DISS. ETH Nr. 18077

Online Optimizations Using Hardware Performance Monitors

ABHANDLUNG
zur Erlangung des Titels

DOKTOR DER WISSENSCHAFTEN

der

ETH ZÜRICH

vorgelegt von

Florian Thomas Schneider
Dipl. Ing. ETH

geboren am 31. Juli 1979

von Österreich

Angenommen auf Antrag von
Prof. Dr. Thomas R. Gross, Referent
Prof. Dr. Timothy Roscoe, Korreferent
Dr. Vijay S. Menon, Korreferent
Prof. Dr. Matthias Hauswirth, Korreferent

2009

Abstract

Feedback-directed optimization is becoming more and more important in modern execution environments. On one hand modern CPUs impose additional difficulties for the compiler to generate efficient code: Features like multi-core and complex cache architectures make things even harder for the compiler.

On the other hand, modern programming languages are often not well suited for classic compiler optimizations (due to dynamic class loading, polymorphism, etc.). JIT compilation with feedback-guided optimization can provide significant benefits for such programs since the compiler/runtime is able to adapt to the running program and does not require complex static analysis like in an ahead-of-time compiler. Programming languages like Java or C# make intensive use of references and generally deal with a large number of small objects. As a result, data access pattern are often irregular and difficult (if not impossible) to analyze statically.

This thesis focuses on how to gather feedback provided from the hardware platform via hardware performance monitors (HPM) and use it for program optimizations in a managed runtime for Java programs. We present HPM information as another source of feedback – in addition to traditional profile-guided optimization. We work in a dynamic compilation environment and all optimizations are done as the program runs. When doing online optimizations the runtime overhead of performance monitoring is of special importance.

We look at what kind of information is available and useful for optimizations. We discuss the different problems and challenges that come with collecting HPM information. The information has to be precise and unbiased. Different hardware platforms offer different ways of obtaining HPM information. We look at two platforms in more detail: the P4 (IA-32) and the Itanium Montecito (IPF) processor

Since the HPM data collection runs concurrently with the application it should have a very low execution time overhead. Our solutions adaptively adjusts the sampling period for event sampling to limit the amount of data that has to be processed by the JVM. We present a solution to efficiently collect HPM information in an online setting with a consistently small overhead of less than 1% on average.

The raw HPM information has to be mapped back to the source program to be useful for performance analysis or compiler optimizations. We extended an existing JVM to generate the meta-information necessary to perform this mapping similar to a symbolic debugger. Detailed instruction-level information is required to achieve a reliable mapping.

We present different applications of fine-grained (instruction-level) HPM information:

We perform a detailed performance analysis of Java applications. HPM information can be used to identify performance-critical parts of a program at the instruction-level. Our system is also able to measure data address profiles that allow to map HPM samples about cache misses back to data structures on the heap. We show how to use data address profiles to compare the performance behavior of different generational garbage collection algorithms.

As an example of how to use HPM feedback for optimizations, we present two examples: We show how HPM information can be used to perform object co-allocation – an online optimization that improves data locality at GC time in a Java VM. In the best case the execution time is reduced by 16% (3% on average) and L1 cache misses by 28%.

We also demonstrate how HPM feedback can be used to improve loop unrolling in a high-performance static compiler. We perform per-application and per-loop adaptation of loop unrolling parameters using data about stall cycles from the PMU. In the best case we achieve a 39% speedup (6% average) over the default heuristic at the highest optimization level.

Zusammenfassung

Feedback-gesteuerte Optimierungen werden in modernen Laufzeitumgebungen immer wichtiger. Auf der einen Seite haben moderne Prozessoren immer mehr Funktionen, die es einem Compiler schwer gestalten effizienten Code zu generieren. Beispiele dafür sind Multi-Core-Prozessoren und immer komplexere Speicherhierarchien.

Auf der anderen Seite haben moderne dynamische Progammiersprachen Eigenschaften, die fuer klassische Compiler-Optimierungen nicht gut geeignet sind (dynamisches Class-Loading, Polymorphismus, etc.). JIT Compiler haben hier den Vorteil, da sie durch Feeback zur Laufzeit mehr Informationen über ein laufendenes Programm verwenden können und so adaptiv optimieren können ohne komplexe statische Analyse zu benötigen. Programmiersprachen wie Java oder C# verwenden viele Referenzen und eine grosse Zahl von kleinen Objekten. Die Speicherzugriffsmuster sind daher oft unregelmässig und schwierig (wenn nicht unmöglich) statisch zu analysieren.

Diese Dissertation konzentriert sich darauf, wie Informationen, die vom Prozesso durch Hardware-Performance-Monitore (HPM) zur Verfügung gestellt werden, effizient gesammelt und in einer Java Laufzeitumgebung zur Programmoptimierung genutzt werden können. Informationen von den HPM ist eine weitere Quelle von Feedback neben traditioneller Profiling-Information. Unser System ist in eine Java VM mit JIT compiler eingebettet, wobei alle Optimierungen während der Laufzeit ausgeführt werden. Deshalb ist es extrem wichtig, die Feedback-Daten möglichst effizient zu sammeln.

Zuerst betrachten wir welche Art von Informationen zur Verfügung stehen und welche für Optimierungen benutzt werden können. Um HPM-Informationen zu bekommen müssen verschiedene Probleme gelöst werden: Die Informationen müssen genau und unverzerrt sein. Je nach Hardware-Architektur gibt es verschiedene Möglichkeiten das zu erreichen. Diese Arbeit stellt zwei Architekturen genauer vor und beschreibt wie HPM-Informationen genutzt werden könnnen: den P4 Prozessor und den Itanium Montecito-Prozessor.

Da die HPM-Datensammlung und -verarbeitung parallel zur Programmausführung passieren, müssen die zusätzlichen Kosten minimal gehalten werden. Unsere Lösung verwendet ein adaptives Sampling-Intervall um die Datenmenge zu begrenzen und zuverlässig einen geringen Overhead zu erreichen ($< 1\%$ im Durchschnitt).

Dann müssen die gesammelten rohen HPM-Daten zurück mit dem Source-Programm verbunden werden, um sie in der Java VM weiterzuverwenden. Wir erweitern eine existierende JVM, indem wir Meta-Informationen über jede Applikation im JIT Compiler

speichern, ähnlich wie ein symbolischer Debugger. Um die genaue Instruktion im Source-Programm zu finden ist es notwendig, dass die HPM-Daten auf eine Maschineninstruktion genau sind.

Als nächstes stellen wir verschiedene konkrete Anwendungen der erstellten Infrastruktur vor: Zuerst führen wir eine detailierte Performance-Analyse einiger Java-Programme durch. Die HPM-Daten helfen dabei performance-kritische Load-Instruktionen zu finden.

Mit unserem System können wir auch HPM-Daten mit Addressen im Speicher assozieren (d.h. Speicheraddress-Profile zu erstellen). Damit ist es möglich Datenstrukturen eines Programms zu finden, die für die Gesamt-Performance entscheidend sind. In unseren Experimenten verwenden wir Speicheraddress-Profile um die Cache-Performance verschiedener Garbage-Collector-Algorithmen im Detail zu vergleichen.

Zum Schluss zeigen wir zwei Optimierungen um zu zeigen, wie HPM-Informationen als Feedback verwendet werden kann. Die erste Optimierung ist Objekt-Koallokation, eine Online-Optimierung für bessere Cache-Performance beim Speicherzugriff auf Java-Objekte. Im besten Fall erreichen wir damit eine Beschleunigung um bis zu 16% (3% im Durchschnitt) und reduzieren die Anzahl der Cache-Misses um bis zu 28%.

Die zweite Optimierung ist Loop-Unrolling. Hier demonstrieren wir wie HPM-Informationen die existierenden Heuristiken für Loop-Unrolling in einem statischen Compiler verbessern kann. Im besten Fall resultiert eine Verbesserung um bis zu 39% (6% im Durchschnitt) verglichen mit der Standard-Heuristik auf dem hoechsten Optimierungs-Level.