Hardwarebeschleunigung von paralleler Ablaufplanung für Multicoresysteme
Zwischenbericht zum 1. Jahr

Nina Engelhardt

14. Juni 2012

Ziel meiner Arbeit ist es, durch speziell angepasste Hardware die Synchronisation und Aufgabenverteilung für parallele Programme auf Multicoresystemen zu verbessern. Als Grundlage dient dafür das VMS-System, ein Framework dass es ermöglicht, Laufzeitumgebungen für verschiedenste parallele Programmiermodelle mit nur wenig Aufwand zu implementieren.

1

In den ersten 6 Monaten habe ich wie geplant das VMS-System analysiert. VMS umfasst die Grundlegende Struktur, die allen Laufzeitumgebungen gemein ist: sie beinhaltet den Wechsel zwischen Programm und Laufzeitumgebung, und die Möglichkeit, eine Zeitliche Ordnung zwischen zwei Codepunkten in zwei virtuellen Prozessoren zu garantieren (semantikfreie Synchronisation). Auf Basis dieser Elemente muss der Laufzeitumgebungsentwickler nur noch die jeweilige Semantik der parallelen Umgebung hinzufügen. Diese wird in Form eines Plugins bereitgestellt, das zwei Teile enthält: einen Assigner, der entscheidet, welche Aufgaben als nächstes bearbeitet werden, und einen Request Handler, der die Implementation der parallelen Konstrukte liefert.

Diese zweiteilige Struktur ermöglicht zwei Ansatzpunkte für die Beschleunigung: den VMS-Kern selbst – Verbesserungen an dieser Stelle kommen allen Laufzeitumgebungen zu Gute – und das jeweilige Plugin. VMS ist minimal gehalten und bietet nur wenige Möglichkeiten zur Verbesserung, jedoch haben Änderungen hier den größten Effekt. VMS ist außerdem, da es die grundlegendsten Elemente zusammenfasst, bereits sehr nah an der existierenden Hardware und die häufigen Aufgaben sind gut unterstützt. Zusätzliche Beschleunigung kann hier nur durch sehr ressourcenintensive Lösungen (z.B. zusätzliche Registersätze für schnellen Kontextwechsel) erwartet werden. Beschleuniger für Pluginfunktionen haben deutlich mehr Spielraum, besonders für Sprachen mit komplexen, hardwarefernen Konstrukten (und dies sind die besonders interessanten, weil sie sich der Struktur des Problems nähern und damit dem Programmierer die Arbeit abnehmen, das Problem in Hardwarenahe Konzepte umzudenken bzw. übersetzen zu müssen.) Da die Pluginfunktionen aber Sprachspezifisch sind, kommen Beschleunigungen an dieser Stelle nur dem Teil der Programme zugute, die in dieser Sprache geschrieben sind. Die Herausforderung wird also sein, Hardwareelemente zu finden, die möglichst flexible und vielfältige Verwendung finden können. Haupsächlich wird aber die Programmiersprache StarSs anvisiert werden, die im Fachgebiet AES häufig Verwendung findet.

VMS ist die praktische Ausformulierung eines Modells des Parallelismus, die von Dr. Sean Halle unter dem Namen “Holistic Model of Parallel Computation”[ ] vorgeschlagen wurde. Dieses Modell isoliert die schedulingrelevanten Teile eines Programms und legt das Augenmerk besonders auf die Effekte von Scheduling Decisions auf die Performance.

Während meiner Analyse von VMS stieß ich auf mehrere unklare Stellen in der Theorie, und habe diese in Zusammenarbeit mit Dr. Halle ausgearbeitet. Insbesondere habe ich das Modell um das Konzept der Schichten erweitert, das dem Umstand Rechnung trägt, dass in einem System meistens mehrere hierarchisch untergeordnete Scheduler existieren. So werden die Arbeitseinheiten, über die die Laufzeitumgebung als Eins entscheidet, möglicherweise vom Betriebssystem noch einmal unterbrochen und verteilt, spätestens aber vom Prozessor in kleinere Einheiten – Assemblerbefehle – zerteilt, und auf unterschiedliche Funktionseinheiten aufgeteilt. Im Fall von Out-of-order Prozessoren wird deutlich, das auch dies parallele Aufgabenverteilung ist, und das Holistische Modell auch auf diese anwendbar ist. Zur Zeit arbeiten wir an einem Journal Article der das Holistische Modell detailliert darlegt.

Zu den Errungenschaften des Holistischen Modells gehören auch zwei grafische Darstellungen, die Unit Constraint Collection und der Scheduling Consequence Graph, deren Ziel es ist, die Gründe der (guten oder schlechten) Performance eines parallelen Programms ersichtlich zu machen. Die Unit Constraint Collection (kurz UCC) zeigt die Arbeitseinheiten (units) die im Programm definiert werden und die Randbedingungen (constraints) ihrer Ausfürbarkeit, die vom Programm selbst ausgehen. Der Consequence Graph (CG) repräsentiert eine realisierte Ausführung des Programms und zeigt die räumliche und zeitliche Platzierung der Arbeitseinheiten, die der Scheduler gewählt hat, sowie die unterschiedlichen Randbedingungen, die diese Wahl eingeschränkt haben. Diese können vom Programm selbst stammen, sich aus Ressourcenbegrenzungen ergeben, oder durch Beschränkungen der Laufzeitumgebung entstehen.

Nachdem ich VMS bereits zum Zweck der Analyse des Laufzeitverhaltens instrumentalisiert hatte, habe ich diese Infrastruktur wiederverwendet, um aus den gewonnenen Daten die räumliche und zeitliche Platzierung der Arbeitseinheiten im Consequence Graph abzuleiten. Zusätzlich habe ich ein VMS-Plugin, das synchrones Senden und Empfangen von Nachrichten zwischen Threads in einem Programm unterstützt (synchronous send-receive, kurz SSR), instrumentalisiert, um die Constraints zu erfassen, die für die Konstruktion der UCC und des CG notwendig sind. Schließlich erstellte ich ein kleines Programm, das diese Aufzeichnungen entgegennehmen und UCC & CG darstellen kann.

Quasi mit dem ersten Lauf stellte sich heraus, dass die Matrixmultiplikation, die ich seit Monaten als Testapplikation verwendete, einige Performanceprobleme hatte. So hatte z.B. der applikationseigene Lastenverteiler einen Bug, sodass von 40 Prozessorkernen nur 3 den Großteil der Arbeit zugeteilt bekamen, und 10 weitere einen kleinen Teil, die 27 restlichen jedoch gar keine. Des weiteren wurde die Arbeit so verteilt, dass der Kern, der den Lastenverteiler ausführt, als erster Arbeit zugeteilt bekam, und diese Arbeit dann den Verteiler für längere Zeit unterbrach, bevor er weiteren Kernen Arbeit zuteilen konnte. All dies war in den Statistiken nicht aufgefallen, in der Visualisierung jedoch offensichtlich. Als ich noch eine detailliertere Aufteilung des Overheads, der für eine bestimmte Arbeitseinheit aufgewendet wird, und mehrere Metriken (Takte, Befehle, Cache Misses) hinzugefügt hatte, wurden auch subtilere Effekte sichtbar. Dabei stellte sich z.B. heraus, dass die zentralisierte Architektur von VMS, das alle Informationen die die Laufzeitumgebung über den Zustand des Programms bereithalten muss in einem gemeinsamen Pool speicherte, auf den nur ein Kern gleichzeitig zugreifen kann, einen größeren Engpass als erwartet darstellte. Da mir keine äquivalenten Performance-debugging Tools bekannt waren, und mehrere Mitarbeiter des Fachgebiets bereits händeringend nach solchen gesucht hatten, habe ich über dieses Tool einen weiteren Artikel verfasst [Anhang?], den ich voraussichtlich im August für die PPoPP-2013 Konferenz einreichen werde.

2

Aufgrund der gewonnenen Erkentnisse verändert sich mein Plan wie folgt:

Da die Vermutung, dass ein zentralisiertes System, in dem ein Master-Kern die Aufgabenverteilung für das gesamte System übernimmt, die beste Lösung sein würde, nicht so gesichert wie am Anfang erscheint, werde ich mich stärker den verschiedenen Möglichkeiten der Parallelisierung zuwenden. Dabei wird mir das BeeFarm-multiprozessorsystem [http://www.bscmsrc.eu/sites/default/files/ als Grundlage dienen. Dieses System ist auf MIPS Prozessoren basiert, die eine Coprozessorschnittstelle enthalten, über die zusätzliche Hardwarekomponenten einfach integriert werden können.

Der erste Schritt wird sein, die VMS-basierte StarSs-Laufzeitumgebung auf die BeeFarm-Platform zu portieren. Diese reine Software-Implementierung wird die Vergleichsbasis gegenüber der der Effekt der unterschiedlichen Hardwarebeschleunigerkonfigurationen gemessen werden kann.

Als nächstes muss ein beschleunigender Coprozessor für das StarSs-Plugin erstellt werden. Wie erwartet besteht der Großteil der StarSs-Runtimeaktivität aus dem Nachschlagen in Hashtabellen, jedoch stellte sich auch heraus, dass das Zuteilen von Speicherplatz über *malloc* überraschend viel Zeit in Anspruch nimmt. Dabei gilt es, zwischen vier unterschiedlichen Möglichkeiten zur Beschleunigung abzuwägen und die richtige Kombination zu treffen:

- Spezialisiertere, weniger flexible Hardware ist schneller als programmierbare, multifunktionale Hardware
- Eigenständige Abläufe können getrennt und parallel ausgeführt werden
- Innerhalb dieser Abläufe kann durch Pipelining der Durchsatz gesteigert werden
- Mehrere Instanzen eines Ablaufs können parallel arbeiten.

Im Falle von StarSs gibt es zwei zeitintensive Abläufe mit sehr unterschiedlicher Verteilbarkeit: Die Erstellung von neuen Tasks ist strikt sequentiell definiert, mit nur einem einzigen "Master", der neue Tasks in Auftrag geben darf. Die Zugriffsrechte der Tasks auf Ein- und Ausgangsspeicherzonen werden durch die Reihenfolge, in der sie in Auftrag gegeben wurden, bestimmt. Dieser Ablauf findet zwangsläufig nur in einem Kern statt, und ist der einzige, der neue Einträge in der Hashtabelle erstellen kann, sodass es sinnvoll erscheint, hierfür einen speziellen Coprozessor zu erstellen, der nur zu einem designierten "Master-Kern" hinzugefügt wird. Der zweite Ablauf findet zum Ende jedes Tasks statt. Er gibt die Ein- und Ausgangsspeicherzonen auf die der Task Zugriffsrechte hatte frei und prüft nach, welche wartenden Tasks durch diese Freigaben laufbereit werden. Unter den laufbereiten Tasks muss dann einer ausgewählt und auf dem freigewordenen Kern gestartet werden. Dieser Ablauf ist der rechenintensivste in der StarSs-Laufzeitumgebung, sodass hier ein Hardwarebeschleuniger dringend notwendig ist. Er könnte entweder zentralisiert stattfinden – die "Arbeiter-Kerne" müssten dann den Abschluss eines Tasks an den Master melden und auf die Zuteilung eines neuen Tasks warten – oder dezentral auf dem Kern, der den Task beendet hat. Eine zu starke Zentralisierung führt schnell zu Engpässen, insbesondere für Programme, die viele kurze Tasks enthalten. Auf der anderen Seite wäre es aber sehr ressourcenaufwändig, jedem Kern einen Coprozessor für diesen Ablauf zur Verfügung zu stellen, und dieser wäre unbenutzt während der gesamten Rechenzeit, die im Programm verbracht wird (und dies ist die "nützliche" Zeit, und daher hoffentlich die überwältigende Mehrheit). An dieser Stelle wird es also ein interessanter Punkt sein, zu messen, wie viele Kerne sich einen Coprozessor teilen können, ohne dass es zu größeren Verzögerungen kommt.