Beispiel wird die Einplanungsstrategie fest vorgegeben [3, S. 436-450].

4.2 WCET in Ada

Um Rechtzeitigkeit garantieren zu können, wird die WCET - die Worst-Case Execution Time - verwendet, um auch im ungünstigsten Fall alle oder bestimmte Deadlines garantiert einzuhalten.

"From its beginnings, Ada was designed to allow such analysis in as easy a way as possible" [7, S. 355].

Das Ravenscar-Profil erleichtert die Codeanalyse. Bei der Bestimmung der WCET greift man im Wesentlichen auf zwei Ansätze zurück. Der intuitive Ansatz basiert auf Messergebnissen im sequentiellen Fall. Der zweite Ansatz ist die statische Analyse. Dabei wird ein Model des Prozessors der Zielarchitektur erstellt. Anhand dieses Modells werden für konkrete Instruktionen Zeiten angegeben, die zu einer Gesamtzeit aufsummiert werden [7, S. 353-356].

5. SYNCHRONISATION

Das Hauptproblem von Parallelisierung ist die Synchronisation kritischer Abschnitte und unteilbarer Betriebsmittel. Man unterscheidet in blockierende und nicht-blockierende Synchronisation. Für beide benötigt man einen transaktionalen Speicher. Ansonsten würden Änderungen von Variablen bloß im Cache der ausführenden CPU sichtbar sein. Dazu gibt es in Ada wie in vielen anderen Programmiersprachen das `Volatile` Schlüsselwort. Positiv fällt Ada auf, indem durch `Volatile_Components` auch einzelnen Komponenten eines Arrays diese Eigenschaft zugewiesen werden kann [7, S. 153-155].

5.1 Blockierende Synchronisation

Die einfachste und typischste Technik zur Synchronisation ist der gegenseitige Ausschluss. Dabei gilt es, Verklemmungen von vornherein zu vermeiden. Eine Möglichkeit ist es, eine der notwendigen Bedingungen für eine Verklemmung nicht zu erfüllen [5].

Auf Probleme bei der Prioritätsverletzung wird im Kapitel 7.2 eingegangen. Ein weiterer Nachteil ist, dass der Mehraufwand durch die blockierende Synchronisation vergleichsweise hoch ausfallen kann.

Konkret kann man zur Synchronisation unter Anderem Semaphore oder Monitore nutzten.

Bei dem Versuch einen kritischen Abschnitt zu betreten, der schon durch ein Semaphor gesperrt ist, wird der Faden so lange blockiert, bis der kritische Abschnitt von dem anderen Faden verlassen wurde. Wenn passiv gewartet wird, dann kann die CPU-Leistung für andere Fäden verwendet werden und spart somit Rechenzeit.

Das Monitor-Konzept ähnelt der Synchronisation mit Semaphoren. Der wesentliche Unterschied ist, dass es Prozeduren und Daten gemeinsam enthalten kann. Dabei kann nur ein Faden in den kritischen Bereich, der die Prozeduren und die Daten beinhaltet. D.h. es kann ein größerer Bereich geschützt werden. Dies wird durch eine Warteschlange für das Betreten des Moduls und eine interne Warteschlange für jede Bedingungsvariable umgesetzt. Wenn sich eine Bedingungsvariable ändert, dann kann die Bedingung auf die gewartet wird von einem oder allen wartenden Fäden geprüft werden. In Ada wird das Monitorkonzept (siehe Abb. 2) durch das `protected` Schlüsselwort realisiert [7, S. 130-143].

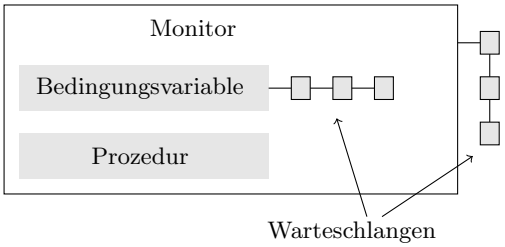


Abbildung 2: Der Monitor darf nur von einem Faden betreten werden. Eine Änderung einer Bedingungsvariable oder das Verlassen des Monitors wird der jeweiligen Warteschlange signalisiert.

5.2 Nicht-blockierende Synchronisation

Zur lock-freien Synchronisation kann man die CAS (Compare And Swap) Operation nutzen. Erst wird anhand des alten Ausgangswertes ein neuer Wert berechnet. Wenn am Ende der Berechnung der Ausgangswert in der jeweiligen Speicherzelle gespeichert ist, dann wird der neue Wert eingetragen. Ansonsten muss die Berechnung wiederholt werden. Das Prüfen und Eintragen muss atomar durchgeführt werden. Infolgedessen muss diese Operation auch durch die Hardware angeboten werden.

Im Gegensatz zu blockierender Synchronisation können bei lock-freier Synchronisation keine Verklemmungen auftreten, da nie aktiv oder passiv auf ein bestimmtes Ereignis gewartet wird.

Die Entscheidung für blockierende oder lock-freie Synchronisation muss in Abh. von der Ablaufplanung und den Anforderungen an das Echtzeitsystem gefällt werden [2].

6. ABLAUFPLANUNG

Um mehrere Arbeitsaufträge parallel zu bearbeiten muss man deren Reihenfolge festlegen. Dies ist eine Designentscheidung, die auch Auswirkungen darauf hat, was bei der Synchronisation zu beachten ist.

6.1 Ablaufplanung: vor oder zur Laufzeit

Zunächst kann man entscheiden, ob man Prioritäten von Aufgaben vor oder zur Laufzeit festlegt. Da beim fixed priority scheduling die Prioritäten vor der Laufzeit festgelegt sind, kann man in der Analyse leichter verifizieren, ob Aufgaben mit hoher Priorität in jedem Fall ausgeführt werden [7, S. 258].

Bei dynamischen Prioritäten wird eine weit aus höhere Flexibilität zu Lasten der Analysierbarkeit erreicht. Zum Beispiel wird bei Earliest Deadline First (EDF) der Faden eingelastet, der die naheliegendste Deadline hat. Dieses Verfahren ist optimal, wenn es eine Möglichkeit gibt, alle Deadlines einzuhalten. Wenn das System durch zu viele Arbeitsaufträge überlastet wird, verhält sich das System nicht deterministisch, weswegen besonders wichtige Arbeitsaufträge nicht garantiert erfüllt werden können [7, S. 276-277].

Ein Ablaufplan kann vor (engl. offline scheduling) oder zur Laufzeit (engl. online scheduling) erstellt werden [7, S. 258-259]. Vor der Laufzeit festgelegte Ablaufpläne sind besser verifizierbar. Viele Echtzeitanwendungen sind allerdings zu komplex, um solch einen Ablaufplan erstellen zu können. So muss man bei fester Ablaufplanung auf asynchrone Ereignisse wie z.B Usereingaben verzichten, da der Ablauf, wann

welche Aufgabe abgearbeitet wird, vor der Laufzeit festgelegt wurde.

6.2 unkontrollierte Prioritätsinversion

Wenn mehrere Fäden unterschiedlicher Priorität auf eine gemeinsame Ressource zugreifen, kann es zu einer Prioritätsinversion kommen. Dieses Problem tritt sowohl bei echter Parallelität als auch bei Pseudo-Parallelität auf. Abbildung 3 zeigt eine Situation, die zu Prioritätsinversion führt.

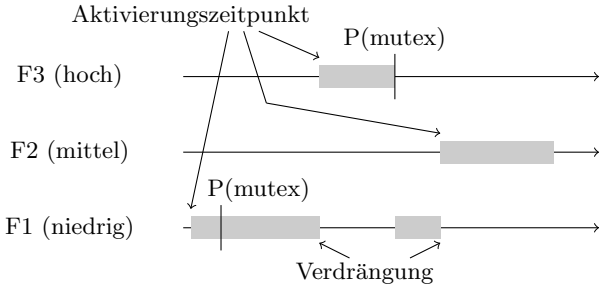


Abbildung 3: In dieser Situation tritt eine Prioritätsinversion auf, indem ein Faden hoher und niedriger Priorität auf das gleiche Betriebsmittel zugreifen. Der Faden mit niedriger Priorität wird von einem Faden mittlerer Priorität verdrängt. Dennoch wartet ein Faden mit hoher Priorität auf Freigabe des kritischen Abschnittes. [7, S. 279]

6.2.1 *Immediate Ceiling Priority Protocol*

Eine Lösungsmöglichkeit des Problems für Einkernprozessoren bietet das Immediate Ceiling Priority Protocol (ICPP) an. Während ein Faden ein unteilbares Betriebsmittel besitzt wird diese temporär auf die Prioritätsgrenze einer Ressource gesetzt. Nach Freigabe des Betriebsmittels wird die Priorität auf den ursprünglichen Wert zurückgesetzt. Die Prioritätsgrenze eine Ressource ist die maximale Priorität aller Fäden, die das unteilbare Betriebsmittel verwenden [7, S. 278-282].

Ein dem ICPP ähnliches Protokoll wird in Ada durch `Locking_Policy (Ceiling_Locking)` umgesetzt [7, S. 306-308].

6.2.2 *Multiprocessor Resource Sharing Protocol*

In Ada bedeutet `Locking_Policy (Ceiling_Locking)` bei Mehrkernprozessoren, dass auf ein unteilbares Betriebsmittel aktiv gewartet wird, wenn ein anderer Faden das Betriebsmittel besitzt.

Eine effizientere Umsetzung liefert das Multiprocessor Resource Sharing Protocol (MrsP). Wenn ein Faden, dem ein unteilbares Betriebsmittel zugeteilt wurde, verdrängt wird, dann übernimmt ein aktiv auf das Betriebsmittel wartender Prozessor den verdrängten Faden. In Ada 2012 wurde das MrsP noch nicht umgesetzt [4].

6.3 Zusammenhang von Synchronisation und Ablaufplanung

Das Problem der Prioritätsinversion kann bei nicht-blockierender Synchronisation nicht auftreten, da der kritische Abschnitt von beliebig vielen Fäden betreten werden darf. Dafür muss berücksichtigt werden, dass insbesondere bei Aufträgen mit einer niedrigen Priorität, Berechnungen mehrmals wiederholt werden können.
Bei Einkernprozessoren mit verdrängender Ablaufplanung sind weniger Kontextwechsel erforderlich und die Synchronisation kritischer Abschnitte ist nicht immer zwingend.

7. SCHLUSS

Parallelisierung kann die Programmausführung insbesondere bei Mehrkernprozessoren deutlich beschleunigen. Die Herausforderungen, die dabei auftreten, wurden aufgezeigt und Lösungsmöglichkeiten vorgestellt. Ada eignet sich gut, um diese Anforderungen zu gewährleisten, da Mechanismen zur Parallelisierung (`task`), Synchronisation (`protected`) oder Ablaufplanung (`Ceiling_Locking`) nativ in der Sprache integriert sind.

Weiterhin müssen in Ada bestehende Konzepte gegen unkontrollierte Prioritätsinversion bei Mehrkernprozessoren umgesetzt werden, um dem Bedarf an Parallelisierung Rechnung zu tragen.

Bis jetzt bietet C++ im Sprachstandard keine nativen Sprachmechanismen für Parallelisierung in Echtzeitsystemen an. In zukünftigen Entwicklungen wird sich zeigen, ob der C++ Sprachstandard um notwendige Mechanismen erweitert wird, um effizienter, besser analysierbare parallele Echtzeitanwendungen zu schreiben.

8. LITERATUR

[1] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. The Real-Time Specification for Java. *Addison-Wesley*, 2000.

[2] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353, 2008.

[3] A. Burns and A. Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, Cambridge, 2007.

[4] A. Burns and A. Wellings. Locking Policies for Multiprocessor Ada. *ACM Ada Letters*, 33(2):59–65, 2013.

[5] E. G. Coffman, Jr., M. J. Elphick, A. Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[6] Free Software Foundation, Inc. eCos Reference Manual, http://ecos.sourceware.org/docs-3.0/pdf/ecos-3.0-ref-a4.pdf
2008.

[7] J. W. McCormick, F. Singhoff, and J. Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*. Cambridge University Press, Cambridge, 2011.

[8] Oracle Corporation. Deterministic garbage collection: Unleash the power of java with oracle jrockit real time, http://www.oracle.com/us/technologies/java/oracle-jrockit-real-time-1517310.pdf
2008.

[9] B. Venu. Multi-core processors - an overview. *CoRR*, abs/1110.3535, 2011.