Hardwarebeschleunigung von paralleler Ablaufplanung für Multicoresysteme
Zwischenbericht zum 1. Jahr

Nina Engelhardt

22. 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 Virtualzed Master-Slave System, kurz VMS, ein Framework dass es ermöglicht, Laufzeitumgebungen für verschiedenste parallele Programmiermodelle mit nur wenig Aufwand zu implementieren.

1 Ausgeführte Arbeiten

In den ersten 6 Monaten habe ich wie geplant das VMS-System analysiert. VMS umfasst die Grundlegende Struktur, die allen Laufzeitumgebungen (*Runtimes*) gemein ist: sie beinhaltet den Wechsel zwischen Programm- und Umgebungscode und die Möglichkeit, eine zeitliche Ordnung zwischen zwei Codepunkten in zwei virtuellen Prozessoren zu garantieren (semantikfreie Synchronisation). Auf dieser Basis 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. Die von VMS gelieferten grundlegenden Elemente sind sehr nah an der existierenden Hardware, deshalb sind die meisten häufigen Aufgaben bereits 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. Diese Sprachen sind von großer Bedeutung, 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 OmpSs anvisiert werden, da sie im Fachgebiet Architektur Eingebetteter Systeme bereits verwendet wird.

VMS ist die praktische Ausformulierung eines Modells des Parallelismus, die von Dr. Sean Halle unter dem Namen "Holistic Model of Parallel Computation"[2] 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 Problemstellen in der Theorie, und habe diese in Zusammenarbeit mit Dr. Halle gelöst. 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. Wenn man an Out-of-order Prozessoren denkt, wird deutlich, das auch dies parallele Aufgabenverteilung ist. Das Holistische Modell ist auch auf diese anwendbar. Zur Zeit arbeiten wir an einem Zeitschriftenartikel 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 bereits für die Analyse des Laufzeitverhaltens Messschnittstellen in VMS eingefügt hatte, habe ich diese Infrastruktur wiederverwendet, um aus den gewonnenen Daten die räumliche und zeitliche Platzierung der Arbeitseinheiten für den 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), modifiziert, um die Constraints zu erfassen, die für die Konstruktion der UCC und des CG notwendig sind. Schließlich erstellte ich ein 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 vorherigen, statistischen Analysen 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) zur Visualisierung hinzugefügt hatte, wurden auch weniger offensichtliche 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 [1], den ich voraussichtlich im August für die PPoPP-2013 Konferenz einreichen werde.

2 Zukünftige Aktivitäten

Wegen der Arbeit an zwei unvorhergesehenen Veröffentlichungen hat sich die Arbeit an der Hardwareplattform etwas verzögert, sie hat aber mein Verständnis des Problems vertieft und einige neue Blickwinkel eröffnet, die auch in das Design der Plattform einfließen. 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 Verteilung zuwenden. Dabei wird mir das BeeFarm-Multiprozessorsystem[3] als Grundlage dienen, das am Fachgebiet zur Verfügung steht. Dieses System ist auf MIPS Prozessoren basiert, die eine Coprozessorschnittstelle enthalten, über die zusätzliche Hardwarekomponenten einfach integriert werden können.

Der erste Schritt wird sein, VMS und die darauf basierten Laufzeitumgebungen (insbesondere OmpSs) 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 die Plugins erstellt werden. Dabei ist bei der Auswahl der Beschleuniger immer zu berücksichtigen, dass spezialisiertere, weniger flexible Hardware schneller ist als programmierbare, multifunktionale Hardware – jedoch nur seltener anwendbar.

OmpSs hat die komplizierteste Laufzeitaktivität unter den verfügbaren Plugins, daher werde ich hier zuerst ansetzen. Wie erwartet besteht der Großteil der OmpSs-Runtimeaktivität aus dem Nachschlagen in Hashtabellen; diese werden ebenso im SSR-Plugin für die Kommunikation von Nachrichten verwendet. Hashtabellen und ähnliche Assoziativspeicher werden auch anderweitig in vielen Bereichen benutzt, von Routing in Netzwerkprotokollen [4] bis zu Graph-basierten Algorithmen [5]. Da die einfache Implementierbarkeit von VMS-Plugins besonders der Entwicklung von Domain-Spezifischen Sprachen (und den entsprechenden Laufzeitumgebungen) zugutekommt, ist es wahrscheinlich, dass ein Hashtabellenbeschleuniger auch vielen zukünftigen Plugins von Nutzen sein wird.

Zudem stellte sich auch heraus, dass das Allozieren von Speicher über *malloc* relativ viel Zeit in Anspruch nimmt. Da diese Funktion in sehr vielen Plugins enthalten ist, und häufig in der sequentiellen oder minder parallelen Vorbereitungsphase des Programms vor dem Start der eigentlichen parallelen Arbeit benutzt wird, könnte eine Beschleunigung hier einen großen Einfluss haben.

Neben der Auswahl ist auch die Anzahl und Platzierung der Beschleuniger abzuwägen. Im Falle von OmpSs gibt es zwei zeitintensive Abläufe mit sehr unterschiedlicher Verteilbarkeit: Die Erstellung von neuen Tasks ist strikt sequentiell definiert, ausgehend von 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. Aus diesem Grund erscheint es sinnvoll, 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, welche wartenden Tasks durch diese Freigaben laufbereit werden. Unter den laufbereiten Tasks wählt er einen aus und startet ihn auf dem freigewordenen Kern. Hier wäre Hardwarebeschleunigung dringend notwendig, da dieser Ablauf der rechenintensivste in der OmpSs-Laufzeitumgebung 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. Zudem bliebe dieser ungenutzt während der gesamten Rechenzeit, die im Programm verbracht wird (und dies ist die "nützliche" Zeit, und daher idealerweise 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.

Die übrigen Schritte bleiben unverändert. Die OmpSs-Laufzeitumgebung muss anschließend so angepasst werden, dass sie von den neuen Coprozessoren gebrauch machen kann. Anhand mehrerer bereits in OmpSs existierender Programme, darunter Standardbenchmarks und ein H.264 Videoencoder, kann dann die Performance evaluiert werden. Um die Kosten der größeren Flexibilität zu evaluieren, kann des weiteren der OmpSs-spezifische

Hardwarescheduler Nexus++ herangezogen werden, der hier im Fachgebiet entwickelt wurde, und sämtliche im vorigen Absatz beschriebenen Abläufe übernimmt.

Zusammengefasst ergibt sich daraus der folgende Zeitplan:

VMS und OmpSs-Plugin auf BeeFarm portieren	3 Monate
Hardwarebeschleuniger implementieren	6 Monate
Varianten erstellen	3 Monate
Integration in VMS-OmpSs	3 Monate
Test, Performanceevaluation, Optimierung	3 Monate
Niederschrift	6 Monate

Literatur

[1] Nina Engelhardt, Sean Halle, Ben Juurlink; *Integrated Performance Tuning Using Semantic Information Collected by Instrumenting the Language Runtime*; in progress

[2] Sean Halle, Albert Cohen; *Support of Collective Effort Towards Performance Portability*; Proceedings of 3rd USENIX Workshop on Hot Topics in Parallelism, May 2011

[3] Nehir Sonmez, Oriol Arcas, Gokhan Sayilar, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, Satnam Singh, and Mateo Valero; *From plasma to beefarm: design experience of an FPGA-based multicore prototype*; Proceedings of the 7th international conference on Reconfigurable computing: architectures, tools and applications (ARC'11), 2011

[4] Lorenzo De Carli, Yi Pan, Amit Kumar, Cristian Estan, Karthikeyan Sankaralingam; *PLUG: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-speed Routers*; SIGCOMM, August 2009

[5] Betkaoui, B., Thomas, D.B., Luk, W., Przulj, N.; *A framework for FPGA acceleration of large graph problems: Graphlet counting case study*; International Conference on Field-Programmable Technology (FPT), 2011