Einführung

Als Profiler werden Programmierwerkzeuge bezeichnet, die das Laufzeitverhalten von Software analysieren. Es gibt unterschiedliche Problembereiche in der Softwareentwicklung, die durch ineffiziente Programmierung ausgelöst werden. Ein Profiler hilft dem Entwickler durch Analyse und Vergleich von laufenden Programmen die Problembereiche aufzudecken. Daraus kann man Maßnahmen zur strukturellen und algorithmischen Verbesserung des Quellcodes ableiten.

In diesem Artikel werden drei verschiedene Profiler für die Programmiersprache C und C++ vorgestellt. Zu Demonstrationszwecken bietet sich das Programm für den Advanced Encryption Standard (AES)
an, da sich der Algorithmus besonders gut für eine Analyse des Laufzeitverhaltens eignet.

Bereiche des Profilings

Man unterscheidet beim Profiling prinzipiell drei verschiedene Bereiche. Der wohl wichtigste Bereich ist die Laufzeitanalyse, d.h. die Erkennung von sogenannten "Hot Spots". Dabei handelt es sich um Codeabschnitte, die besonders viel Rechenzeit vereinnahmen. Daneben spielen aber auch die Speichernutzung und die Nebenläufigkeit eine Rolle.

Messen von Geschwindigkeit

Die häufigste Anwendung eines Profilers ist das Zählen und Messen der Aufrufe und Durchläufe von Funktionen bzw. Methoden. Dies ermöglicht es dem Programmierer herauszufinden, wo sich eine **Optimierung** des Programms lohnt. Eine Optimierung von Funktionen, die nicht häufig verwendet werden, ist der Gesamtleistung des Programms nicht sonderlich zuträglich und erschwert in der Regel die Wartbarkeit des Quellcodes. Deshalb wird das Hauptaugenmerk auf Funktionen gelegt, die oft aufgerufen werden und in der Summe der Aufrufe viel Zeit benötigen.

Speichernutzung

Ein weiterer Aspekt ist die Verfolgung von Speichernutzung durch ein Programm. Der Profiler soll dabei helfen, den Ge- und Verbrauch von Arbeitsspeicher zu optimieren und ggf. Fehler in

der Programmierung aufzudecken, durch die ungenutzte Speicherbereiche nicht freigeben werden. Daraus resultiert ein Speicherleck .

Nebenläufigkeit

Moderne Profiler bieten auch die Möglichkeit, nebenläufige Prozesse (Threads) in ihrem Lebenszyklus grafisch (zum Beispiel als Balken- oder Netzdiagramm) darzustellen. Diese optische Aufbereitung soll einem Programmierer helfen, das Laufzeitverhalten von nebenläufigen Prozessen besser zu interpretieren und Fehler durch Verklemmung (Deadlock) aufzudecken.

Technische Aspekte des Profilings

Das Profiling eines Programms selbst verursacht in der Regel eine Beeinflussung der zu analysierenden Anwendung. Üblicherweise verlangsamt der Profiler selbst die Ausführungsgeschwindigkeit. Außerdem entsteht bei Analysen von großen Programmen eine sehr große Menge an Daten. Es gibt unterschiedliche Techniken beim Profiling, die eine solche Beeinflussung, ggf. unter Verlust der Analysegenauigkeit, verschieden stark ausprägen. Außerdem ermöglichen die Profiler es zu bestimmen, welche Programmteile überhaupt analysiert werden sollen.

Das Profiling einer nativen Anwendung gestaltet sich oftmals schwierig, da ein kompiliertes Programm im Endzustand stark optimiert vorliegt. In der Regel ist dafür der Compiler zuständig, der den resultierenden Maschinencode auf Geschwindigkeit und/oder Größe hin optimiert. Dabei verliert der Code sowohl seine Symbolnamen (engl. name mangling) als auch seine ursprüngliche Struktur. Profiler, welche sich nicht in eine Entwicklungsumgebung integrieren, verlieren auf diese Weise die Möglichkeit "Hot Spots" dem Quellcode zuzuordnen. Auch eine Auflistung der ursprünglichen Funktionsnamen aus dem Quellcode ist nicht mehr möglich, so dass das Profiling erschwert wird. Abhilfe schaffen hier die sogenannten Debug Builds. Diese enthalten noch alle Symbolinformationen, alle Assertions, sind nicht optimiert und die Runtime-Checks sind noch aktiv. Da Profiler und Debugger ähnliche Daten verwenden und oftmals zusammen in einem Werkzeug ausgeliefert werden, erläutern wir zunächst den Debugger.

Debugger

Wenn Sie einen Debugger verwenden, erwarten Sie mit diesem schrittweise durch den Quellcode schreiten zu können, Haltepunkte im Code setzen zu können, die Werte von diversen Variablen einzusehen, inklusive derer von komplexen benutzerdefinierten Datentypen. Eine typisches ausführbares Programm (engl. executable) besteht allerdings nur aus eine Sequenz roher Bytes, hauptsächlich Maschineninstruktionen mit Datensegmenten und

spezifischen Metadaten. Wenn die ausführbare Datei vom Betriebssystem geladen und ausgeführt wird, verwendet das System zusätzlich verschiedene Speicherarten, z.B. den Stack, Heap, etc. Auch hierbei handelt es sich um Rohdaten. Ein Debugger kann nicht wissen, welche Zeile des Quellcodes zu der gerade ausgeführten Maschineninstruktion gehört. Auch ist nicht bekannt, welche Adresse im Stack zu der lokalen Variable korrespondiert. Die Lösung für diese Probleme bieten die Debuginformationen, das Bindeglied zwischen der abstrakten Programmiersprache und den Rohdaten in der ausführbaren Anwendung.

Arten von Debuginformationen

Lassen Sie uns eine Zusammenfassung der Informationen anfertigen, die von einem Debugger und Profiler gebraucht werden, um ihre Arbeit durchführen zu können. Wir beschränken uns in diesem Artikel auf die Debugger von Microsoft auf einer Intel x86 Plattform. Die nachfolgende Tabelle listet alle Arten von Debuginformationen auf, die derzeit verwendet werden.

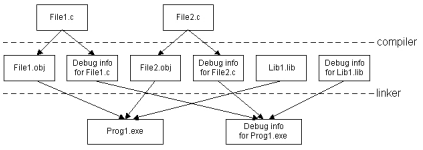
Arten von Information	Beschreibung
Öffentliche Funktionen und Variablen	Diese Art von Information beschreibt die Funktionen und Variablen, die
Private Funktionen und Variablen	Diese Art von Information beschreibt alle Funktionen und Variablen, in
Quellcode- und Zeileninformationen	Diese Art von Information bildet jede Zeile in jeder Quellcodedatei auf
Typinformation	Für jede Funktion oder Variable, kann die Debuginformation zusätzlich
FPO Information	Bei einer Funktion, die mit FPO (engl. frame pointer omission) kompilie
Bearbeitungs- und Fortsetzungsinformation	Diese Art von Information hilft Visual Studio dabei die Bearbeitungs- u

Diese Art von Information beschreibt lokale Variablen, die Stelle und den Namen und Variablen, die

Build Prozess

Der Bauprozess einer typischen Anwendung vollzieht sich in zwei Schritten - Kompilieren und Linken. Zuerst analysiert der Parser im Compiler die Quellcodedateien und generiert die Maschineninstruktionen, die in den Objektdateien gespeichert werden. Jede Quellcodedatei wird in eine Objektdatei kompiliert. Anschließend bindet der Linker alle verfügbaren Objektdateien zu einer einzigen ausführbaren Datei. Der Linker kann angewiesen werden zusätzliche Bibliotheken einzubinden, die in der Regel eine Kollektion von Objektdateien darstellen.

Wenn Sie Debuginformationen für eine ausführbare Datei produzieren wollen, müssen ebenfalls zwei Schritte durchlaufen werden. An dieser Stelle werden allerdings zunächst für jede Quellcodedatei Debuginformationen generiert. Zum Schluß werden auch hier wieder alle Objektdateien vom Linker gebunden. Dieser Vorgang wird in der nachfolgenden Abbildung demonstriert.



Standardmäßig produzieren Compiler und Linker keine Debuginformationen. Daher lassen sich in jedem Schritt entsprechende Optionen spezifizieren, um den Compiler anzuweisen ganz bestimmte Debuginformationen zu erstellen. Es lässt sich exakt definieren welche Art von Debuginformationen produziert werden sollen, welches Debugformat verwendet werden soll und wo diese Debuginformationen gespeichert werden sollen.

Compiler

Der Compiler kann mit den forgenden Optionen angewiesen werden Debuginformationen für jede Quellcodedatei zu erstellen: /Zd, /Z7, /Zi, /ZI (alle Optionen sind auch in der IDE konfigurierbar).
Die /Zd Option veranlasst den Compiler Debuginformationen im COFF Format zu produzieren und in der resultierenden Objektdatei zu speichern.
Die /Z7 Option veranlasst den Compiler Debuginformationen im CodeView Format zu produzieren und in der resultierenden Objektdatei zu speichern.
Die /Zi Option veranlasst den Compiler Debuginformationen im Program Database Format zu produzieren und in einer separaten .PDB zu speichern.
Die /ZI Option ist fast identisch zu /Zi, aber die resultierenden Debuginformationen beinhalten zusätzlich Daten für die Bearbeitungs- und Fortsetzungsfunktion.
Der Name einer .PDB Datei, die von /Zi und /ZI verwendet wird, ist VCX.PDB. Das kann mit der Compileroption /Fd geändert werden

Die zusammengefassten Informationen werden in der Tabelle dargestellt.

Option	Format	Datei	Inhalt
/Zd	COFF	.OBJ Datei	- Öffentliche Funktionen und Variablen - Quellcode- und Zeileninformationen - FPO Information
/Z7	CodeView	.OBJ Datei	- Öffentliche Funktionen und Variablen - Private Funktionen und Variablen - Quellcode- und Zeileninformationen - Typinformation - FPO Information
/Zi	Program Database	.PDB Datei	

- Öffentliche Funktionen und Variablen
- Private Funktionen und Variablen
- Quellcode- und Zeileninformationen
- Typinformation
- FPO Information

/ZI	Program Database	.PDB Datei

- Öffentliche Funktionen und Variablen
- Private Funktionen und Variablen
- Quellcode- und Zeileninformationen
- Typinformation
- FPO Information
- Bearbeitungs- und Fortsetzungsdaten

Profiler

Auf dem Markt befinden sich zahlreiche Analyse- und Profilingwerkzeuge, kostenfreie und kostenpflichtige. Interpretierte Umgebungen, wie Java oder .NET, bieten meist von Haus aus Werkzeuge zur Laufzeitanalyse. So gibt es zahlreiche kostenlose .NET und Java Profiler. Die IDE NetBeans stellt kostenlos einen sehr leistungsfähigen Profiler für Java Anwendungen zur Verfügung.

Auf dem Gebiet der nativen Anwendungen ist das Angebot deutlich eingeschränkter. Der Intel ® VTune Performance Analyzer
ist ein kommerzieller Profiler für die Softwareanalyse auf x86 und x64 basierten Plattformen. Es handelt sich bei VTune um einen sehr leistungsfähigen Profiler mit grafischer Benutzeroberfläche und einer Vielzahl von Funktionen. Der Profiler ist für Linux und Windows verfügbar. Die Version für Linux ist kostenfrei, für Windows wird eine zeitlich befristete Probeversion angeboten.

In dem nachfolgenden Abschnitt sollen drei kostenfreie Profiler für C und C++ Anwendungen erläutert werden. Darunter findet sich ein einfacher Profiler für gelegentliche Analysen von kleinen Programmen, bis hin zu einem ausgefeilten Profiler für große Anwendungen und einem umfangreichen Bedarf an Daten.

Shiny

Shiny ist ein leichtgewichtiger, schneller und vollständig dokumentierter High-Performance

Profiler für C/C++/Lua. Er lässt sich sehr leicht verwenden, ohne größere Eingriffe am eigenen Code oder Projekt vornehmen zu müssen. Die Resultate werden in einem Aufrufbaum dargestellt und nach der Zeit sortiert. Die Ergebnisse können bei Bedarf auch in Ogre3D gerendert und in jeder beliebigen Grafikengine verwendet werden.

- Sehr wenig Overhead, sehr leistungsfähig und akkurat bei der Laufzeitanalyse.
- Unglaublich einfach in der Handhabung und gute Dokumentation des Quellcodes.
- Resultate werden in einem Aufrufbaum dargestellt und nach der Laufzeit sortiert.
- „On-the-fly“ Glättung der Mittelwerte für wissenschaftliche und Main-Loop orientierte Programme, wie z.B. Spiele.
- Gerenderte Graphen und farbig dargestellte Ausgaben in Ogre3D und anderen Grafikengines.
- Plattformunabhängiger, objektorientierter und frei verwendbarer Quellcode.

Shiny ist Open Source und kann auf Sourceforge unter der Adresse http://sourceforge.net/projects/shinyprofiler/
heruntergeladen werden.

Die Verwendung von Shiny ist einfach. Der Profiler besteht nur aus zwei Bibliotheken, deren Prototypen über Header in das eigene Projekt eingebunden werden. Anschließend lassen sich die Analysefunktionen im Quellcode aufrufen. Jede Funktion, die analysiert werden soll, beginnt mit dem Makro PROFILE_FUNC(). Die Resultate können am Ende mit PROFILER_OUTPUT() zusammengefasst ausgegeben werden, nachdem sie mit PROFILER_UPDATE() ausgewertet wurden.

[code xml:lang="cpp"]#include "Shiny.h" #ifdef _WIN32 #include #else // assume POSIX #include #endif #include using namespace std; void millisleep(unsigned int milliseconds) { #ifdef _WIN32 Sleep(milliseconds); #else usleep(milliseconds * 1000); #endif } void Recursion(int calls_left) { PROFILE_FUNC(); // begin profile until end of block if (calls_left > 0) Recursion(calls_left - 1); } void MSleep() { PROFILE_FUNC(); // begin profile until end of block millisleep(100); } int main() { Recursion(12); MSleep(); PROFILER_UPDATE(); // update all profiles PROFILER_OUTPUT(); // print to cout return 0; }[/code]

In dem dargestellten Beispiel werden die beiden Funktionen Recursion() und MSleep() analysiert. Shiny produziert daraufhin das folgende Ergebnis.

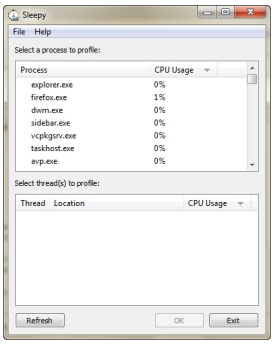
[thumb src="images/tutorials/cpp/shiny.png" arg="200;;;Im Konsolenfenster zeigt Shiny die Funktionen an. Sowohl die Anzahl der Aufrufe, als auch die verbrauchte Zeit werden aufgelistet."]Shiny Profiler[/thumb]

Ein offensichtlicher Nachteil von Shiny ist der Umstand das der Profiler einen Eingriff in den Quellcode erfordert, d.h. eine manuelle Instrumentierung durch Erweiterung des Quelltextes durch Befehle stattfindet. Allerdings lassen sich die Daten auf diese Weise auch direkt programmtechnisch auswerten und Funktionen können ganz selektiv analysiert werden.

Very Sleepy

Very Sleepy ist ein freier Profiler, der auf dem Projekt Sleepy basiert. Der C/C++ CPU Profiler für Windows Betriebssysteme kann als Installer erworben werden und ist dank statistischer Auswertung vollständig unabhängig vom zu testenden Programm nutzbar. Bei der statistischen Auswertung wird die Programmanalyse nicht exakt mit jedem Programmbefehl einer Messung unterzogen. Es wird vielmehr in einem bestimmten zeitlichen Zyklus die Laufzeit gemessen. Dieses Verfahren nennt man auch **Sampling**. Der Profiler ist in der Lage auch Programme mit mehreren parallel laufenden Threads zu analysieren. Very Sleepy bezieht Stichproben aus dem EIP Register, um zu messen in welchem Codeabschnitt das Programm gerade seine Zeit verbringt. EIP hält den Befehlszeiger in Intel Architekturen und beherbergt die lineare Adresse der nächsten zu ausführenden Instruktion innerhalb des Codesgements. Der Profiler ist sehr schnell und liefert sehr zuverlässig elementare Laufzeitdaten.

Das Programm ist ist Open Source und kann von der Seite http://www.codersnotes.com/sleepy bezogen werden.



Das Profiling von Algorithmen ist eine wesentliche Komponente bei der Laufzeitanalyse. Kryptographische Algorithmen, wie der Rijndael-Algorithmus, der die Grundlage für den Advanced Encryption Standard bildet, sind besonders interessant für diese Art von Analyse. Very Sleepy ist einfach zu bedienen und kann zu einem beliebigen Zeitpunkt gestartet werden, um Stichproben von einem Programm anzufertigen.

Starten Sie dazu den Profiler und wählen Sie aus der Liste der laufenden Prozesse das zu untersuchende Programm. Anschließend werden im unteren Fenster die zu diesem Programm

gehörenden Threads angezeigt. Sie können nun auf den gewünschten Thread doppelklicken. Very Sleepy beginnt damit in festen Intervallen Stichproben von dem zu analysierenden Objekt zu sammeln. Dieser Vorgang kann zu einem beliebigen Zeitpunkt abgebrochen werden. Nach dem Abbruch fertigt der Profiler die Statistiken an. Zu diesen Statistiken zählt neben den Funktionsaufrufen auch die visuelle Anzeige der "Hot Spots" mit den korrespondierenden Zeilen im Quellcode, sofern die zu untersuchende ausführbare Datei Debuginformationen mit einem Verweis auf die Quellcodedatei enthält.

[thumb src="images/tutorials/cpp/Very Sleepy Profiling.png" arg="200;;;In der Operation MixColumns, bei der mit der Funktion mulGaloisField2_8 im endlichen Körper multipliziert wird, verbringt das analysierte Programm die meiste Zeit."]Very Sleepy Profiling[/thumb]

Very Sleepy zeigt einen vollständigen Überblick über alle aufgerufenen Funktionen, angeführt von der Funktion, die am meisten Rechenzeit auf der CPU beansprucht hat. Im unteren Dialogfeld wird auch die betreffende Stelle farbig im Quellcode angezeigt, mit den prozentualen Anteilen. Im rechten Dialogfeld ist der Aufrufbaum abgebildet, an dem erkennbar ist, welche Funktion die betreffende Funktion aufgerufen hat.

Mit dem Profiler Very Sleepy können Sie Laufzeitanalysen abspeichern und jederzeit mit anderen Stichproben vergleichen. Die Wirkung einer Optimierung lässt sich auf diese Weise einfach überprüfen.

Der Profiler Very Sleepy eignet sich hervorragend zur unkomplizierten Laufzeitanalyse von Programmen auf Windows. Very Sleepy beschränkt sich auf wesentliche Daten und gestattet es sehr schnell Problembereiche in Anwendungen aufzudecken.

AMD CodeAnalyst Performance Analyzer

Der „AMD CodeAnalyst Performance Analyzer“ ist ein kostenfreier und sehr leistungsfähiger Profiler vom US-amerikanischen Chiphersteller Advanced Micro Devices, Inc. (AMD). Der Profiler wurde speziell für Prozessoren von AMD entworfen, kann aber auch mit einigen kleinen Einschränkungen auf Systemen mit Intel-Prozessoren verwendet werden. CodeAnalyst integriert sich nahtlos mit der Entwicklungsumgebung Microsoft Visual Studio 2008 und 2010. Das Programm steht sowohl für Microsoft Windows, als auch für das Linux Betriebssystem zur Verfügung.

- **Systemweites Profiling:** CodeAnalyst wurde entworfen, um die Performance von Binärmodulen zu analysieren, inklusive User-Mode Applikationsmodulen und Kernel-Mode Treibermodulen. Timer-basiertes Profiling, Ereignis-basiertes Profiling und auf Instruktionen-basiertes Sampling sammeln Daten von allen aktiven Prozessoren in einem

Multiprozessorsystem.

- **Timer-basiertes Profiling (TBP)**:

1. Die zu optimierende Applikation läuft auf höchster Geschwindigkeit auf einem System, auf dem auch CodeAnalyst läuft. Es werden Stichproben vom EIP in festdefinierten Intervallen gesammelt, um wahrscheinliche Engpässe oder Optimierungsmöglichkeiten zu identifizieren.

2. Die feinste Auflösung beträgt 0.1 ms.

3. **Ereignis-basiertes Profiling (EBP):** CodeAnalyst EBP wurde entworfen, um Hardwareereignisse auf dem AMD Athlon™, AMD Athlon™ XP, AMD Opteron™, AMD Athlon™ 64, AMD Family 10h und AMD Family 11h zu analysieren. Mit der Ereignis-Multiplexing Technik, ist CodeAnalyst EBP in der Lage bis zu 32 verschiedene Hardwareereignisse, während einer einzigen Stichprobe zu messen.
4. **Instruktionen-basiertes Sampling (IBS):** Instruktionen-basiertes Sampling ist eine neue Messtechnik für die Performance einer Anwendung, unterstützt von allen AMD Barcelona (Family 10h) Prozessoren. IBS hat die folgenden Vorteile:

1. IBS verbindet präzise Hardwareereignisse mit den Instruktionen, die die Ereignisse ausgelöst haben. Ein fehlgeschlagener Zugriff auf einen Datencache wird beispielsweise mit der AMD64 Instruktion verknüpft, die für den gescheiterten Speicherschreib- und lesezugriff verantwortlich war.
2. IBS sammelt eine große Bandbreite an Hardwareereignissen in einem einzigen Messzyklus.
3. IBS sammelt neue Informationen, wie die Latenzzeit für fehlgeschlagene Speicherzugriffe.

4. **Call Stack Sampling (CSS):** Durch die Kombination von TBP, EBP oder IBS kann das Call Stack Sampling Daten an Hot Spots über die Aufrufer-Gerufener-Beziehung sammeln.

5. **Thread Profiling (nur Windows Versionen):** CodeAnalyst's Thread Profiling Ansichten zeigen ein Threaddiagramm und nicht-lokale Speicherzugriffe.

6. **Post Process:** CodeAnalyst zeigt die Verteilung von Stichproben ohne Debuginformationen an.
1. Die Laufzeitdaten werden interpretiert und nicht einfach als Rohdaten angezeigt.
2. Flexible Konfiguration der Ansicht und der Verwaltung.
3. (Nur Linux Versionen) Stichproben können separat für Inline Funktionen oder für die Elternfunktionen gesammelt werden.

CodeAnalyst bietet kostenfrei ein sehr breites Spektrum an Funktionen an, sogar ohne

Debuginformationen. Leider steht die gesamte Bandbreite nur auf Systemen mit AMD-Prozessoren zur Verfügung. Dennoch lassen sich auch auf Intel-basierten Systemen Anwendungen analysieren.

Der Profiler von AMD integriert sich nahtlos mit der Entwicklungsumgebung Microsoft Visual Studio. In dem nachfolgenden Bild sehen Sie einen Ausschnitt des Profilings mit CodeAnalyst in Visual Studio 2010.

[thumb src="images/tutorials/cpp/CodeAnalyst Profiling.png" arg="200;;;CodeAnalyst integriert sich direkt in Visual Studio und ermöglicht es Anwendungen aus der Entwicklungsumgebung zu starten und zu analysieren."]CodeAnalyst Profiler[/thumb]

CodeAnalyst ist in der Lage neben dem Quellcode in Hochsprache, auch den vom Compiler erzeugten Maschinencode (Assembler) anzuzeigen.

[thumb src="images/tutorials/cpp/CodeAnalyst Profiling 2.png" arg="200;;;CodeAnalyst zeigt neben den bekannten Daten über kritische Funktionen auch den Code in Assembler an."]CodeAnalyst Profiler[/thumb]

Der AMD CodeAnalyst Performance Analyzer kann von der Seite http://developer.amd.com/CPU/CODEANALYST/Pages/default.aspx
bezogen werden.

In diesem Artikel haben Sie drei kostenlose und relativ unterschiedliche Profiler kennengelernt, mit denen sich native C/C++ Anwendungen analysieren lassen. Ziel einer Programmanalyse ist ein effizientes Laufzeitverhalten der Anwendung. Damit ein Vergleich auch objektiv gelingt, ermöglichen viele Profiler, wie Very Sleepy und CodeAnalyst die gesammelten Analysedaten zu speichern und zu vergleichen. Somit können Programmänderungen in ihrem Laufzeitverhalten verglichen und bewertet werden.