Software-Entwicklung für Multicore-Systeme

Was gibt es Neues, und wo geht die Reise hin?

André Schmitz, Green Hills Software

Viele Embedded Systeme verwenden bereits heute Multicore Prozessoren, und dieser Anteil steigt stetig. Die Entwicklung und Software Migration auf diese Architekturen wird immer leichter, und es gibt bereits sehr ausgereifte Technologien zur Entwicklung von Software für Multicore Systeme. Dieser Beitrag wirft einen Blick auf die Technologien im Bereich Code-Generierung, Echtzeitbetriebssysteme und Debugging die die Entwicklung von Multicore Software erleichtern. Außerdem wird ein Blick in die Zukunft gewagt und gefragt, wie diese Technologien mit der zu erwartenden steigenden Anzahl von Cores skalieren.

Warum Multicore?

Es gibt verschiedene Gründe für den Umstieg auf Multicore Systeme. Meist ist das Ziel mithilfe einer Multicore CPU eine höhere Performance zu erreichen, eine Performance, die man in vergleichbarer Weise nicht durch Erhöhung der Taktrate eines Cores erreichen könnte. Denn im Vergleich zu einem hoch getakteten Singlecore System braucht zum Beispiel ein herunter getaktetes Dualcore System deutlich weniger Strom und bietet dabei aber mehr Rechenleistung. Ein weiterer Grund für den Einsatz von Multicore könnte der Wunsch sein mehrere Betriebssysteme auf einer CPU laufen zu lassen. Auf einem Dual-Core könnte ein Core mit Linux und ein Core mit einem RTOS laufen. Heutzutage spricht man in diesem Zusammenhang oft von Virtualisierung, früher wurde hier auch von „Asymmetric Multicore Processing“ (AMP) gesprochen (siehe unten).

Es gibt auch Anwendungsfälle in denen Multicore zur Erreichung von Funktionaler Sicherheit eingesetzt wird. Eine typische Konfiguration ist das Lockstep Prinzip bei dem zwei ähnliche Cores die gleichen Algorithmen ausführen und am Ende geprüft wird, ob das Ergebnis beider Berechnungen identisch ist [1]. Dies erhöht die Hardware Fehlertoleranz. Manchmal wird auch versucht zwei Software Komponenten auf verschiedenen Cores laufen zu lassen und argumentiert dann, dass diese dadurch wechselwirkungsfrei liefen. Dies ist natürlich nur dann der Fall, wenn diese Cores keine gemeinsam genutzten Ressourcen haben (Caches, Bus, etc.), was in den meisten Multicore Hardware Systemen aber nicht der Fall ist (siehe locker vs. eng gekoppelte Systeme weiter unten).

Multicore Hardware

In der Regel kann zwischen heterogenen Systemen mit unterschiedlichen Cores und homogenen Systemen mit gleichen Cores unterschieden werden. Heterogene Systeme werden vorwiegend für in sich asymmetrische Aufgaben verwendet, bei denen zum Beispiel ein General Purpose Prozessor (z.B. ARM) die Benutzerschnittstelle und die Steuerung des Systems übernimmt, während ein Rechenprozessor (z.B. DSP oder NPU) die effiziente und schnelle Datenverarbeitung und Ein-/Ausgabe sicherstellt. Gerne wird dabei der Zugriff auf bestimmte Peripherie nur einzelnen Cores gewährt. Bei heterogenen Systemen liegt also der Vorteil in der zweckgebundenen Leistungsfähigkeit der Cores. Homogene Systeme hingegen

bieten eine elegante und energiesparende Lösung zur Vergrößerung der Rechenleistung eines Systems.

Betrachtet man die Anbindung der Cores an den Speicher, so unterscheidet man hier zwischen eng gekoppelten Systemen, bei denen alle Cores auf den gleichen Speicher Zugriff haben, und locker gekoppelten Systemen, bei denen jeder Core seinen eigenen Speicher hat. Bei Letzteren findet die Kommunikation in der Regel nicht direkt über „Shared Memory“, sondern über andere Kanäle statt. Bei eng gekoppelten Systemen skaliert wiederum die Leistung nicht proportional mit der Anzahl der Prozessoren, da der Bus hier meist zu einem Engpass wird und die Cache Kohärenz sichergestellt werden muss.

Dieser Performance-Engpass kann mithilfe spezieller Hardwarearchitekturen verbessert werden. Eine mögliche Architektur ist die Non-Uniform Memory Architecture (NUMA), bei der jeder Core lokalen, schnellen Speicher hat, dessen Zugriffe nicht durch Bus-Arbitrierung gebremst werden. Der Zugriff auf den Speicher der anderen Cores ist über eine High-Speed Verbindung realisiert, die zwar Cache-kohärent ist, jedoch deutlich langsamer als der Zugriff auf den lokalen Speicher.

Programmierung von Multicore Systemen
Bei der Softwareentwicklung für Multicore-Systeme ist es erforderlich die zu erledigenden Aufgaben sinnvoll auf die Cores zu verteilen. Bei heterogenen Systemen ist diese Verteilung deutlich offensichtlicher und muss vom Entwickler manuell vorgenommen werden. Die Programmierung dieser Systeme ist aber eher aufwändig, da man oft unterschiedliche Tools (Compiler und Debugger) für die unterschiedlichen Cores benötigt und die Interprozessor-Kommunikation in der Regel Hardware spezifisch realisiert werden muss. Falls Betriebssysteme zum Einsatz kommen, sind diese meist auf den beteiligten Cores unterschiedlich, was einen weiteren Schwierigkeitsgrad hinzufügt, wenn es um die Interoperabilität geht.

Betriebssystem Lösungen
Bei homogenen Multicore Systeme findet man meist zwei unterschiedliche Ansätze die Aufgaben auf die Cores zu verteilen (AMP und SMP), welches Nachfolgend beschrieben werden.

AMP
Schaut man sich die Fachliteratur an, so findet man teilweise ganz unterschiedliche Auffassungen davon was Asymmetrischen Multi Processing (AMP) wirklich bedeutet. Manche verwenden den Begriff AMP für heterogene Hardware Architekturen, in denen nicht alle Cores Zugriff auf die gleichen Ressourcen haben [2]. Andere betrachten es als reine Software Überlegung bei homogenen Hardware Architekturen [3]. Ich verwende AMP hier jetzt nur immer als Software Aspekt auf ansonsten homogener Multicore Hardware. Letztlich entscheidet aber bei allen Formen von AMP der Entwickler über die Partitionierung und Aufteilung der Funktionen zwischen den Cores. Auf jedem Core läuft eine eigenständige und unabhängige Software, die mit oder ohne Betriebssystem realisiert sein kann. Falls Betriebssysteme verwendet werden, müssen diese nicht notwendigerweise auf allen Cores identisch sein. Letzteres wird heute auch gerne Virtualisierung genannt,

wenngleich man unter Virtualisierung wiederum auch etwas ganz anderes verstehen kann.

Ein AMP Software Ansatz kann hilfreich sein, wenn man für bestimmte Aufgaben Rechenzeit garantieren möchte, wenngleich diese Garantien aufgrund der unter Umständen geteilten Ressourcen nicht wirklich gegeben sind. Leider sind Anwendungen auf AMP Systemen aber meist weniger portabel und es besteht keine Möglichkeit zur Laufzeit eine dynamische Lastverteilung zu erreichen. Ähnlich wie bei heterogenen Systemen benötigt man auch hier einen fehlertoleranten Nachrichtenaustausch zwischen den Cores.

SMP

Symmetrisches Multi Processing (SMP) ist nur möglich auf homogenen, eng gekoppelten Systemen unter Verwendung eines SMP-fähigen Betriebssystemen (OS). Eine Instanz des Betriebssystems läuft auf allen Cores und übernimmt die Aufgabe der Verteilung von Threads auf die Cores. Dadurch ist es prinzipiell möglich, eine existierende Multi-Threaded Anwendung, die bisher auf einem Singe-Core läuft, aber bereits auf einem SMP-fähigen OS implementiert wurde, ohne weiteres mit deutlichem Performancegewinn auf einem Multicore-System auszuführen. Das OS verteilt Threads automatisch zwischen den Cores, und alle Ressourcen und internen Abläufe werden transparent vom OS überwacht [4]. Natürlich gibt es hier einige Fallstricke, die es zu beachten gibt.

Als weitere Stärke von SMP-Systemen zählt die Teilung von Ressourcen. Diese ermöglicht ein hohes Maß an Flexibilität, kann aber durch die echte Nebenläufigkeit von Threads auch zum Problem der Ressourcen-Konkurrenz führen. In SMP-Systemen können Threads mit unterschiedlicher Priorität echt gleichzeitig ausgeführt werden. Ist also jemand bei der Entwicklung der Singlecore Anwendung bisher davon ausgegangen, dass ein niedrig priorisierter Thread nur dann laufen kann, wenn kein höher priorisierter läuft, dann könnte er bei einem SMP-System plötzlich Überraschungen erleben. Es ist ratsam ein SMP-OS mit Memory Management Unit (MMU) Unterstützung zu nutzten, um damit die Granulierung der Software direkt auf Prozess-Ebene realisieren zu können, und nicht auf Thread-Ebene. Damit werden ärgerliche Fehler vermieden, die beim Multi-Threading gerne auftreten. Man verwendet dann die MMU zur Isolation vom OS Komponenten und definiert saubere Schnittstellen zwischen den Komponenten.

Möchte man Echtzeit-Verhalten realisieren, bedarf es auch auf dem Multicore-System eines Echtzeitbetriebssystems (RTOS). Dieses sollte sinnvollerweise eine kleine, deterministische Interrupt-Latenz haben und den Determinismus einer Applikation beim Umstieg auf Multicore beibehalten. Neben dem Determinismus sollte das OS auch gut skalieren, wenn weitere Cores hinzugefügt werden. Diese beiden Eigenschaften hängen meist von der Art der Kernel Parallelisierung ab, also welche Art von „Locks“ zum Schutz der kritischen Bereiche des Kernels implementiert sind. Es gibt hier sehr gut skalierende Ansätze, wie zum Beispiel die des Linux Kernels, die aber vergleichsweise undeterministisch sind. Die bessere Echtzeitfähigkeit und Zuverlässigkeit eines RTOSs wird hingegen manchmal auf Kosten der Skalierbarkeit erreicht.

Scheduling und Core Affinität
Nicht zuletzt wird, wie bereits erwähnt, mit steigender Zahl der Cores der Speicher-Bus zum Flaschenhals. Um wenigstens die Caches der Cores effektiv zu nutzen, macht es Sinn die Threads an bestimmte Cores zu binden (Affinität). Hier kann man z.B. über Betriebssystemaufrufe Threads statisch an Cores binden (Benutzerdefinierte Affinität). Dies reduziert zwar die Flexibilität der Lastverteilung zur Laufzeit, kann aber zum Beispiel bei der Verarbeitung von Interrupts helfen, indem der vom Interrupt aufgeweckte Thread auf den Core gelegt wird, der auch den Interrupt behandelt. In diesem Fall wird zum einen die Performance durch eine effizientere Nutzung des Caches verbessert, zum anderen kann auf einen Inter-Prozessor-Interrupt (IPI) verzichtet werden.

Ein SMP-OS kann darüber hinaus versuchen, Threads automatisch immer auf dem gleichen Core auszuführen, falls das möglich ist („Natürliche Affinität“). Hierbei wird bei Beibehaltung der Flexibilität und der Prioritätsgarantien zur Laufzeit die Cachenutzung verbessert und damit die Performance erhöht. Letztlich kann man mit der Core Affinität ein SMP System zum Teil in die Richtung Software AMP konfigurieren, und das kann man dann als Bound Multicore Processing (BMP) bezeichnen.

Automatische Parallelisierung
Es gibt darüber hinaus auch Ansätze zur automatisierten Parallelisierung von Code mithilfe von Bibliotheken. In diesem Bereich hat sich in den Letzte Jahren einiges getan. Diese sind zwar meist einfach zu nutzen und skalieren gut, gehen aber alle mit gewissen Einschränkungen einher. Einige funktionieren nur sinnvoll auf großen Schleifen (z.B. bei Verwendung von OpenMP) oder erfordern ein explizites Design mit Hilfe von C++ Klassen (Parallelization objects) [5]. Natürlich helfen auch die neueren C++ Versionen (C++11 und neuer) um Threading auf SMP zu realisieren. Es gibt mehr und mehr Tools die helfen sollen existierenden, seriellen Code vollständig automatischen zu parallelisieren. Für Sprachen wie Fortran scheint das ganz gut zu funktionieren, weil Fortran strengere Garantien bzgl. Speicher Überlappung (Aliasing) aufweist als die Sprachen C oder C++. Letztere erlauben indirekte Adressierung, Rekursionen und viele andere dynamische Aspekte, die es einem Parallelisierungstool schwer macht vollkommen automatisch zu parallelisieren. Die existierenden Lösungen am Markt erfordernd meist eine Unterstützung des Entwicklers oder können eben nur ganz bestimmte Aspekte eines Programmes parallelisieren, wie die besagten Schleifen [6].

Speichermodelle
Das Verhalten von Speicherzugriffen auf Multicore Systemen muss auch im speziellen betrachtet werden. Neben der offensichtlichen Notwendigkeit zum Schutz geteilter Ressourcen vor nebenläufigem Zugriff (Thread Safety), z.B. mithilfe von Atomaren Operationen oder Mutexen, muss man insbesondere beachten wie sich der Zugriff auf Speicher von verschiedenen Cores zueinander verhält. Die Reihenfolge in der ein Core auf Speicher schreibt kann völlig anders sein als die Reihenfolge in der ein anderer Core diese Zugriffe im Speicher sieht. Hier geht es nicht um das möglicherwiese stattfindenden umsortieren von Instruktionen eines Compilers zur Optimierung der Ausführungsgeschwindigkeit, sondern es geht um die Optimierung der Hardware beim Zugriff auf den Speicher. Um Problem aufgrund dieses Umsortierens zu vermeiden, bieten die Instruktionssatz Architekturen spezielle

Barrier Instruktionen an, die es erlauben verschiedene Arten von Speicherzugriffen vor der Barrier garantiert auszuführen zu lassen bevor andere Arten von Speicherzugriffen nach der Barrier stattfinden. Auch hier gibt es dann auch wieder Unterstützung von den neuen C++ Varianten. Der Software Entwickler sollte also auf jeden Fall die Probleme im Zusammenhang mit Speichermodellen kennen. [7]

Entwicklungswerkzeuge

Weitere Herausforderungen in Umgang mit Multicore Systemen sind die Vermeidung von Softwarefehlern, die Suche nach den nicht vermeidbaren Fehlern und die Optimierung des Codes. Aus meiner Sicht ist die Suche nach Software Fehlern immer noch eine der größten Herausforderungen während der Softwareentwicklung. Ein Multicore-System macht die Sache dann aufgrund der höheren Komplexität nicht wirklich einfacher. Umso mehr ist hier also ein strukturierter Entwicklungsprozess wichtig der hilft Fehler von vornherein zu vermeiden oder so einfach und früh wie möglich zu finden. Bereits bei Singlecore Systemen kennt man die Probleme beim Umgang mit virtueller Nebenläufigkeit (z.B. Race Conditions) und diese werden nicht einfacher, wenn man mit echter Nebenläufigkeit zu tun hat. Daher ist es essentiell, die passenden Entwicklungswerkzeuge zu haben, die die Softwareentwicklung für Multicore-Systeme effizient unterstützt.

Für die Treiber-Entwicklung sind zum Beispiel solche Hilfsmittel optimal, die eine synchrone Laufzeitsteuerung (Stop-Mode) aller Cores des Systems ermöglichen, um mehrere Cores gleichzeitig zu starten und anzuhalten. Falls ein Betriebssystem verwendet wird, ist darüber hinaus die Möglichkeit zum Anwendungs-Debugging (Run-Mode) nützlich, bei dem auf Thread-Level zur Laufzeit Programmfehler gesucht werden können, ohne den Rest des Systems anhalten zu müssen. Zum besseren Verständnis des Laufzeitverhaltens der nebenläufigen Prozesse ist ein Werkzeug zur Analyse der Betriebssystem-Events sowohl bei AMP- als auch bei SMP-Systemen unerlässlich. Soll eine nicht-intrusive Analyse des Programmablaufes durchgeführt werden, die eine einfache Möglichkeit zur Suche von versteckten oder schwer reproduzierbaren Software-Fehlern bietet, sind Werkzeuge zur Multicore Trace-Erfassung und Trace-Analyse nötig [6]. Optimal ist es natürlich, Fehler automatisiert zu finden. Dies erreicht man entweder durch statische Source-Code Analyse oder beim Ausführen des Programms durch automatische Laufzeit-Fehlersuche.

Ausblick

Der Trend zu mehr Multicore, speziell im Embedded Bereich, wird sicherlich weitergehen, alleine um die Gier nach Performance bei gleichzeitig geringer Leistungsaufnahme stillen zu können, und wir müssen uns überlegen, wie wir die damit verbundenen Herausforderungen meistern werden. Bzgl. Hardware wird es vermutlich vorwiegend Weiterentwicklung geben in drei Richtungen

1) Funktionale Sicherheit durch Lockstep
2) anwendungsspezifische heterogene Architekturen
3) homogene high-end Multicore SOCs

Was die Code Generierung angeht, können einzelne Bereiche von der automatischen Codegenerierung profitieren, doch die Software Entwickler und System Architekten werden immer noch genug mit dem Thema Multicore-Software Architektur zu tun haben (Nebenläufigkeit, Memory Modelle, etc.). Wir müssen sicherstellen, dass

Betriebssysteme für Multicore ausreichend gut skalieren. Beim Tooling sind wir auf einem guten Weg, wenn es aber um das den für die Analyse von Komplexen Systemen sehr wichtigen Aspekt des „Programm-Tracing“ geht, müssen wir mit immer größeren Datenmengen pro Sekunde rechnen. Die aktuellen Hardware Trace Lösungen, auch wenn sie hocheffiziente serielle Protokolle verwenden, werden vermutlich nicht ausreichen um den gesamten Programmablauf ausreichend detailliert in Echtzeit abbilden zu können. Hier muss noch Arbeit geleistet werden, damit wir auch in Zukunft immer genau verstehen können was unser Programm zu welcher Zeit genau auf welchem Core tut.

Referenzen

[1] https://de.wikipedia.org/wiki/Lockstep_(Computertechnik)
[2] http://en.wikipedia.org/wiki/Asymmetric_multiprocessing
[3] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CFHBJBIE.html
[4] http://en.wikipedia.org/wiki/Symmetric_multiprocessing
[5] http://en.wikipedia.org/wiki/OpenMP
[6] https://en.wikipedia.org/wiki/Automatic_parallelization_tool
[7] https://en.wikipedia.org/wiki/Memory_ordering

Autor

Andre Schmitz erhielt sein Diplom in Physik 1997 an der Universität Bonn. Anschließend entwickelt er bei der FhG Steuerungs- und Simulations-Software für Autonome Roboter. Von 2000 bis 2005 entwickelte Herr Schmitz Embedded Software für UMTS Kommunikationssysteme. Seit 2005 ist Herr Schmitz bei Green Hills Software für die technische Unterstützung von Kunden und die Durchführung von Schulungen zuständig. Herr Schmitz ist seitdem regelmäßig Referent bei diversen Fachkonferenzen.

Kontakt

Internet: www.ghs.com
Email: aschmitz@ghs.com