Graduiertenkolleg 1103

Embedded Microsystems

Albert-Ludwigs-Universität Freiburg

Program Optimisation for Embedded Systems

Statusbericht

Annette Bieniusa

Betreuer: Prof. Dr. Peter Thiemann
Lehrstuhl: Programmiersprachen

Freiburg, im September 2008

Institut für Informatik

Institut für Mikrosystemtechnik

1 Aktueller Stand der Promotion

Derzeit befinde ich mich im zweiten Promotionsjahr. Nach einer Einarbeitungsphase in die Forschung zu parallelem und nebenläufigem Rechnen habe ich mich vertieft mit Software Transactional Memory beschäftigt und bearbeite dieses Thema nun im Kontext von eingebetteten Systemen und Sensornetzwerken.

2 Zusammenfassung der Dissertation

Das Erstellen von Software für eingebettete Systeme ist durch die Begrenzung von Ressourcen (verfügbare Energie, Speicherkapazität, Prozessorleistung, ...) stark eingeschränkt. Gleichzeitig wird von eingebetteten Systemen eine besonders hohe Zuverlässigkeit erwartet. Diese Einschränkungen und Erwartungen zwingen den Softwareentwickler dazu, von typischen Softwareentwicklungsprozessen abzuweichen und Programme spezifisch für die jeweilige Anwendung zu entwicklen. Dieses Vorgehen ist ineffizient und verschlechtert die Wartbarkeit.

Typische Einsatzgebiete von eingebetteten Systemen (Auto- und Maschinenbau, Prozess- und Gebäudeautomation) enthalten mittlerweile meist mehrere miteinander verbundene Einzelkomponenten. Programmiert man Anwendungen für diese Zielsysteme, begibt man sich auf natürliche Art und Weise in das Gebiet des parallelen und verteilten Rechnens. Hierdurch erhöht sich die Komplexität der Anwendung um ein vielfaches. Zusammen mit der Forderung nach Effizienz, Zuverlässigkeit, Wart- und Erweiterbarkeit ergibt sich somit ein wesentlicher Forschungsbedarf.

Im Rahmen meiner Dissertation habe ich bislang zwei Aspekte dazu betrachtet:

- Verwendung von Domain Specific Languages (DSL)
- Nebenläufige Programmierung in verteilten Systemen, speziell Software Transactional Memory in verteilten Sensornetzwerken

Auf diese werde ich im Folgenden detaillierter eingehen.

2.1 DSL für probabilistische Algorithmen auf FPGAs

In technischen Anlagen wird häufig eine statistische Auswertung der Messdaten bereits zeitnah in der Sensorelektronik durchgeführt. Um die Kosten und Fehleranfälligkeit bei der Implementierung dieser Systeme zu senken, ist das Hardware/Software Co-Design eine bewährte Methode. Unter diesem Schlagwort versteht man den integrierten Entwurf von Hardware- und Softwareanteilen, wobei Systemteile, für die eine hohe Veränderlichkeit prognostiziert wird, in Software implementiert werden, während rechenintensive Bestandteile bevorzugt in Hardware realisiert werden.

Diverse Filtertechniken, wie beispielsweise Partikel-Filter, nutzen stochastische Modelle der Systemdynamik, um Aussagen über den Zustand eines System zu treffen (siehe Abbildung 1). Basierend auf gemessenen Sensordaten wird der Zustand des Systems konform zum Modell je nach Situation zu einem zukünftigen, gegenwärtigen oder vergangenen Zeitpunkt prognostiziert, ermittelt oder rekonstruiert. Um diese probabilistischen Berechnungen konzise zu modellieren, wurden spezielle Sprachen (domain specific languages, DSLs) entwickelt, die die Semantik dieser Berechnungen reflektieren. Pfeffer [7] unterstüzt beispielsweise diskrete Wahrscheinlichkeitsverteilungen der Art $dist[p_1 : e_1, \ldots, p_n : e_n]$ mit $\sum_{i=1}^{n} p_i = 1$. In ähnlicher Weise erweitert Thrun [10] C++ um probabilistische Sprachkonzepte. Ramsey und Pfeffer [8] präsentieren einen stochastischen Lambda-Kalkül, dessen denotationale Semantik auf der Wahrscheinlichkeitsmonade beruht.

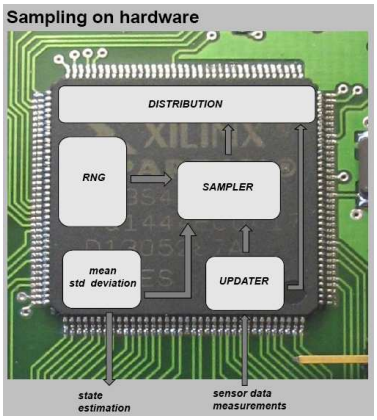

Abbildung 1: Informationsfluss in Partikel-Filtern.

Wir folgen dem Ansatz von Park *et al.* [6], Wahrscheinlichkeitsverteilungen durch entsprechende Sampling-Methoden zu repräsentieren. Der Sampling-Mechanismus gestattet es, aus vorgegebenen Wahrscheinlichkeitsverteilungen Stichproben zu entnehmen und so diverse Verteilungen durch relative Häufigkeitsverteilungen zu approximieren. Der besondere Reiz hierbei besteht in der einheitlichen syntaktischen und semantischen Behandlung sowohl diskreter als auch kontinuierlicher Verteilungen. Insbesondere können mit diesem Ansatz auch nichtlineare Systemmodelle behandelt werden, ohne weitere vereinfachende Annahmen und Approximationen anwenden zu müssen.

Wir haben den Prototyp einer funktionalen DSL hierfür entwickelt. RINSO [2, 1] ist eine einfach getypte funktionale Sprache, basierend auf dem Lambda-Kalkül, mit syntaktischer Unterstützung für monadische Sprachkonstrukte. Neben einer formalen Definition der Syntax, Semantik und des Typsystems haben wir eine Transformation der monadischen Ausdrücke nach CPS (continuation passing style) entwickelt.

In einem nächsten Schritt wird die Einbindung des Sampling-Mechanismus und weiterer wahrscheinlichkeitstheoretischer Konstrukte erfolgen. Diese so entstehende Probability-Monade enthält als essentiellen Bestandteil einen performanten Pseudo-Zufallszahlengenerator. Des Weiteren arbeiten wir an einem Compiler, der eine effiziente Kompilierung von RINSO-Programmen nach VHDL durchführt. Hierbei soll die Möglichkeit, Berechnungen parallel ausführen zu können, optimal genutzt werden. Der so erzeugte VHDL-Code kann dann für Simulationen und die Implementierung in FPGAs verwendet werden.

In dem hier vorgestellten Anwendungsbereich lässt sich Parallelität auf Datenebene lokal zu Optimierungszwecken nutzen. Um auch global, d.h. auf Netzwerkebene, vorhandene Ressourcen möglichst erschöpfend zu nutzen, müssen Methoden des nebenläufigen Progammierens eingesetzt werden.

2.2 Nebenläufige Programmierung in verteilten Systemen

Um auch Programme für eingebettete Systeme portabel und wiederverwendbar zu gestalten, wird in den letzten Jahren immer häufiger versucht, virtuelle Maschinen als Abstraktionsschicht zur konkreten Hardware einzusetzen. Diese erlauben durch ihre komplexeren Befehlssätze eine kompaktere Darstellung von Programmen, so dass die zumeist knappe Ressource Speicherplatz effektiver genutzt werden kann. Darüber hinaus können bei Bedarf zeitkritische Berechnungen durch

so genannte Just-in-Time Compiler effektiv ausgeführt werden. Die Java Virtual Machine (JVM) ist ein Beispiel für eine (auch kommerziell) erfolgreiche Umsetzung dieser Idee, die mittlerweile auch auf ressourcenschwachen Plattformen eingesetzt wird.

Um Speicherkonsistenz bei nebenläufigen Anwendungen zu erreichen, wurde bislang entweder auf Mechanismen zum wechselseitigen Ausschluss durch Locking oder explizite Kommunikation (message passing) seitens des Programmierers zurückgegriffen. Diese Ansätze erhöhen jedoch die Komplexität und Fehleranfälligkeit bei der Erstellung von Anwendungen, und skalieren nur bedingt mit der Größe des Netzwerks.

In Anlehnung an Dan Grossmans Analogie von Garbage Collection und Software Transactional Memory (STM) [4] haben wir einen extensiven Vergleich verschiedener Methoden zur Programmierung von multi-threaded Anwendungen und der Softwareentwicklung mit Hilfe von Versionskontrollsystemen ausgeführt. Neben klaren Ähnlichkeiten in der Grobstruktur eröffnet die Gegenüberstellung im Detail einige interessante Beobachtungen und Ideen, wie sich die beiden Bereiche gegenseitig befruchten können.

Einige Aspekte hiervon sind bereits in einen STM-Mechanismus eingeflossen, der in verteilten Sensornetzwerken zum Einsatz kommen soll.

2.3 Software Transactional Memory in verteilten Sensornetzwerken

Software Transactional Memory [9] ist ein modernes Programmierparadigma, das in den letzten Jahren in den Fokus der Forschung gerückt ist. Kritische Codeabschnitte in Threads, in denen auf Bereiche des Shared Memory zugegriffen wird, werden als spezielle Einheiten, Transaktionen, gekennzeichnet. Dieser Code wird isoliert und atomar zu anderen nebenläufigen Berechnungen ausgeführt. Die Laufzeitumgebung übernimmt dabei das Speichermanagement und koordiniert die Lese- und Schreibzugriffe der Transaktionen. Durch das optimistische Ausführen der atomaren Einheiten wird ein höherer Grad an Parallelität erreicht, der allerdings durch ein eventuelles Rollback im Falle eines nicht auflösbaren Konflikts und einen Overhead in der Speicherverwaltung durch die Laufzeitumgebung zu bezahlen ist. Diese Idee wird schon seit einigen Jahren erfolgreich im Datenbankenbereich eingesetzt, der sie ursprünglich entnommen wurde.

Eine Integration von STM in eine Programmiersprache muss eine Vielzahl von Entwurfskriterien betrachten, die in Abhängigkeit zu der betrachteten Programmiersprache und den typischen Anwendungsszenarien stehen. Der Prozess der Bewertung dieser Entwurfskriterien ist Gegenstand aktueller Forschung, ebenso wie die Portierung von existierenden Anwendungen auf diese neue Systemarchitektur.

Es gibt zwar mittlerweile einige Ansätze, STM-Sprachkonstrukte in Java zu integrieren und entsprechende JVMs zu konstruieren. Mit DSTM2 [5] stellen die Sun Microsystems Labs ein flexibles Framework zur Verfügung, mit dem diverse Ausprägungen von STM für Java getestet werden können. Dieses zielt darauf ab, die Nützlichkeit und Umsetzbarkeit von STM-Entwürfen rasch in einer Prototyp-Implementierung zu testen. Es ist allerdings ungeeignet, um performante und größere Applikationen in verteilten Szenarien zu erstellen. Carlstrom *et al.* [3] greifen für ihre JVM-Implementierung auf spezialisierte Hardware-Unterstützung zurück. Dieser Ansatz lässt sich allerdings nicht auf andere Prozessortypen übertragen und steht damit in Kontrast zur Plattformunabhängigkeit, wie sie von Java eigentlich angestrebt ist.

Für den Einsatz in verteilten Systemen wie Sensornetzwerken eignen sich die bisherigen Implementierungen nicht, da sie zumeist in Form von Bibliotheken zur Verfügung stehen. STM erfordert jedoch von der Konzeption grundlegende Änderungen im Speichermanagement und daher eine angepasste JVM, um performante, zuverlässige und sichere Programme auszuführen.

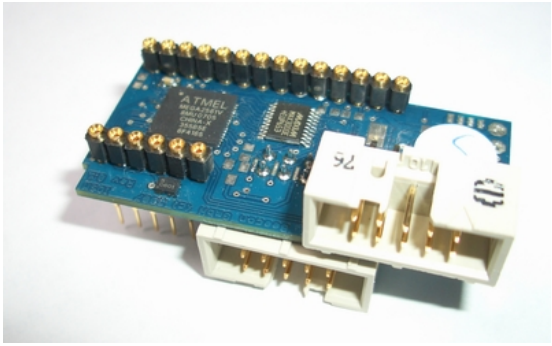

Abbildung 2: Eine Zielplattform von Ambicomp, das Beecon IOSM, basiert auf dem ATMega2561 mit Steckerleisten für das EthernetSM und Sensoren.

In Kooperation mit der Forschungsgruppe um Professor Dr. Thomas Fuhrmann (TU München/TU Karlsruhe) arbeiten wir an einer verteilten virtuellen Maschine für eingebettete Systeme. Die Ambicomp Virtual Machine (ACVM) verarbeitet Java Bytecode, der durch einen Transkodierschritt an den Befehlssatz der ACVM angepasst wurde. Hierbei wird der Code analysiert, gelinkt und signiert in einem BLOB (binary large object) zum Aufspielen auf die Hardware abgelegt. Ausserdem optimiert der Transcoder die Anwendung für die jeweilige Hardware und bindet die passenden Bibliotheken ein. Eine Zielplattform der ACVM sind AVR 8-bit Mikrokontroller, wie sie typischerweise in Sensornetzwerken verwendet werden (siehe Abbildung 2). Spezielle System-Bibliotheken unterstützen die Speicherverwaltung, Kontrolle der IO-Schnittstellen, z.B. die Netzwerkschnittstellen zur Kommunikation, und stellen die Funktionalität des Betriebssystems zur Verfügung, das dadurch obsolet wird.

Speicherbereiche mit Objekten, die von mehreren Threads gelesen und geschrieben werden, werden in Ambicomp in einem globalen gemeinsam genutzten Heap realisiert, der über alle Knoten des Netzwerkes verteilt ist. Ein sogenannter Orakelmechanismus organisiert den Zugriff auf die GAOs (globally accessible objects) transparent für den Anwender. Domains regeln die Zugriffsrechte, so dass lediglich Knoten, die auch in einer Anwendung partizipieren, auf zugehörige GAOs Zugriff erhalten.

Derzeit spezifizieren und implementieren wir einen STM-Mechanismus für Ambicomp, der speziell den Ansprüchen einer ressourcenschwachen Umgebung gerecht wird. Die Speicherverwaltung des globalen Heaps und speziell die Auflösung von Konflikten bei Objektzugriffen erfolgt dabei transparent für den Programmierer durch die Laufzeitumgebung.

Darüber hinaus entwickeln wir Analysen, die aufgrund einer statischen Detektion von möglichen Konflikten bei Speicherzugriffen durch Adaption des Schedulings einen niedrige Konfliktrate erzielen. Weitere Forschungsbereiche umfassen Algorithmen zur Verwaltung von Redundanzinstanzen von Objekten, um Ausfallsicherheit durch Checkpointing zu errreichen. Des Weiteren arbeiten wir an einer Integration von STM Konzepten in Java, mit denen typische Anwendungsfälle möglichst einfach und fehlerfrei durch den Programmierer implementiert werden können. Dazu zählen Typsysteme oder andere Annotationen, aber auch die Eingliederung von bisherigen Sprachelementen zur nebenläufigen Programmierung.

Durch den Paradigmenwechsel von Single- zu Multicore- und Manycoreprozessoren sind diese Ansätze auch für den Einsatz in PCs oder Rechenclustern interessant. In einem weiteren Projekt „JCell“, das im Rahmen des BmBF-Förderprogramms „IKT 2020: Forschung für Innovationen“ beantragt wurde, soll daher die Portierung des Systems auf die Architektur des Cell-Processors (IBM) evaluiert werden.

Literatur

[1] A. Bieniusa and P. Thiemann. “How to CPS transform a monad”. In *Tagungsband zu Trends in Functional Programming (TFP’ 08)*. Gennep, Netherlands, 2008, pp. 270–284.

[2] A. Bieniusa. “RINSO goes random”. In *Tagungsband zum 14. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS’ 07)*. Timmendorfer Strand Niendorf, Germany, 2007, pp. 6–11.

[3] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. “Executing java programs with transactional memory”. *Science of Computer Programming*, vol. 63, pp. 111–129, 2006.

[4] D. Grossman. “The transactional memory / garbage collection analogy”. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*. Montreal, QC, Canada, 2007, pp. 695–706.

[5] M. Herlihy, V. Luchangco, and M. Moir. “A flexible framework for implementing software transactional memory”. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*. Portland, OR, USA, 2006, pp. 253–262.

[6] S. Park, F. Pfenning, and S. Thrun. “A probabilistic language based upon sampling functions”. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*. Long Beach, CA, USA, 2005, pp. 171–182.

[7] A. Pfeffer. “IBAL: A probabilistic rational programming language”. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*. Seattle, WA, USA, 2001, pp. 733–740.

[8] N. Ramsey and A. Pfeffer. “Stochastic lambda calculus and monads of probability distributions”. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL ’02)*. 2002, pp. 154–165.

[9] N. Shavit and D. Touitou. “Software transactional memory”. In *Proceedings of the 14th ACM Symposium on Principles of distributed computing (PODC ’95)*. New York, NY, USA, 1995, pp. 204–213.

[10] S. Thrun. “Towards programming tools for robots that integrate probabilistic computation and learning”. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA 2000)*. San Francisco, CA, USA, 2000, pp. 306–312.