10 Debugger und Profiler

Die Fehlersuche beansprucht einen großen Teil der Zeit während der Entwicklung einer Applikation. In einer Multithread-Anwendung kann die Fehlersuche besonders schwierig sein, da es hier eine Gruppe von Fehlern gibt, die sonst keine Rolle spielen. Diese Synchronisationsfehler können meist nur indirekt, etwa durch das Auftreten eines Deadlocks, festgestellt werden. Die Untersuchung eines solchen Fehlers wird zusätzlich dadurch erschwert, dass der Debugger den Testlauf einer Anwendung in Bezug auf das Zeitverhalten der einzelnen Threads massiv beeinflussen kann. Der Abschnitt 10.1 *Der Debugger und die Untersuchung von Synchronisationsfehlern* gibt eine kurze Definition des Synchronisationsfehlers an, der Abschnitt 10.2 *Der Einfluss des Debuggers* beschäftigt sich mit dem Einfluss des Debuggers auf das Verhalten einer Anwendung und der Abschnitt 10.3 *Die zusätzlichen Möglichkeiten des Debuggers* mit den zusätzlichen Mitteln, die der Debugger zur Untersuchung einer Multithread-Anwendung bereitstellt. Der Abschnitt 10.4 *Deadlock Detection* geht auf das generelle Problem des Erkennens von Deadlocks ein und der Abschnitt 10.5 *Der Einsatz mehrerer Compiler* schließlich erwähnt ein Problem, das durch den Einsatz mehrerer Compiler auftreten kann.

Im zweiten Teil beschäftigen sich die Abschnitte 10.6 *Der Profiler* und 10.7 *Grundsätzliches zum Einsatz des Profilers* mit dem Thema der Performance-Messung. Diese sollte in jedem Fall mit Hilfe eines Profilers durchgeführt werden. Zunächst wird erläutert, was ein Profiler ist und wie er insbesondere mit einer Multithread-Anwendung eingesetzt wird. Der Abschnitt 10.8 *Drei Beispiele für den Einsatz des Profilers* schließlich zeigt ganz praktisch an einem Beispiel, wie der Profiler verwendet wird.

10.1 Der Debugger und die Untersuchung von Synchronisationsfehlern

Es gibt kaum einen Bereich, bei dem der Unterschied zwischen einer Singlethread- und einer Multithread-Anwendung deutlicher zu Tage tritt als bei der Fehlersuche. Das beginnt damit, dass bestimmte Fehler gar nicht auftreten, sobald das Programm im Debugger läuft. Andere treten viel seltener auf und wieder andere nur im Zusammenhang mit einem Debugger. Bei einer Multithread-Anwendung kommt es viel häufiger zu einem solchen Verhalten als bei einer Singlethread-Anwendung. Damit es kein Missverständnis gibt: Ich meine den identischen Code und nicht den Unterschied im Verhalten der Debug- zur Release-Version.

Praktisch alle Fehler, die auf Grund einer mangelhaften Synchronisation entstehen, sind vom Zeitverhalten der einzelnen Threads abhängig. Mit solchen *Synchronisationsfehlern* meine ich alle Fehler,

- die zu einem Deadlock führen können.
- die zum inkonsistenten Schreiben in ein Objekt führen können.
- die zum inkonsistenten Lesen aus einem Objekt führen können.

Die Ursache dieser Fehler kann darin bestehen, dass die Synchronisation eines Objektes mit der eines anderen kollidiert. Möglich ist auch, dass ein Objekt unzureichend oder gar nicht synchronisiert oder unzulässig angewendet wird. Ein Synchronisationsfehler kann auch aus einer unzureichenden Fehlerbehandlung resultieren. Diese Gruppe von Fehlern tritt nur in Multithread-Anwendungen auf.

Synchronisationsfehler werden fast immer nur an ihren Folgen erkannt: Ein Thread »hängt«, ein Datensatz wird nicht geschrieben. Der Code ist syntaktisch korrekt und verhält sich in den meisten Fällen auch wie gewünscht. Das Problem tritt immer nur auf, wenn bestimmte Umstände zusammentreffen. Dazu gehört immer eine bestimmte zeitliche Reihenfolge, in der mehrere Threads Aktionen ausführen.

Zunächst gelten für eine Debug-Session mit einer parallelen Anwendung die gleichen Regeln wie für eine Singlethread-Anwendung. Ein paar dieser Regeln, die gerne vergessen werden, führe ich hier noch einmal auf:

- Die Release-Version sollte immer mit Debug-Informationen gebaut werden.
- Die Release-Version sollte frühzeitig getestet werden.
- Bei Problemen sollte testweise die Optimierung der Release-Version verändert werden.

Das Debuggen der Release-Version ist zwar etwas schwieriger als das der Debug-Version, aber genauso möglich und wichtig. Lassen Sie sich nicht davon abschrecken, dass Sie auf Grund der Optimierung durch den Compiler manche Variablen nicht einsehen können und dass die Reihenfolge, in der der Debugger den Code abarbeitet, manchmal überraschende Wendungen nimmt. Hier kann der letzte der genannten Punkte hilfreich sein.

Für die Untersuchung einer Multithread-Anwendung bietet der Debugger besondere Möglichkeiten an, aber er beeinflusst die Anwendung auch. Auf beide Aspekte gehe ich in den nächsten Abschnitten ein.

10.2 Der Einfluss des Debuggers

Es ist nie möglich, zwei Testläufe unter exakt gleichen Randbedingungen durchzuführen. Bei einer Singlethread-Anwendung ist das fast immer egal, bei einer Multithread-Anwendung fast nie. Der Ablauf einer Multithread-Applikation reagiert auf alle Änderungen der CPU-Auslastung durch andere Prozesse. Eine Singlethread-Anwendung ist dagegen weitgehend immun. Selbst wenn es gelingt, die Randbedingungen so konstant zu halten, dass ein Testlauf unter ziemlich ähnlichen Bedingungen wiederholt werden kann, so sorgt der Einsatz eines Debuggers für eine massive Störung. Er führt nämlich zu einer impliziten und zufälligen Synchronisation der Threads und er verändert das Zeitverhalten der einzelnen Threads relativ zueinander.

Wenn Sie eine Anwendung debuggen, müssen Sie das zum Teil drastisch veränderte Zeitverhalten der einzelnen Threads berücksichtigen, wenn es um Synchronisationsprobleme geht. Der Einfluss auf den Programmablauf kann in gewissem Rahmen dadurch begrenzt werden, dass an den kritischen Stellen auf ein zeilenweises Debuggen verzichtet wird und Breakpoints sparsam verwendet werden.

10.2.1 Breakpoints und zeilenweises Debuggen

Immer dann, wenn der Debugger auf einen Breakpoint trifft, muss er den entsprechenden Thread aussetzen. Ebenso muss er alle anderen Threads unterbrechen, bis die Ausführung des Programms fortgesetzt

wird. Dadurch greift der Debugger in die Reihenfolge ein, in der der Code auf den unterschiedlichen Threads ausgeführt wird. Das »Timing« wird verändert. Selbst wenn dieser Eingriff geringfügig ist und nicht gleich zu einem Versatz um Tausende von Anweisungen führt, so kann er doch genügen, um das Reproduzieren eines Synchronisationsfehlers zu erschweren oder ganz unmöglich zu machen.

Wenn Sie nicht nur einzelne Breakpoints setzen, sondern zeilenweise durch den Code debuggen (Single Stepping), ist der Einfluss des Debuggers noch gravierender. In diesem Fall kann es zu erheblichen Verschiebungen zwischen der Ausführung des aktiven, also gerade debuggten Threads und den anderen Threads der Anwendung kommen. Intern realisiert der Debugger das Single Stepping dadurch, dass er auf die jeweils nächste Codezeile einen temporären Breakpoint setzt. Während eine einzelne Anweisung des aktiven Threads ausgeführt wird, erhalten auch die anderen Threads je ein Quantum der CPU. Während des Quantums können aber sehr viele Instruktionen abgearbeitet werden, so dass die anderen Threads scheinbar wesentlich schneller arbeiten als der gerade aktive. Das wird auch nicht dadurch kompensiert, dass ihnen nicht bei jeder Anweisung des aktiven Threads eine Zeitscheibe zugewiesen wird.

Es ist leider nicht möglich, die Größe dieser Verschiebung zwischen den Threads zu quantifizieren. Sie können jedoch davon ausgehen, dass sie ganz erheblich sein kann und umso größer ist, je mehr einzelne Anweisungen Sie auf diese Weise und auf dem gleichen Thread debuggen.

Bestimmte Anweisungen wie etwa Synchronisationspunkte beeinflussen die Verschiebung und werden von ihr beeinflusst. So ist es wahrscheinlich, dass der Thread, den Sie gerade schrittweise debuggen, als letzter einen Synchronisationspunkt erreicht. Wenn beispielsweise zwei Threads mit gleicher Wahrscheinlichkeit einen Mutex akquirieren können, so ändert sich die Wahrscheinlichkeitsverteilung drastisch, wenn einer von beiden vor Erreichen des Synchronisationspunktes schrittweise debuggt wird. Dieser Thread wird so gut wie immer deutlich später die entscheidende Stelle des Codes erreichen und somit erheblich verminderte Chancen haben, als erster den Mutex zu besitzen. Wenn aber genau das die Voraussetzung ist, um ein bestimmtes Szenario zu reproduzieren, verhindert der Debugger den Erfolg.

Es ist also wichtig, beim Versuch, einen Synchronisationsfehler zu reproduzieren, den Unterschied zwischen dem zeilenweisen Debuggen eines Threads und der Verwendung von Breakpoints zu beachten. Durch den sparsamen Einsatz von Breakpoints wird das Timing erheblich weniger beeinflusst als durch das manuelle Debuggen des Threads. Daher sollte es nicht verwundern, dass manche Probleme nur reproduziert werden können, wenn Breakpoints an bestimmten Stellen eingesetzt werden – oder eben gerade nicht verwendet werden. Umgekehrt treten manche Probleme erst dadurch auf, dass bestimmte Breakpoints gesetzt werden.

10.2.2 Das Ausgabe-Fenster

Aber es kommt auch dann zu einer impliziten Synchronisation, wenn die Anwendung einfach nur im Debugger läuft und gar keine Breakpoints gesetzt sind oder die gesetzten nicht angesprungen werden. Denn immer dann, wenn ein Thread in das Ausgabe-Fenster des Debuggers schreibt, wird diese Ausgabe intern synchronisiert. Egal welche Funktion verwendet wird, um in das Ausgabe-Fenster zu schreiben, letztlich wird immer die Funktion `OutputDebugString` aufgerufen. Dabei spielt es keine Rolle, ob sie direkt aufgerufen wird oder indirekt über die MFC, die ATL oder eine andere Bibliothek. Sie alle greifen auf diese Funktion zurück. Intern verwendet `OutputDebugString` einen Mutex, um die Ausgabe zu synchronisieren. Allerdings nur dann, wenn auf die Anwendung tatsächlich ein Debugger angewendet wird. Anderenfalls tut die Funktion gar nichts.

Wenn also ein Debugger mitläuft, und nur dann, führt der parallele Zugriff auf die Funktion `OutputDebugString` dazu, dass die beteiligten Threads synchronisiert werden. Das hat zwar den Vorteil, dass die Ausgabe schön geordnet erscheint, aber es verändert das Zeitverhalten der Threads erheblich. Daher kann es hilfreich sein, einige Ausgaben vorübergehend auszukommentieren, wenn der Verdacht besteht, dass sie den Programmablauf während des Debuggens stören.

Dieses Verhalten unterscheidet sich deutlich von der Ausgabe in das Konsolen-Fenster. Hier wird die Ausgabe nicht synchronisiert. Daher kann es vorkommen, dass Wörter und Sätze wild gemischt werden. Aber dafür wird das Zeitverhalten der Threads nicht beeinflusst.

10.2.3 Zeitabhängige Anweisungen

Alle Anweisungen, die in irgendeiner Weise zeitabhängig sind, können eine veränderte Bedeutung für den Programmablauf erhalten, wenn die Anwendung im Debugger läuft. Insbesondere dann, wenn zeilenweise debuggt wird. Denn auch wenn ein Thread vorübergehend unterbrochen wird (Break), läuft die Zeit weiter. Am offensichtlichsten ist das bei den `Sleep`-Funktionen. Ein `Sleep` von 100 Millisekunden suspendiert einen Thread für mehrere Zeitscheiben, die ihm potenziell zugewiesen werden können. Wenn aber manuell debuggt wird, sind 100 Millisekunden weniger, als gewöhnlich für das Abarbeiten von einer Zeile zur nächsten benötigt wird. Die Zeitspanne bekommt also eine ganz andere Bedeutung. Konkret heißt das, dass ein Thread effektiv viel weniger lange wartet, wenn ein anderer Thread gerade manuell debuggt wird.

Das gilt natürlich nicht nur für die `Sleep`-Funktion, sondern für alle, die von irgendeiner vorgegebenen Zeitspanne abhängig sind. Eine Funktion, die eine Datenbankabfrage durchführt, bringt eine relativ kürzere Verzögerung mit sich, wenn die Anwendung gerade zeilenweise debuggt wird.

10.3 Die zusätzlichen Möglichkeiten des Debuggers

Der Debugger bietet einige Fähigkeiten an, die nur oder in besonderem Maße für eine Multithread-Anwendung von Bedeutung sind. Das sind zwar manchmal nur Kleinigkeiten, aber ihre Kenntnis kann das Leben erheblich vereinfachen.

Die folgende Beschreibung gilt – streng genommen – nur für das Microsoft Visual Studio der Versionen .net 2002 und .net 2003. Wenn Sie eine andere Entwicklungsumgebung verwenden, wird das meiste zwar auch so ähnlich sein, aber eben doch etwas anders aussehen. Möglicherweise werden einige Eigenschaften auch gar nicht unterstützt.

10.3.1 Der Debugger überwacht alle Threads

Der Debugger überwacht jederzeit alle Threads. Der aktuelle Thread ist derjenige, der das letzte Break des Debuggers verursacht hat und dessen Code angezeigt wird. Alle anderen Threads laufen natürlich mit. Der Debugger sorgt dafür, dass alle Threads mehr oder weniger gleichmäßig abgearbeitet werden. Während Sie also schrittweise und langsam durch den Code eines Threads debuggen, laufen die anderen Threads nicht mit voller Geschwindigkeit weiter, sondern werden ebenfalls immer wieder unterbrochen. Dadurch stellt der Debugger sicher, dass entweder alle Threads suspendiert sind oder keiner. Dabei ist es egal, mit welcher Geschwindigkeit durch den Code debuggt wird. Trotzdem wird das Zeitverhalten der Threads beeinflusst, wie in Abschnitt 10.2 *Der Einfluss des Debuggers* erläutert wurde.

Wenn Sie also einen Thread debuggen und diesen vorübergehend stoppen, so können Sie sicher sein, dass alle anderen Threads ebenfalls temporär gestoppt sind. Die komplette Anwendung ist für den Moment eingefroren und kann beliebig untersucht werden. Dabei ist es auch egal, ob Sie das »Break« manuell gesetzt haben oder ob irgendein Thread auf einen Breakpoint getroffen ist.

Was der Debugger allerdings nicht überwacht, ist die Zugehörigkeit eines Fensters zu einem Thread. Um das zu ermitteln, müssen Sie das zum Developer Studio gehörende Programm Spy++ einsetzen. Mit dessen Hilfe können Sie sehr leicht die IDs des zum Fenster gehörenden Threads und Prozesses in Erfahrung bringen.

10.3.2 Der Wechsel zwischen den Threads

Sie können jederzeit bestimmen, welchen Thread Sie aktiv debuggen möchten. Dazu sollten Sie das Developer Studio so konfigurieren, dass es Ihnen eine Liste (Abbildung 10.1) oder ein Fenster (Abbildung 10.2) mit allen existierenden Threads anbietet. Und ebenso sollte es Ihnen eine Liste oder ein Fenster mit dem Call Stack des aktiven Threads zur Verfügung stellen. Dadurch erhalten Sie jederzeit einen Überblick über alle Threads der Anwendung. Indem Sie einen der Threads in der Liste oder im Fenster auswählen, springt der Debugger an die Position, die aktuell ausgeführt wird, und bringt die angezeigten, threadspezifischen Informationen, wie etwa den Call Stack, auf den entsprechenden Stand. Das Ganze funktioniert aber nur, wenn die Anwendung gerade steht, also nach einem »Break«.

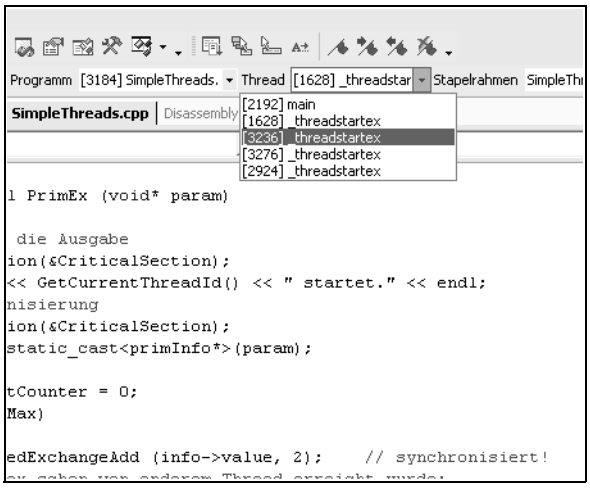

Abbildung 10.1: Die Thread-Liste im Microsoft Developer Studio

Der Debugger führt automatisch einen Wechsel des Fokus durch, wenn er auf einen Breakpoint trifft. Der von diesem Breakpoint betroffene Thread erhält den Fokus und die threadspezifischen Informationen der Fenster werden entsprechend aktualisiert. Wenn viele Breakpoints verwendet werden, ist es daher wichtig, den Überblick zu behalten, zu welchem Thread die angezeigten Informationen überhaupt gehören. Beachten Sie auch, dass selbst das zeilenweise Debuggen zu einem Wechsel des Fokus führen kann. Entweder, weil der Debugger auf einem anderen Thread auf einen Breakpoint trifft oder aber weil die gleiche Funktion parallel auf einem anderen Thread bearbeitet wird. Beim Single Stepping setzt der Debugger ja einen temporären Breakpoint auf die nächste Codezeile. Wenn diese Zeile als Erstes auf einem anderen Thread erreicht wird, so wechselt der Fokus.

Manchmal kann so ein automatischer Fokus-Wechsel auch störend sein, wenn etwa ein bestimmter Thread manuell debuggt werden soll und dann zwischendurch Wechsel stattfinden. Über die Thread-Liste oder das Thread-Fenster können Sie einfach zum gewünschten Thread zurückwechseln. Machen Sie sich in jedem Fall darauf gefasst, dass Sie nicht unbedingt den Thread debuggen, den Sie zu debuggen glauben.

Weder die Thread-Liste noch das Thread-Fenster zeigen an, welcher Thread der Primary Thread ist. Dies können Sie aber über den Call Stack feststellen. Wechseln Sie zu den verschiedenen Threads und werfen Sie einen Blick auf den Call Stack. Der Thread, dessen Call Stack die Hauptfunktion (`WinMain` etc.) enthält, ist der Primary Thread.

Die gleiche Technik können Sie auch verwenden, um zu sehen, welcher Thread gerade auf ein Synchronisationsobjekt wartet. Wenn der Call Stack eines Threads eine Funktion wie `WaitForSingleObject` enthält, so können Sie daraus schließen, dass und warum sich ein Thread im Warte-Zustand befindet.

10.3.3 Das Suspendieren und Aktivieren von Threads

Das Zeitverhalten eines Threads kann auch manuell beeinflusst werden. Es ist möglich, einen Thread temporär zu suspendieren. Etwa um zu verhindern, dass ein Thread demjenigen, der gerade untersucht wird, »davoneilt«. Das Ganze ist auch nützlich, um zu verhindern, dass ein Thread vor einem anderen eine Ressource wie etwa einen Mutex akquiriert.

Im Thread-Fenster (Abbildung 10.2) können Sie über das Kontext-Menü beliebige Threads über den Befehl »Sperren« (»Freeze«) suspendieren und über »Entsperren« (»Thaw«) wieder aktivieren. Nebenbei zeigt das Fenster noch die Priorität und den Suspend-Zähler. Dieser wird ja über `SuspendThread` inkrementiert und über `ResumeThread` dekrementiert. Wenn Sie den Thread manuell suspendieren, hat das keinen Einfluss auf diesen Zähler.

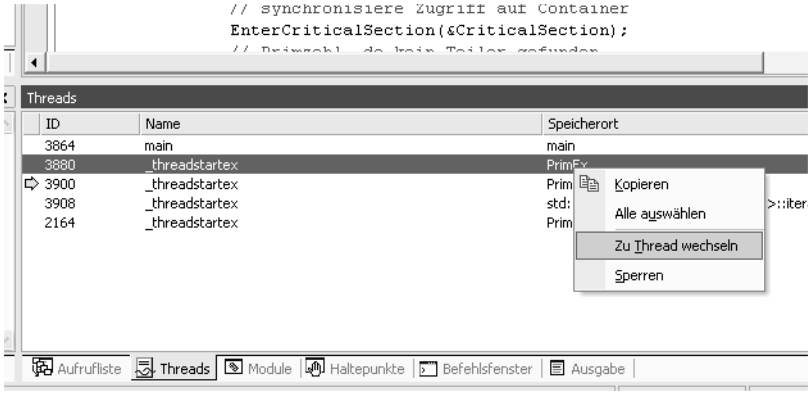

Abbildung 10.2: Das Thread-Fenster im Microsoft Developer Studio

Beachten Sie, dass der Thread, solange er suspendiert ist, keine Nachrichten mehr empfangen kann – falls es sich um einen GUI-Thread handelt.

10.4 Deadlock Detection

Auch bei sorgfältiger Planung und Entwicklung können Sie das Auftreten eines Deadlocks nicht mit Sicherheit ausschließen. Wenn es dann in der Testphase oder im Produktivbetrieb zu einem Deadlock kommt, so besteht die erste Schwierigkeit darin, diesen auch als solchen zu erkennen. Wenn die Anwendung komplett einfriert, liegt der Schluss nahe, aber so einfach ist es selten. Viel wahrscheinlicher ist es, dass beispielsweise eine Anwendung, die ihre Daten in einer Datenbank abspeichert, hin und wieder einen Datensatz nicht sichert, weil der zuständige Worker-Thread »hängt«. Wenn für jeden Speichervorgang eines Datensatzes ein eigener Thread verwendet wird, fällt das Ganze nicht so schnell auf. Irgendwann viel später werden die betroffenen Daten vermisst oder jemand stellt fest, dass sich einzelne Daten nicht auf dem aktuellen Stand befinden. Dann ist es jedoch fast ausgeschlossen, den entsprechenden Thread zu identifizieren oder gar die Ursachen aufzuklären. Daher ist es auch unter dem Aspekt der Fehlersuche gut, ein potenzielles Deadlock in eine Fehlermeldung umzuwandeln. Ich verwende hierzu die C++/C#-Ausnahmen, aber Sie können auch auf einen anderen Fehlerbehandlungsmechanismus zurückgreifen, wenn Sie keine Ausnahmen einsetzen wollen.

In Abschnitt 9.4 *Strategien zum Umgang mit unbehandelten oder kritischen Ausnahmen* habe ich empfohlen, alle Warte-Funktionen ausschließlich mit einem endlichen Timeout-Wert zu verwenden. Das ermöglicht die

Anwendung eines Retry-Mechanismus und somit letztlich die Vermeidung eines echten Deadlocks. Vor allem aber erleichtert es ungemein das Erkennen eines potenziellen Deadlocks.

Wenn der Rückgabewert der Warte-Funktion entsprechend behandelt wird, kann der Fehler protokolliert werden. Welche Daten nützlich sind, hängt von der Funktion ab, aber mit Sicherheit sollte die Position im Quellcode und die Thread-ID gemeldet werden. Im Extremfall kann das bis zu einem kompletten Dump des Stacks gehen.

Wenn ein solches Deadlock erkannt wurde, ist es trotzdem meist nicht leicht zu reproduzieren. Daher ist es hilfreich, bereits während des Programmlaufs, der zum Deadlock führt, möglichst viele Daten zu sammeln. Insbesondere über all die Threads, die am Deadlock beteiligt sind. Eine Möglichkeit besteht darin, den Aufruf aller Synchronisationsfunktionen einschließlich der übergebenen Parameter zu protokollieren. John Robbins beschreibt in seinem hervorragenden Buch [Rob00] sehr detailliert, wie eine DLL entwickelt werden kann, die genau diese Aufgabe erfüllt. Das resultierende Log-File erleichtert die Ursachenforschung erheblich. Das Schreiben einer solchen DLL hat eigentlich nichts mit Multithreading zu tun. Sie kann zum Protokollieren jeglichen Funktionsaufrufs eingesetzt werden. Wenn Sie also wissen möchten, was ein Funktions-Hook ist und wie Sie selber so etwas anwenden können, dann lesen Sie bei John Robbins nach.

Es gibt zahlreiche andere Vorschläge, wie ein Deadlock erkannt werden kann. Leider sind das immer Lösungen, die sich nur auf einen Teil der Synchronisationsobjekte oder nur unter bestimmten Umständen anwenden lassen. Die weitreichendste Lösung ist tatsächlich ein generisches Logging über Hook-Funktionen, wie von John Robbins vorgeschlagen. Allerdings bedeutet die Entwicklung und Pflege eines solchen Tools einen nicht unerheblichen Aufwand. Daher kann es ratsam sein, ein entsprechendes Tool zu kaufen, statt es selber zu entwickeln. In Kombination mit angemessenen Timeout-Werten lassen sich die meisten Deadlocks gut erkennen und interpretieren.

Eine enorme Menge an Informationen liefert ein professionelles Tool wie AppSight von Identify Software (`www.identify.com`). Er protokolliert alle gewünschten Funktionsaufrufe, bereitet sie grafisch auf und kann so ein sehr umfangreiches Bild liefern. Wenn Sie also nicht den Ehrgeiz entwickeln, unbedingt eine eigene Log-DLL zu schreiben, empfehle ich Ihnen einen Blick auf AppSight.

Auch beim Einsatz all dieser Mittel kann es im Einzelfall immer noch sehr schwierig sein, ein Deadlock und seine Ursachen aufzuklären. Zwar können die Warte-Funktionen mit einem Timeout-Wert versehen werden, nicht aber eine Critical Section der Win32 API. Daher ist ein Deadlock auf Grund einer Critical Section wesentlich schwieriger zu erkennen. Wenn es Indizien gibt, die dafür sprechen, dass sich ein Thread in einem Deadlock befindet, ohne dass es zu einem Timeout kam, kann immer noch versucht werden, den Debugger auf den bereits laufenden – oder eher hängenden – Prozess anzuwenden. Auf diese Weise kann zumindest die Stelle des Deadlocks isoliert werden, wenn die Funktionsaufrufe nicht protokolliert wurden.

Es gibt leider keine absolut zuverlässige Methode, um alle Arten von Deadlocks zu erkennen und die benötigten Informationen zu sammeln. Allerdings ist es schon sehr hilfreich, so oft wie möglich Timeout-Werte zu setzen und den Programmablauf zu protokollieren. Letzteres sollte während der Entwicklungs- und Testphase regelmäßig geschehen. Im produktiven Einsatz kann die Methode angewendet werden, wenn es konkrete Hinweise auf Deadlocks gibt.

10.4.1 Unit Testing

In den letzten Jahren wurde sehr viel zur Anwendung von Testverfahren geschrieben. Insbesondere das so genannte *Unit Testing* wird sehr häufig empfohlen. Dahinter verbirgt sich der in anderen Ingenieurbereichen schon lange weit verbreitete Gedanke, dass es einfacher ist, einen Fehler zu finden, wenn nur eine einzelne Komponente getestet wird, statt ein ganzes System. Die Testergebnisse werden mit Sollwerten verglichen, was häufig mit Hilfe von Tools geschieht. Für die Mehrzahl der Fälle ist dieses Vorgehen sicherlich auch richtig. Nicht jedoch im Hinblick auf Synchronisationsfehler in einer Multithread-Anwendung.

Das Problem besteht darin, dass es eben nicht ausreicht, ein erwartetes Testergebnis reproduzieren zu können. Denn das heißt noch lange nicht, dass keine Synchronisationsfehler auftreten können. Der umgekehrte Schluss bleibt natürlich richtig: Wenn im Unit-Test ein Synchronisationsfehler bemerkt wird, ist er einfacher zu identifizieren, als wenn ein komplexes System untersucht wird.

Daher sollte dem Unit Testing im Hinblick auf Synchronisationsfehler keine übermäßige Bedeutung geschenkt werden. Wesentlich sinnvoller kann das Durchführen eines Code-Reviews sein, wenn dabei ganz besonders auf verschachtelte Locks geachtet wird. Aber das funktioniert auch nur dann, wenn beim Schreiben des Codes Wert auf eine möglichst große Klarheit gelegt wurde.

10.5 Der Einsatz mehrerer Compiler

Grundsätzlich ist es eine gute Praxis, bei der Entwicklung einer Komponente mehrere Compiler einzusetzen. Einerseits wird so eine maximale Kompatibilität des Codes gewährleistet und andererseits kann ein schneller Compiler für den häufigen Bau der Debug-Version des Codes und ein hoch optimierender Compiler für die Release-Version verwendet werden.

Unterschiedliche Compiler erzeugen jedoch unterschiedlichen Code. Daher divergiert nicht nur der resultierende Binärcode, auch die Ergebnisse der beiden Programme können dann unterschiedlich sein, wenn der Quellcode im Sinne des C++-Standards nicht definierte oder nicht eindeutige Anweisungen enthält (das gilt prinzipiell auch für C#, allerdings gibt es für C# bisher erst einen Compiler). Natürlich kann auch ein Compiler-Fehler zu unterschiedlichem Code führen, aber das ist erfahrungsgemäß die Ausnahme.

Das gerade beschriebene Verhalten gilt sowohl für Multithread- als auch Singlethread-Anwendungen. Aber es gilt in besonderem Maße, wenn Sie einen MP-optimierenden Compiler neben dem Standard-Compiler einsetzen. Also wenn Sie etwa den Microsoft- oder Borland-Compiler während der Entwicklung und des Debuggings verwenden und erst danach den Intel-Compiler mit MP-Optimierung einsetzen. Grundsätzlich ist das durchaus sinnvoll, allerdings müssen Sie damit rechnen, dass Ihr Binärcode nach der Optimierung durch den zweiten Compiler neue Probleme enthält, auf die Sie bei der bisherigen Fehlersuche noch gar nicht gestoßen sind, weil sie einfach nicht vorhanden waren. Bei einer Singlethread-Anwendung ist die Wahrscheinlichkeit hierfür nur bei Abweichungen des Quellcodes oder des Compilers vom C++/C#-Standard gegeben, also eher begrenzt. Bei einer Multithread-Anwendung ist sie wesentlich größer, da der MP-optimierende Compiler von einem anderen Quellcode ausgeht.

Natürlich sollte immer auch die Release-Version getestet werden. Das gilt aber ganz besonders, wenn mehrere Compiler verwendet werden und die Release-Version MP-optimiert wurde.

10.6 Der Profiler

Ein Profiler ist eine Software, die die Laufzeit einer Anwendung sehr präzise misst. Unter Anwendung kann sowohl ein einzelnes Programm als auch eine Gruppe von Programmen oder auch nur eine Teilkomponente verstanden werden. Der Profiler kann sowohl die Laufzeit einzelner Funktionen als auch einzelner Anweisungen messen. Daneben sammelt er zusätzliche Daten wie die Anzahl der Funktionsaufrufe, der Schleifendurchläufe oder auch der Speicherzugriffe. Außerdem bereitet der Profiler diese Daten auf und stellt sie dem Benutzer grafisch und möglichst übersichtlich zur Verfügung.

Die regelmäßige Verwendung eines Profilers kann, je nach Projekt-Anforderungen, beinahe ebenso wichtig sein wie der Einsatz des Debuggers. Ein Profiler sollte vor allem dann angewendet werden, wenn ein Programm hohe Anforderungen an die CPU stellt. Bei GUI-Anwendungen ist das nicht immer einfach zu beurteilen, da sie die meiste Zeit auf Benutzereingaben warten und somit die Gesamtlaufzeit des Programms keinen Anhaltspunkt auf die Belastung des Prozessors liefert. So stellt eine Textverarbeitung meist keine hohen Hardware-Anforderungen und die Effizienz spielt nur eine untergeordnete Rolle. Aber es kann Teilbereiche geben, für die das nicht gilt. Der Extremfall etwa ist eine integrierte Spracherkennung,

die sehr aufwändig ist. Oder auch die Grammatikprüfung kann kurzfristig die CPU belasten. Bei einer in Bezug auf die Laufzeit unkritischen Anwendung kann die Performance für Teile des Programms also durchaus eine wichtige Rolle spielen. Bei Anwendungen, die per se hohe Anforderungen an die Hardware stellen, muss natürlich besonderer Wert auf eine gute Effizienz gelegt werden. Das können Server-Anwendungen wie Datenbanken, CRM-Programme oder numerische Simulationen sein. Oder es kann sich um so unterschiedliche Anwendungen wie eine GUI-Applikation zur 3D-Visualisierung, eine Entwicklungsumgebung oder ein Spiel handeln. In jedem Fall sollte ein Profiler eingesetzt werden, wenn klar ist, dass ein Teil der Anwendung unter dem Aspekt des Laufzeitverhaltens kritisch sein könnte.

Wenn ein Profiler während der Entwicklung eingesetzt werden soll, dann muss dies frühzeitig erfolgen und nicht erst, wenn das Projekt kurz vor der Fertigstellung steht. Das Problem eines späten Einsatzes ist ein dreifaches:

- die Interpretation der Daten
- die Änderung der betroffenen Programmteile
- die Auswirkung auf den Zeitplan

Das gilt natürlich sowohl für Multithread- als auch für Singlethread-Anwendungen. Allerdings liegen die Dinge in der erstgenannten Gruppe von Anwendungen meist komplizierter. Hier ist es sehr viel schwieriger, die Abfolge der Aktionen und das Zusammenspiel der einzelnen Threads zu beurteilen. Durch einen frühzeitigen Einsatz können Erfahrungen gewonnen und Varianten getestet werden, wenn der Code noch relativ einfach ist. Das erleichtert die Interpretation der Daten.

Wenn die Anwendung weit fortgeschritten und parallelisiert ist, sind Änderungen in der Parallelisierung oft schwer durchzuführen. Erweist sich etwa die Art der Synchronisation als ineffizient, ist eine Änderung aufwändig. Als Beispiel mag die Klasse `ArrayList` des .NET Frameworks dienen, die ja schon in Abschnitt 2.5 *Die Parallelisierung einer Anwendung* verwendet wurde. Es ist sehr leicht, ihre synchronisierte Version einzusetzen. Aber es ist schwierig, sie nachträglich durch einen STL-Container zu ersetzen. Besonders dann, wenn sie häufig verwendet wird. Der Profiler kann frühzeitig klären, mit welchem Mehraufwand durch den Einsatz der synchronisierten Version von `ArrayList` zu rechnen ist. Der letzte Abschnitt dieses Kapitels wird hierzu ein Beispiel zeigen.

Ein anderes Problem paralleler Anwendungen sind überraschend lange Wartezeiten in den Synchronisationsfunktionen. Deadlocks können meist durch die im Abschnitt 10.4 *Deadlock Detection* vorgestellten Strategie erkannt werden, was schwierig genug ist. Nicht einfacher ist die Beurteilung von Wartezeiten. Ein Profiler kann hierzu die Mittelwerte und gegebenenfalls auch die Extrema angeben, und das auch noch threadspezifisch. So wird die Beurteilung möglich, ob die Wartezeiten im Rahmen der Erwartungen liegen und ob sie Probleme zur Folge haben können. Ein solches Problem kann sein, dass der Durchsatz einer Anwendung klein ist, obwohl die CPU nicht annähernd ausgelastet wird.

Wenn sich die Parallelisierung einer Anwendung als unzureichend erweist, ist es sehr schwierig, sie kurz vor Ende eines Projektes zu ändern. Das gilt sowohl für die Frage, ob überhaupt mehrere Threads eingesetzt werden sollen, als auch für andere grundlegende Aspekte, beispielsweise wie die Threads miteinander kommunizieren sollen. Eine Anwendung, die bisher die Windows Messages zum Nachrichtenaustausch eingesetzt hat, lässt sich nicht ohne weiteres auf Pipes umstellen. Auch wenn dies technisch nicht allzu schwierig ist, kann es doch erhebliche Nebenwirkungen geben. Das ist zwar alles lösbar, wird aber den Zeitplan sprengen, da nicht nur der Aufwand in der Entwicklung steigt, sondern vor allem der Testaufwand. Es ist daher wichtig, dass das grundlegende Design in Hinblick auf die Parallelität frühzeitig festgelegt, überprüft und eventuell korrigiert wird. Ein wichtiges Mittel der Überprüfung ist der Profiler.

10.7 Grundsätzliches zum Einsatz des Profilers

Ein Profiler misst die Laufzeit einer Anwendung. Entscheidend für die Beurteilung der Performance einer Anwendung ist weniger die Gesamtlaufzeit als vielmehr die Relation zwischen den einzelnen Komponenten. Nur dadurch können Engpässe erkannt und Verbesserungen durchgeführt werden, die insgesamt zur Steigerung der Performance führen.

Aus Entwickler-Sicht sind die Funktionen die wichtigsten Komponenten. Daher misst der Profiler die Zeitspanne, die die CPU mit der Ausführung einer Funktion beschäftigt ist, und bildet die Summe, den Mittelwert etc. über alle Aufrufe dieser Funktion. Andere Komponenten von Interesse sind ganze Module, also meist DLLs, oder einzelne Befehle auf Source-Code oder Assembler-Ebene.

Der Profiler misst pro Funktion oder Modul zwei Zeitspannen. Das ist einmal die Zeit, die zwischen dem Aufruf und der Rückkehr der Funktion vergeht. Diese Zeit wird auch als *Total Time* bezeichnet. Die zweite gemessene Zeitspanne ist immer kleiner oder gleich und bezieht sich auf die Zeit, in der die CPU die Funktion ausführt, ohne den Teil zu berücksichtigen, der mit untergeordneten Funktionen verbracht wird. Sie wird auch *Self Time* genannt. Das bedeutet, dass der Profiler für die `main`-Funktion praktisch die komplette Laufzeit als Total Time messen wird bei einer Self Time, die um mehrere Größenordnungen geringer ist.

Die Messung der Zeitspannen kann in unterschiedlichen Einheiten erfolgen. Eine Möglichkeit ist die Angabe der Zeiten in Sekunden, die andere sind die *Clockticks*. Ein Clocktick ist die kleinste diskrete Zeitspanne eines Prozessors. Seine Dauer ist der Umkehrwert der Prozessor-Frequenz. Eine Instruktion, also eine Assembler-Anweisung, benötigt meist einen oder mehrere Clockticks. Ein moderner Prozessor kann aber auch eine Reihe von Instruktionen innerhalb eines Clockticks gemeinsam abarbeiten. Das Verhältnis von Clockticks zu Instruktionen wird auch als *Clockticks per Instruction* (CPI) bezeichnet. Ein Wert unter 1 gilt als sehr gut. Er allein sagt aber noch nichts über die Optimierung des Codes aus, da viele Aspekte eine Rolle spielen. Ein Code, der viele aufwändige Instruktionen ausführt, kann auch mit einem CPI-Wert weit über 1 hoch optimiert sein.

Die Profiler setzen unterschiedliche Techniken zur Zeitmessung ein, die sich von Hersteller zu Hersteller unterscheiden. Jedes dieser Verfahren hat seine spezifischen Vor- und Nachteile, weswegen jeder Profiler gleich mehrere Techniken kombiniert. Wichtige Unterschiede bestehen beispielsweise darin, ob der Code speziell für den Profiler übersetzt werden muss, ob tatsächlich Zeiten gemessen werden oder ob der Profiler über Stichproben zu einer Statistik gelangt. Das hat Auswirkungen auf die Genauigkeit der Messungen und auch darauf, um welchen Betrag die Laufzeit durch den Profiler ansteigt.

Unter dem Gesichtspunkt des Multithreadings ist es wichtig, die Synchronisation der einzelnen Threads beurteilen zu können. Bei mehreren Threads ist es sehr schwierig, ein genaues Bild zu erhalten. Deshalb ist es wichtig, dass der Profiler die Laufzeiten der verschiedenen Funktionen den einzelnen Threads zuordnen kann. Besonders interessant sind die ermittelten Wartezeiten (*Wait Time*), also die Zeit, die der Thread im Warte-Zustand verbringt. Mit ihrer Hilfe kann beurteilt werden, ob ein Thread häufig beziehungsweise lange auf einen anderen warten muss.

Daneben gibt es viele andere Aspekte, die eine Rolle spielen können. Wenn der Profiler den Zugriff auf den L1- und L2-Cache misst, kann daraus auch abgeleitet werden, ob es Sinn macht, die Zahl der Kontextwechsel zu verringern. Beispielsweise durch eine Änderung der Priorität oder einer Verringerung der Zahl der Threads. Das ist dann aber schon die fortgeschrittene Optimierung, die nur für wenige ausgewählte Programmteile überhaupt in Frage kommen sollte.

Es gibt mehrere sehr gute Profiler, deren Einsatz zu empfehlen ist. Dazu gehört sicherlich der TrueTime-Profiler von Compuware (ursprünglich Numega). Er ist einfach zu bedienen, übersichtlich und nach einer sehr kurzen Einarbeitungszeit anwendbar. Eine Testversion finden Sie unter `www.compuware.com`. Der mit Abstand umfangreichste Profiler wird von Intel angeboten und heißt VTune (Testversion unter `www.intel.com`). Sein Einsatz ist mit einem größeren Aufwand verbunden. Dafür misst er nahezu alles, was auch nur annähernd für die Optimierung eine Rolle spielen könnte. In den folgenden Beispielen verwende ich diesen Profiler, da er auch unter Multithreading-Aspekten am meisten bietet.

10.8 Drei Beispiele für den Einsatz des Profilers

Sie können die folgenden Beispiele mit der Testversion des VTune-Profilers selber nachvollziehen. Bevor Sie allerdings VTune das allererste Mal einsetzen, spielen Sie am besten zunächst das mitgelieferte Tutorial durch. Hier wird alles erklärt, was Sie im Folgenden benötigen. Und planen Sie genügend Zeit ein, denn

die Handhabung eines Profilers will erst gelernt werden. Wenn Sie einen anderen Profiler verwenden, sollte es Ihnen leicht möglich sein, die gleichen oder zumindest ähnliche Messungen durchzuführen. Eventuell fehlen aber die threadspezifischen Daten.

10.8.1 Das Beispiel Simplethreads.exe mit einem Thread

In diesem und dem nächsten Abschnitt verwende ich das Beispiel Simplethreads.exe aus Kapitel 2, um den Profiler einzusetzen und mit seiner Hilfe herauszufinden, wo die Zeit zur Ermittlung der Primzahlen hauptsächlich bleibt. Vermutungen hatte ich bereits in Kapitel 2 angestellt, aber nun wird es möglich, diese handfest zu belegen und das Verhalten bis auf Funktions- oder Anweisungsebene zu durchleuchten. Durch den Vergleich der Ergebnisse der Singlethread-Variante mit denen der Multithread-Variante kann man sehr leicht feststellen, welche Unterschiede hinsichtlich der Laufzeit auftreten und wie sie zu Stande kommen. Dadurch wird es möglich, Konsequenzen für eine weitere Optimierung des Programms zu ziehen.

Da es meistens nicht ganz einfach ist, die Ergebnisse der Laufzeitmessung einer Multithread-Anwendung zu interpretieren, beginne ich mit einer ausführlichen Darstellung der Singlethread-Variante. Ich erläutere zunächst die wesentlichen Aspekte, die für die Auswertung eine Rolle spielen. Dadurch wird die Interpretation der Multithread-Ergebnisse im nächsten Abschnitt wesentlich einfacher. Aber natürlich ist es in der Praxis nicht notwendig, bei jeder Untersuchung einer Multithread-Anwendung zunächst die Singlethread-Variante zu betrachten. In den meisten Fällen wird sie entweder nicht existieren oder sich erheblich von der Multithread-Version unterscheiden. Selbst wenn sie existiert, steigert ihre Untersuchung den Aufwand erheblich und meist unnötig. Voraussetzung ist allerdings ein gewisses Maß an Erfahrung, da die Ergebnisse leicht falsch interpretiert werden können.

In Abschnitt 2.5 *Die Parallelisierung einer Anwendung* wurden die Ergebnisse mehrerer Testläufe von `Simplethreads.exe` vorgestellt und diskutiert. Dabei zeigte sich, dass die Effizienz der parallelen Version mit zwei Threads umso besser wurde, je größer die maximal zu bestimmende Primzahl war. Für die folgenden Untersuchungen wähle ich als obere Grenze den Wert 6.400.000, so dass sich eine Laufzeit der Berechnung von rund 7 Sekunden in der Singlethread-Variante ergibt. Das ist lang genug, um vernünftige Messungen zu erlauben, und kurz genug, um die Messungen leicht wiederholen zu können. Beachten Sie bei der Auswahl der Problemgröße, dass der Profiler die Ausführungsgeschwindigkeit deutlich senken wird. Deutlich heißt, dass der Unterschied mehr als eine Größenordnung betragen kann.

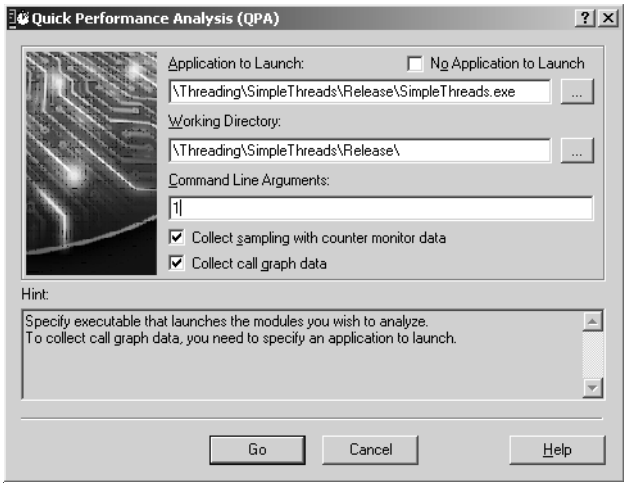

Abbildung 10.3: Der VTune-»Quick Performance Analysis Wizard«

Wenn Sie das Beispiel selber nachvollziehen wollen, dann starten Sie den Intel-Profiler VTune und wählen den QUICK PERFORMANCE ANALYSIS WIZARD aus. Stellen Sie dort als zu startende Anwendung das Programm Simplethreads.exe in der Release-Version ein. Überprüfen Sie vorher, dass das Programm mit den Debug-Informationen übersetzt wurde. Kreuzen Sie beide Check-Boxen zum Sammeln der Daten an und setzen Sie als Kommandozeilen-Argument den Wert 1 ein. Für die Multithread-Variante müssen Sie diesen Wert später auf 2 ändern.

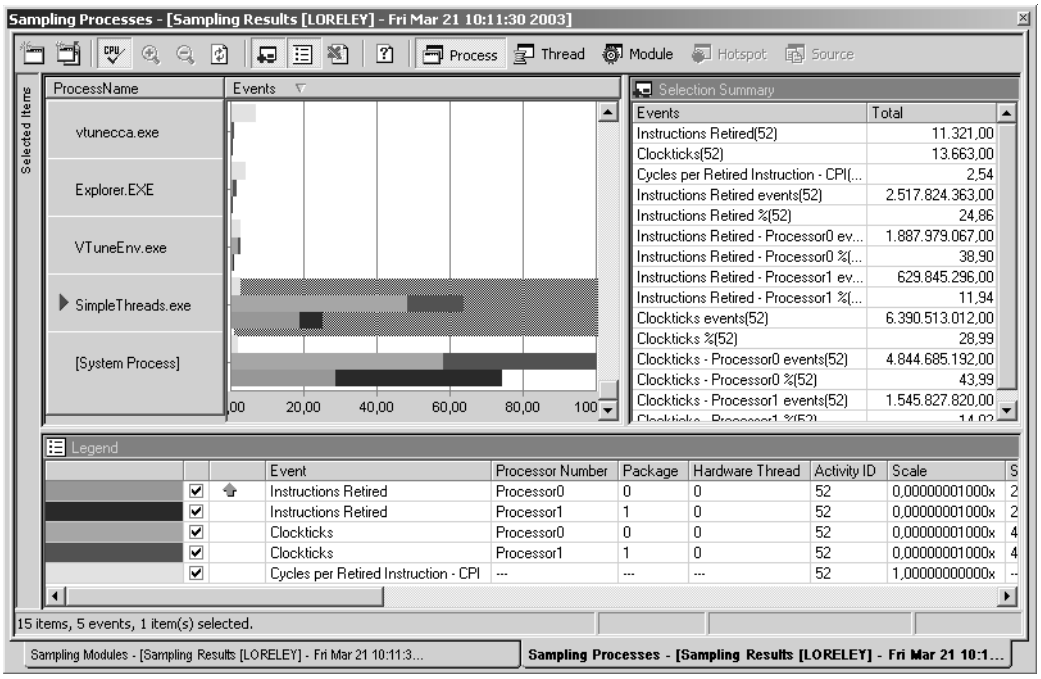

Abbildung 10.4: Eine Übersicht zur Laufzeit von `Simplethreads.exe` mit einem Thread

VTune startet das Programm automatisch und beginnt mit dem Sammeln der Daten. Geben Sie anschließend den Wert 6400000 ein, wenn das Programm Simplethreads.exe dazu auffordert, eine obere Grenze für die Primzahlen anzugeben. Da der Profiler das Programm ein zweites Mal startet, müssen Sie den Vorgang wiederholen. Zwar wird auch die Zeit, die sie für die Eingabe benötigen, gemessen, aber das spielt für die spätere Auswertung keine Rolle. Der Profiler unterscheidet die Zeiten auch darauf, ob auf eine Benutzereingabe gewartet wurde. Dann dauert es noch eine Weile und schließlich ist der Profiler-Lauf abgeschlossen. Mit Hilfe der verschiedenen Fenster können Sie nun durch die Ergebnisse navigieren und sie aus verschiedenen Blickwinkeln betrachten.

Zunächst ist es wichtig, sich einen ersten Überblick zu verschaffen und zu sehen, ob die Ergebnisse plausibel sind. Diese Übersicht liefert der VTune-Profiler im Fenster SAMPLING RESULTS. Gruppieren Sie in diesem nach den Prozessen und Sie erhalten ein ähnliches Ergebnis wie in Abbildung 10.4. Im Balkendiagramm zeigt der mittlere Wert die Zahl der Clockticks für den jeweiligen Prozess und zwar aufgeteilt nach Prozessor. Zu erkennen ist also eine Verteilung von etwa 75 % zu 25 %. Die rechte Tabelle gibt unter anderem die Gesamtzahl der Clockticks an, das sind in diesem Fall etwa 6.39e9. Sie können ignorieren, dass das System mehr Clockticks für sich verbucht. Das ist ein Artefakt auf Grund der Messung. Die anderen Prozesse belasten die Prozessoren nur wenig. Das ist auch gut so, da sie ansonsten die Messung beeinflussen würden.

Die Anzahl der gemessenen Clockticks passt gut ins Bild, da die Laufzeit rund 7 Sekunden beträgt und die Prozessoren mit 933 MHz arbeiten. Einen ersten Hinweis auf die Performance der Anwendung gibt der

Wert `Cycles per Retired Instruction` (CPI), der mit 2.54 eher mager ausfällt. Ein Wert von unter 1 gilt als sehr gut und über 2 als eher schlecht. Der Singlethread-Code lässt somit wahrscheinlich noch eine Menge Spielraum für Verbesserungen, was nicht überraschend ist, da ein sehr einfacher Algorithmus gewählt wurde. Das soll hier aber keine Rolle spielen, denn uns interessiert vor allem der Vergleich zur Multithread-Version.

Einen genaueren Einblick liefert das Fenster CALL LIST. Hier werden alle Funktionen aller Module mit ihren Laufzeiten aufgeführt. Abbildung 10.5 zeigt in der Spalte SELF TIME, dass die gesamte Laufzeit der Anwendung 14.6e6 Mikrosekunden betrug. Also etwa doppelt so lange, wie sie ohne Profiler benötigen würde. Im Fenster wurde die Funktion `Prim` ausgewählt (Focus Function). Sie trägt einschließlich ihrer untergeordneten Funktionen 99.8 % zur Laufzeit bei. Davon liegen 12.8 Sekunden direkt in der Funktion, das sind rund 87.5 %, während 12.5 % von den aufgerufenen Funktionen verursacht werden. Diese Werte zeigen deutlich an, welche Funktion maßgeblich für die Laufzeit verantwortlich ist und eine genauere Betrachtung verdient. Interessant für den späteren Vergleich mit der Multithread-Variante ist auch der Wert `Edge Wait Time`, der angibt, wie lange der Thread während der Ausführung der Funktion im Warte-Zustand war. Der Wert ist mit 67 Millisekunden zu vernachlässigen.

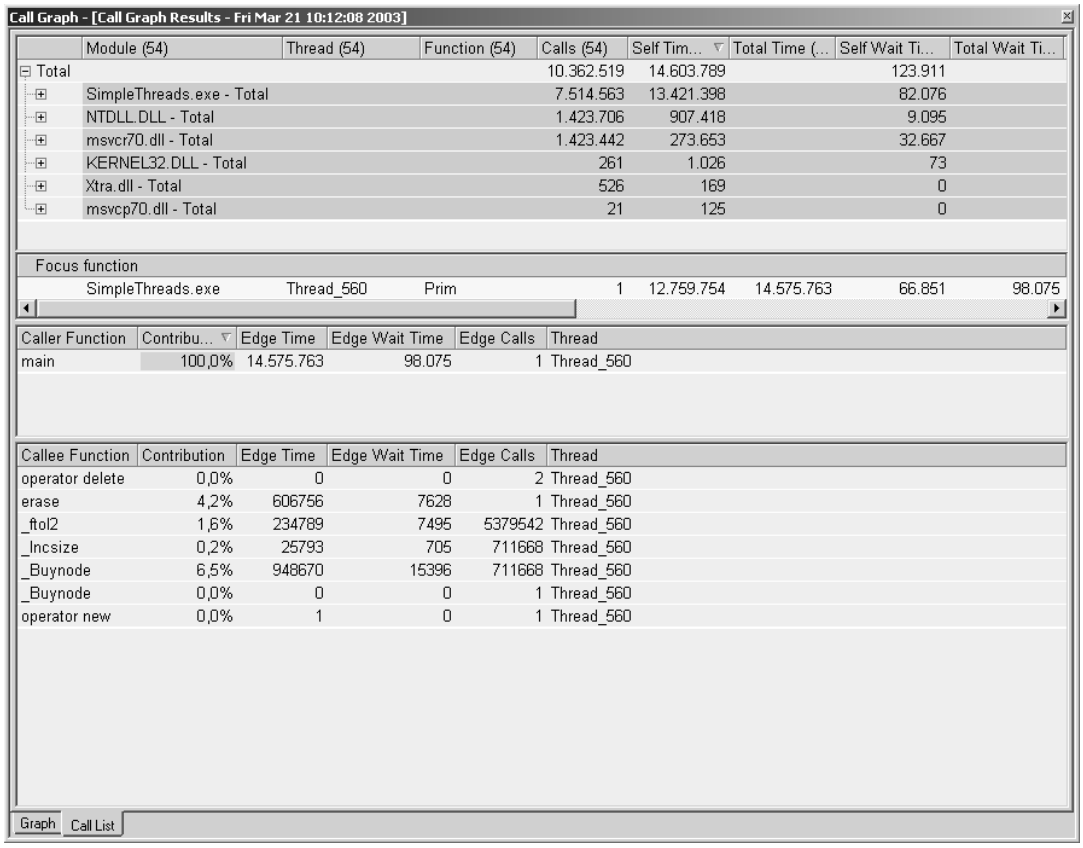

Abbildung 10.5: Die Funktionsliste

Der mittlere Teil des Fensters CALL LIST zeigt die aufrufende Funktion, hier also `main`, und der untere Teil die aufgerufenen Funktionen. Die Listen können nach den verschiedenen Kriterien sortiert werden und sind damit auch sehr gut navigierbar. Sie liefern die entscheidenden Hinweise auf die wesentlichen Funktionen. Die Funktionsliste ist zusammen mit dem Funktionsgraphen das wichtigste Instrument, um die Pro-

grammteile zu identifizieren, die den größten Teil der Zeit konsumieren. Aber genauso wichtig ist die Tatsache, dass es mit ihrer Hilfe möglich ist, die Zusammenhänge zwischen den einzelnen Funktionen zu erkennen. Also welche Funktion wird überhaupt von wem aufgerufen, welchen Anteil hat sie an der übergeordneten Funktion, auf welchem Thread wird sie ausgeführt?

Abbildung 10.6 zeigt den Funktionsgraphen. Er gibt einen besseren visuellen Eindruck von den Zusammenhängen. Allerdings kann er auch schnell unübersichtlich werden, weshalb es wichtig ist, sich auf die wesentlichen Teile zu beschränken. Unser Programm ist aber so einfach, dass die Zusammenhänge leicht zu erkennen sind. Es gibt nur einen Thread, der neben `_DllMainCRTStatup` (diese Funktion ruft `DllMain` auf) die Funktion `mainCRTStartup` aufruft, die ihrerseits `main` aufruft und die wiederum `Prim` etc. Die Messwerte für die einzelnen Funktionen werden im oberen Teil in Tabellenform angegeben. Damit ist es sehr einfach, durch die Funktionen zu navigieren und einen Eindruck von den Zusammenhängen zu bekommen. Beachten Sie bitte, dass die Funktionen pro Thread aufgeführt werden. Das ist zwar hier nur einer, aber damit wird es später möglich sein, auch eine threadbezogene Auswertung durchzuführen.

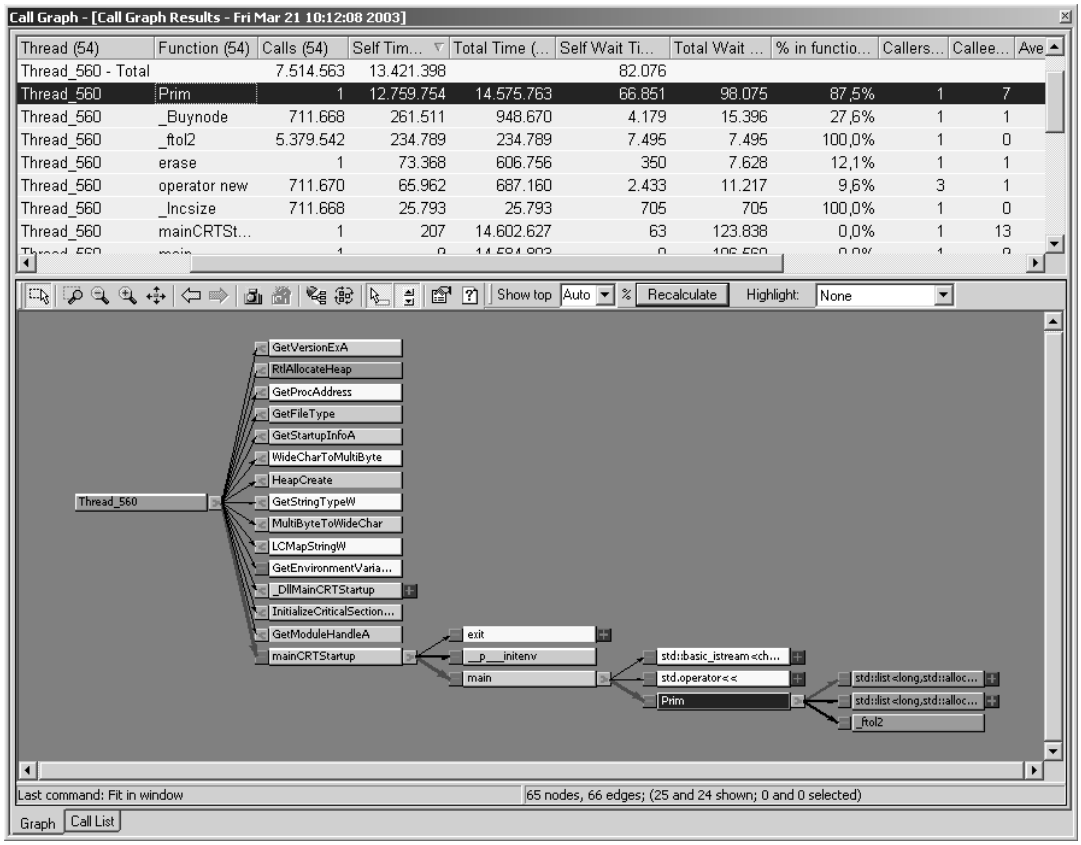

Abbildung 10.6: Der Funktionsgraph

Nachdem wir nun einen guten Überblick gewonnen haben, ist es an der Zeit, sich dem Quellcode zuzuwenden. Das ist direkt aus den beiden Seiten GRAPH und CALL LIST möglich, indem einfach die gewünschte Funktion ausgewählt wird. Hier ist das natürlich `Prim`.

Die Abbildung 10.7 zeigt den Quellcode der Funktion `Prim` zusammen mit den Angaben zur Laufzeit. Ich habe als Einheit den prozentualen Anteil der Anweisung an der Laufzeit des Moduls gewählt. Die Spalte

CLOCKTICKS zeigt, dass die Funktion den weitaus größten Teil der Zeit mit dem Iterieren der Liste verbringt. Die Division und der Vergleich des Teilers spielt nur eine untergeordnete Rolle. An dieser Stelle könnte man sich nun die Befehle genauer anschauen und auch den Assembler-Code mit hinzunehmen. Es wäre möglich, weitere Zähler auszuwerten wie Cache-Fehler oder falsche Verzweigungsvorhersagen. Aber uns interessiert hier ja gar nicht, inwieweit der Algorithmus als solcher noch verbessert werden kann. Wichtig ist vielmehr der Vergleich mit der Multithread-Version, der im nächsten Abschnitt folgt.

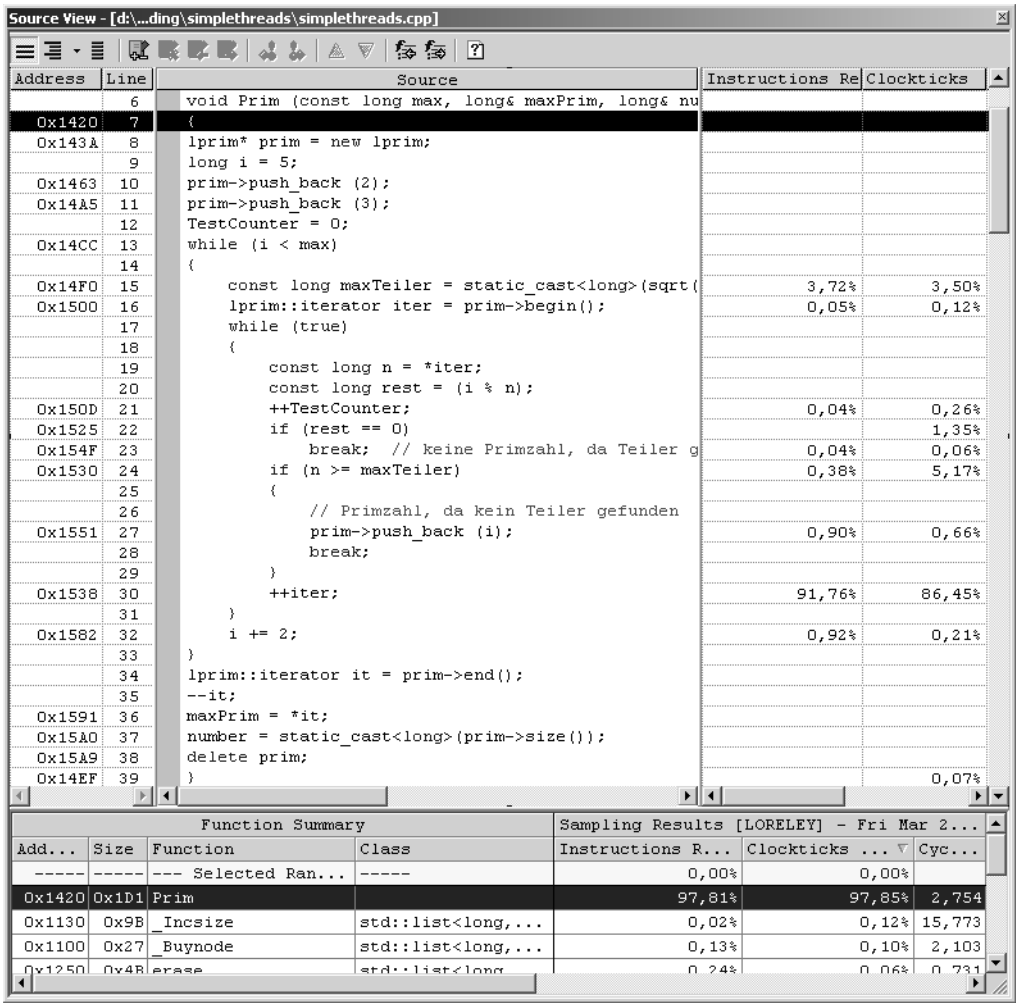

Abbildung 10.7: Der Quellcode der Funktion `Prim`

10.8.2 Das Beispiel Simplethreads.exe mit zwei Threads

Der Profiler muss ein zweites Mal angewendet werden, um die Ergebnisse für die parallele Version mit zwei Threads zu erhalten. Dazu wiederholen Sie das Vorgehen aus dem letzten Abschnitt mit dem Kommandozeilen-Parameter 2. Die Abbildung 10.8 zeigt die Übersicht der Messergebnisse für den Prozess Simplethreads.exe.

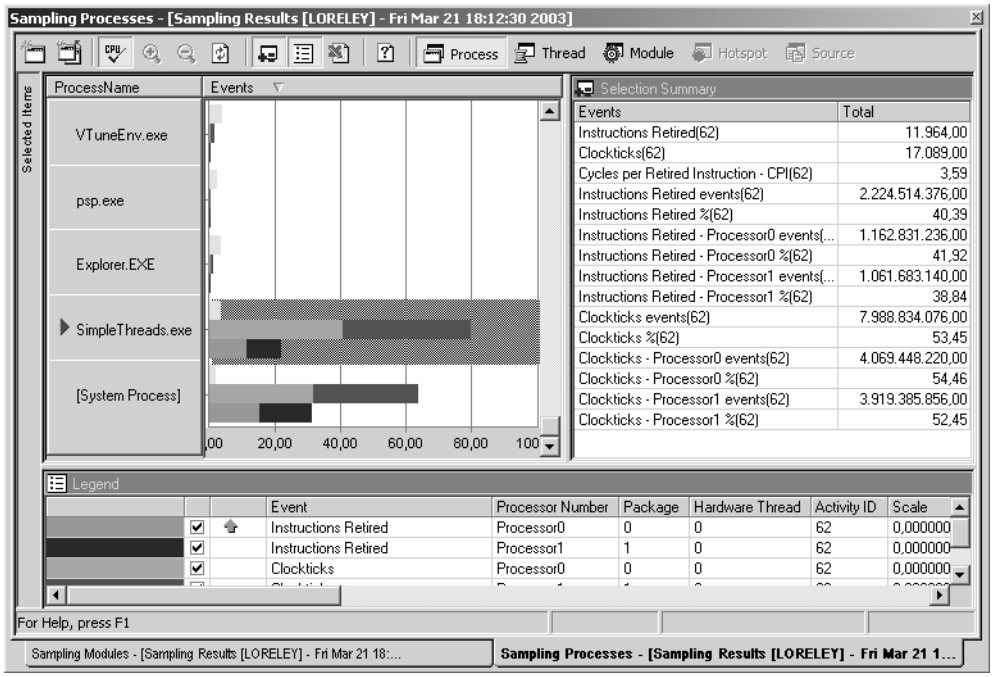

Abbildung 10.8: Eine Übersicht zur Laufzeit von `Simplethreads.exe` *mit zwei Threads*

Die Zahl der Clockticks liegt nun bei 7.99e9 und damit rund 25 % über der Singlethread-Variante. Das bedeutet eine Effizienz von rund 80 % und liegt sehr nah an den Ergebnissen aus Abschnitt 2.5 *Die Parallelisierung einer Anwendung*, wenn man berücksichtigt, dass der Profiler die Messung beeinflusst. Die Verteilung auf die CPUs ist sehr ausgewogen mit 4.07e9 zu 3.92e9 Clockticks. Die Lastverteilung des Betriebssystems ist also für diese Anwendung völlig ausreichend. Der CPI-Wert ist mit 3.59 schlecht. Überraschenderweise ist die Zahl der Instruktionen mit 2.22e9 etwas geringer als in der Singlethread-Version, obwohl zusätzlicher Code für die Synchronisation erforderlich war.

Die parallele Version von Simplethreads.exe besteht aus insgesamt drei Threads, dem Primary Thread und den beiden Threads, auf denen die wesentliche Funktion `PrimEx` parallel ausgeführt wird. Dementsprechend werden die Funktionen in der Call-Liste diesen drei Threads zugeordnet. Die Abbildung 10.8 zeigt, dass der Thread mit der ID 5D8 die `WaitForMultipleObjects`-Funktion aufruft. Es handelt sich also offensichtlich um den Primary Thread, der mit dieser Funktion auf das Ende der beiden Worker-Threads wartet. Der Thread befindet sich fast während seiner kompletten Laufzeit in dieser Funktion, die den Thread rund 23 Sekunden lang suspendiert, bei einer effektiven Ausführungszeit von 14 Mikrosekunden. Übrigens finden Sie sehr weit unten in der Liste auch die Funktion `_beginthreadex` mit 2 Aufrufen von jeweils 280 Mikrosekunden Laufzeit.

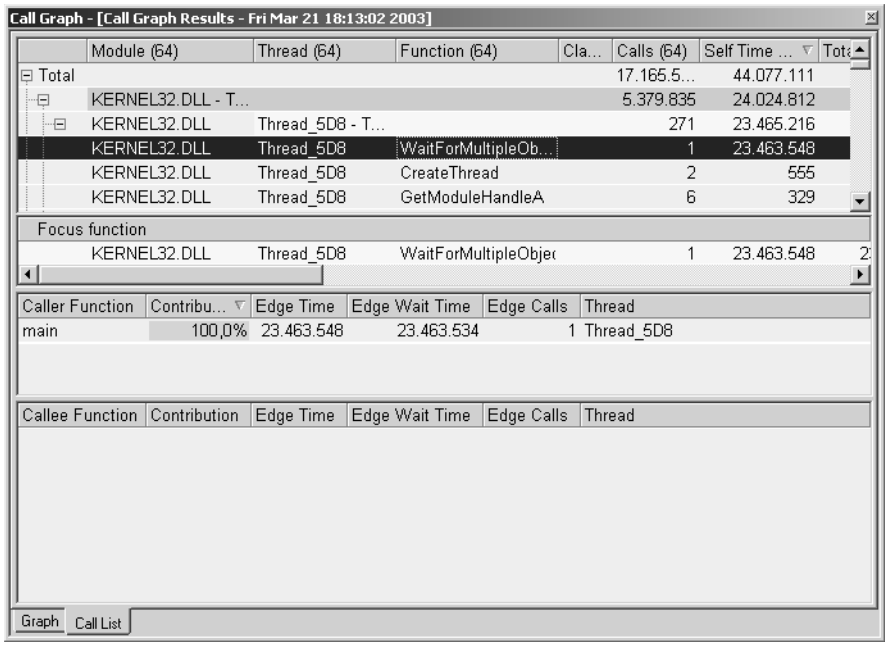

Abbildung 10.9: Die Funktionsliste

Wesentlich mehr passiert auf den beiden anderen Threads mit den IDs 5E4 und 5E8. Hier wird jeweils die Funktion `PrimEx` aufgerufen. Abbildung 10.9 zeigt, dass jede der beiden Funktionen rund 7.1 Sekunden der 7.4 Sekunden Gesamtlaufzeit des jeweiligen Threads für ihren eigenen Code verwendet und die restliche Zeit für Kind-Funktionen. Das ist kein nennenswerter Unterschied zur Singlethread-Version. Der untere Teil des Call-Graphen visualisiert den Zusammenhang der Funktionen und ihre Verteilung auf die verschiedenen Threads. Ich habe einige weniger wichtige Funktionen entfernt, damit der Graph nicht zu unübersichtlich wird. Man kann die drei Threads gut erkennen und die Tatsache, dass zwei von ihnen den identischen Code ausführen.

Um zu ermitteln, welchen Mehraufwand die Synchronisation der beiden Threads mit sich bringt, müssen zunächst die Laufzeiten der entsprechenden Funktionen ermittelt werden. Interessant sind nur die Funktionen innerhalb der `while`-Schleife in `PrimEx`, da nur sie hinreichend oft aufgerufen werden und zum Resultat signifikant beitragen können.

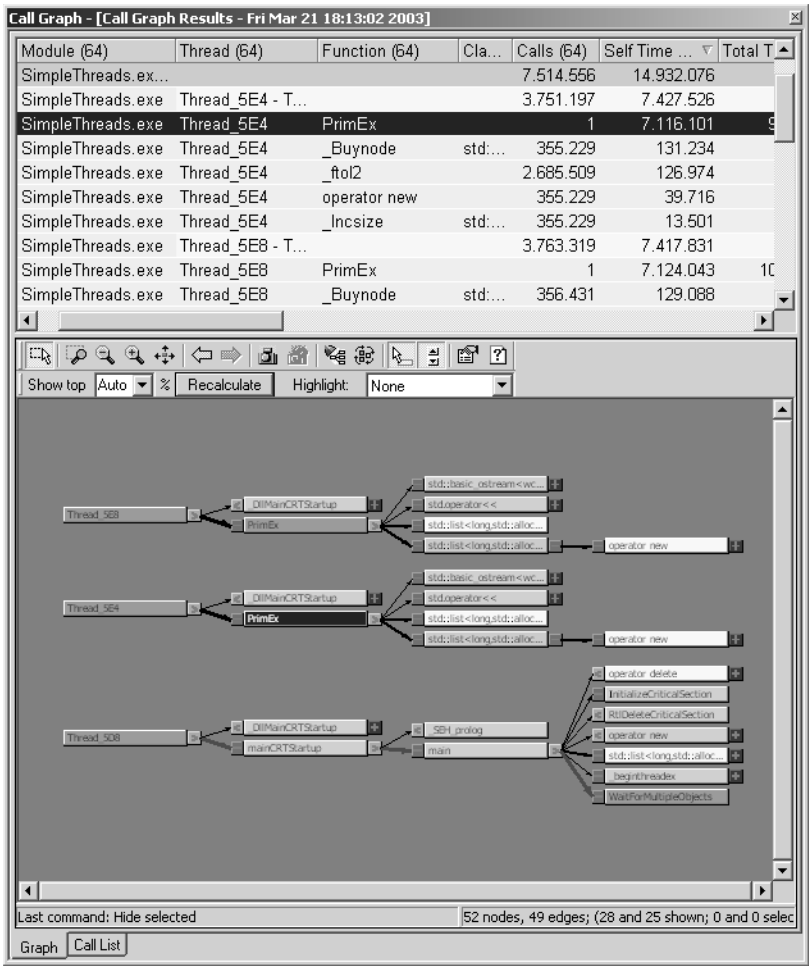

Abbildung 10.10: Der Funktionsgraph

Zunächst ist das die atomare Operation `InterlockedExchangeAdd`. Die Call-Liste zeigt, dass die Funktion pro Thread rund 2.7e6 Mal aufgerufen wird. Die Laufzeit beträgt etwa 0.28 Sekunden, aber die Wartezeit nur 0.6 und 2.5 Millisekunden. Das bedeutet, dass die Synchronisation der beiden Threads auf Grund dieser Funktion nur minimal ist, da so gut wie nie Wartezeiten auftreten. Und 0.28 Sekunden sind bei knapp 15 Sekunden Laufzeit immerhin knapp 2 % Mehraufwand, für den die Funktion verantwortlich gemacht werden kann, wenn man davon ausgeht, dass das Hochzählen der Variablen über eine gewöhnliche Addition vernachlässigt werden kann. Die zweite wichtige Funktion ist das Paar `EnterCriticalSection` und `LeaveCriticalSection`. Pro Thread beträgt der Aufwand für beide Funktionen zusammen knapp 1.7 Sekunden, von denen etwa eine Sekunde reine Wartezeit ist. Das sind 11 % der gesamten Laufzeit. Die Critical Section ist also für etwa die Hälfte des Mehraufwandes in der parallelen Version verantwortlich. Weitere 11 % verursacht der Code von `PrimEx` selber, wie Abbildung 10.11 deutlich macht.

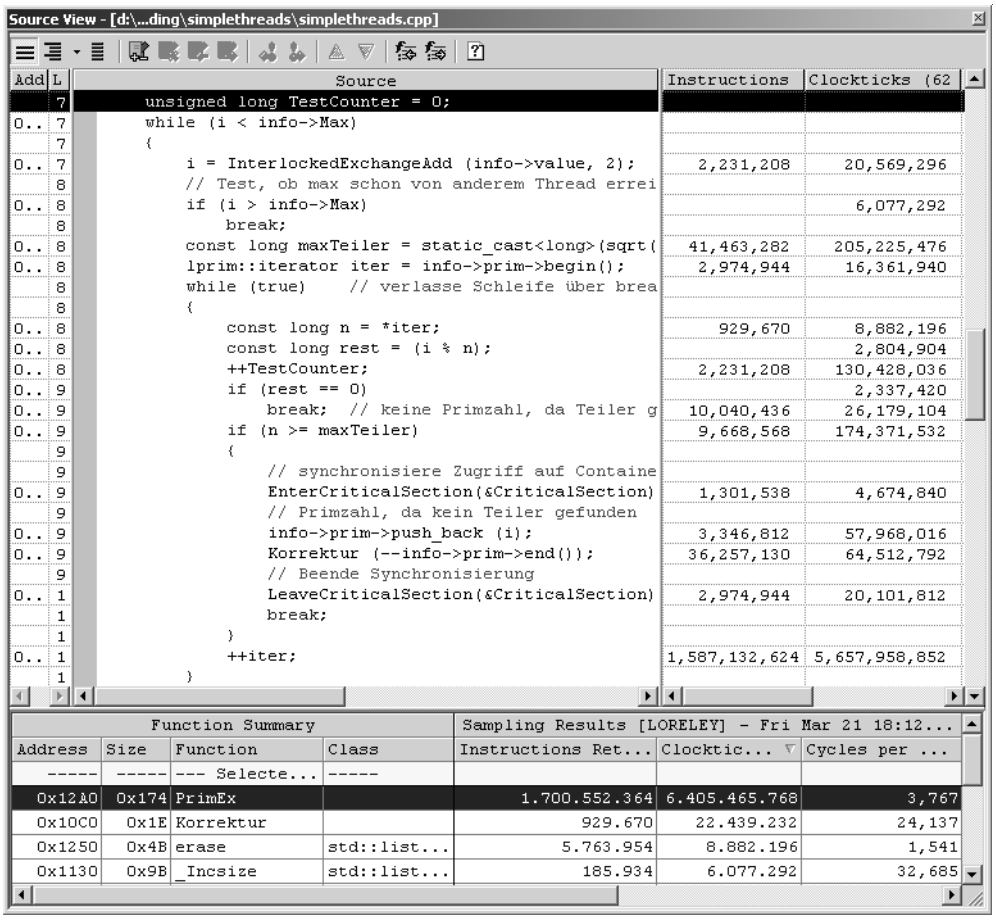

Abbildung 10.11: Der Quellcode der Funktion `PrimEx`

Die Zahl der Clockticks ist mit 6.4e9 um rund 11 % größer als in der Funktion `Prim` der Singlethread-Variante. Das liegt zum Teil daran, dass mehr Funktionsaufrufe notwendig sind. So kostet jeder Aufruf der drei gerade genannten Funktionen jeweils 2e7 Clockticks, das ist zusammen etwa 1 % Mehraufwand. Stärker aber fällt der etwas andere Aufbau der Funktion selbst ins Gewicht. Der Iterator des Containers ist in `PrimEx` nicht ganz so performant und zeichnet für nahezu 10 % des Mehraufwandes verantwortlich, obwohl die Zahl seiner Instruktionen überraschenderweise kleiner ist.

Das Ergebnis dieser Analyse ist also, dass ein Teil des Mehraufwandes der parallelen Version direkt an den Synchronisationsfunktionen hängt und der andere Teil an den Optimierungen, die der Compiler vornehmen kann. Diese hängen aber ebenfalls mit den Synchronisationsfunktionen zusammen. Die Critical Section ist, obwohl sie um eine Größenordnung seltener aufgerufen wird, erheblich teurer als die atomare Operation `InterlockedExchangeAdd`.

Eine Optimierung der parallelen Version ist möglich. Das Ziel muss sein, die Verwendung der Critical Section weiter einzuschränken. Dazu müssten die gefundenen Primzahlen über einen gewissen Zeitraum in einem threadspezifischen Container gesammelt werden. Dazu ist keine Synchronisation erforderlich. Bevor sie selber als Teiler benötigt werden, müssen sie in die Primzahl-Liste eingetragen werden. Dies geschieht aber sehr viel seltener, so dass auch viel weniger Critical Sections erforderlich sind. Außerdem kann ein Thread, der eine gesperrte Critical Section vorfindet, noch einige Iterationen weiterarbeiten,

bevor er erneut versucht, die Critical Section zu akquirieren. Dadurch wird es möglich, sowohl die Wartezeit als auch den Mehraufwand durch den Code der Critical Section zu verringern. Theoretisch wäre es damit möglich, auf eine Effizienz von rund 90 % zu kommen. Ob allerdings die letzen 10 % dann auch noch verbessert werden können, ist fraglich. Dazu wäre gegebenenfalls eine Optimierung der Schleife von Hand notwendig. So etwas lohnt sich aber nur in den seltensten Fällen.

10.8.3 Die C#-Variante

In Abschnitt 2.5 *Die Parallelisierung einer Anwendung* zeigte sich, dass die Umsetzung des Multithreading-Beispiels in C# problematisch war. Zwar ist es sehr einfach, den Code zu schreiben und zu synchronisieren, da das .NET Framework alle benötigten Mittel, namentlich die Klasse `ArrayList`, bereitstellt. Aber das Ergebnis war wegen der äußerst schwachen Effizienz wenig befriedigend. Es konnte vermutet werden, dass die Art, wie der Container sich intern synchronisiert, im Wesentlichen für den Mehraufwand verantwortlich ist.

Ich möchte das Beispiel noch einmal aufgreifen, um zu untersuchen, was genau zu dem Performance-Einbruch führt. Dazu wird wieder der VTune-Profiler eingesetzt. Da die Laufzeit erheblich größer ist als im C++-Beispiel, reduziere ich die Problemgröße auf den Wert 1600000 als maximale Primzahl. Die Laufzeit liegt dann in etwa bei der des vorigen Abschnitts. Das Programm Simplethreadsagain.exe wird direkt in der parallelen Version mit dem Kommandozeilen-Parameter 2 gestartet. Einen ersten Überblick zum Ergebnis gibt Abbildung 10.12.

Die Zahl der Clockticks für den Testlauf beträgt 11.4e9, also fast 50 % mehr als in der C++-Version, obwohl die Problemgröße wesentlich reduziert wurde. Die Verteilung auf die Prozessoren ist ausgeglichen.

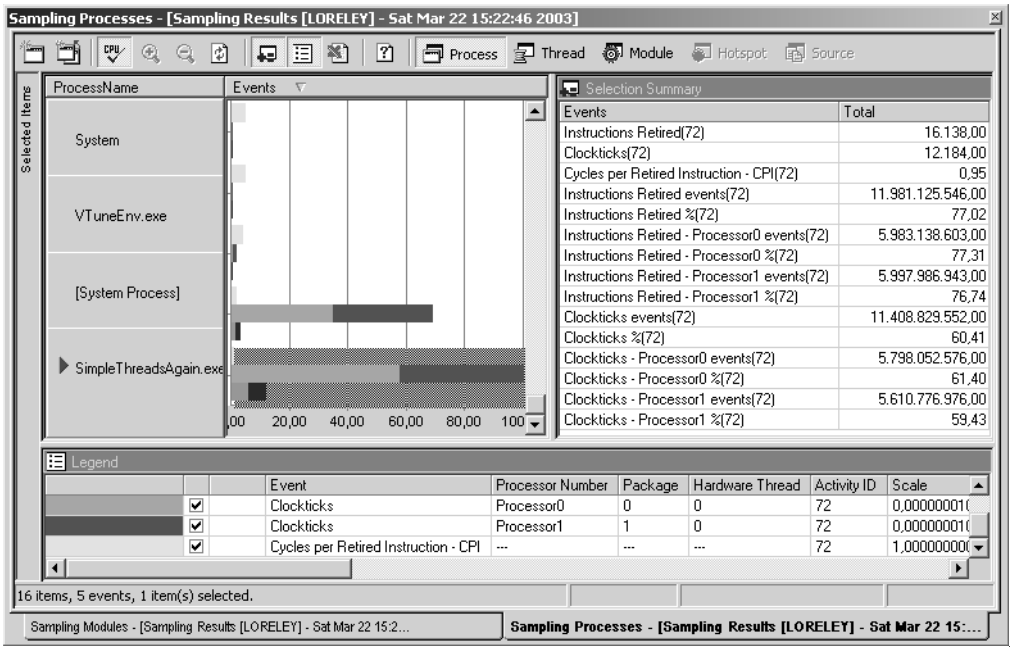

Abbildung 10.12: Eine Übersicht zur Laufzeit von `Simplethreadsagain.exe` mit zwei Threads

Welche Funktion für die größere Laufzeit verantwortlich ist, zeigt die Funktionsliste in Abbildung 10.13. Der Profiler gibt für die Funktion `PrimEx`, die näherungsweise die komplette Laufzeit des Programms umfasst, eine Gesamtzeit (Total Time) von 221 Sekunden an. Dieser Wert darf nicht mit der Laufzeit ver-

wechselt werden und ist einfach nur die Addition der gemessenen Zeiten zwischen dem Beginn der Funktion und ihrem Ende. Der Wert ist auch nicht um den Betrag korrigiert, den der Profiler für sich selbst benötigt.

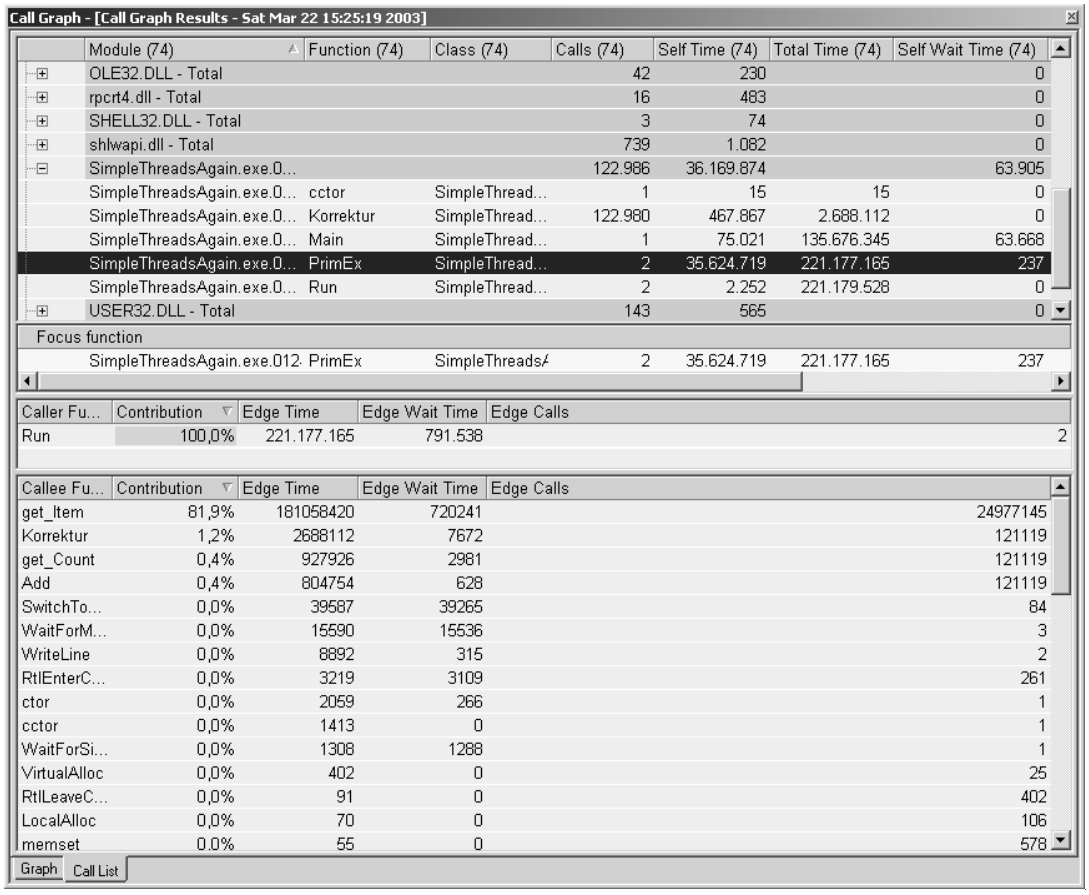

Abbildung 10.13: Die Funktionsliste von `Simplethreadsagain.exe`

Von den 221 Sekunden werden aber nur 36 Sekunden im Code der Funktion selbst verbracht (Self Time), das sind 16 %, der Rest geht auf das Konto der Kind-Funktionen. Dies ist ein drastischer Unterschied zu den Messungen des letzten Abschnittes. Es macht also wenig Sinn, den Code der Funktion `PrimEx` zu untersuchen, vielmehr ist es notwendig, die Aufrufe kostspieliger Funktionen zu verringern. Der untere Teil in Abbildung 10.13 zeigt, welche Funktion ganz überwiegend (82 %) für die große Menge an Clockticks verantwortlich ist, nämlich `get_Item` der Klasse `ArrayList` bzw. der Wrapper-Klasse `SyncArrayList`.

In `PrimEx` wird ja die synchronisierte Variante von `ArrayList` verwendet. Das bedeutet, dass die Memberfunktionen der Klasse über einen Wrapper angesprochen werden, der für die Synchronisation sorgt und dann die eigentliche Memberfunktion aufruft. Der Name der Wrapper-Klasse ist in diesem Fall `SyncArrayList`. Dieser Zusammenhang wäre im unteren Teil von Abbildung 10.14 zu erkennen, wenn die Boxen so groß wären, dass die kompletten Namen reinpassen würden. Die Abbildung zeigt einen Ausschnitt des deutlich größeren Funktionsbaumes. Ich habe den Teil vergrößert, der fast ausschließlich die Laufzeit des Programms bestimmt. Von links nach rechts sind die Namen so zu lesen: `PrimExWrapper`

ruft `PrimEx` und diese Funktion wiederum `SyncArrayList.get_Item`, was letztlich zu `get_Item` der Klasse `ArrayList` führt.

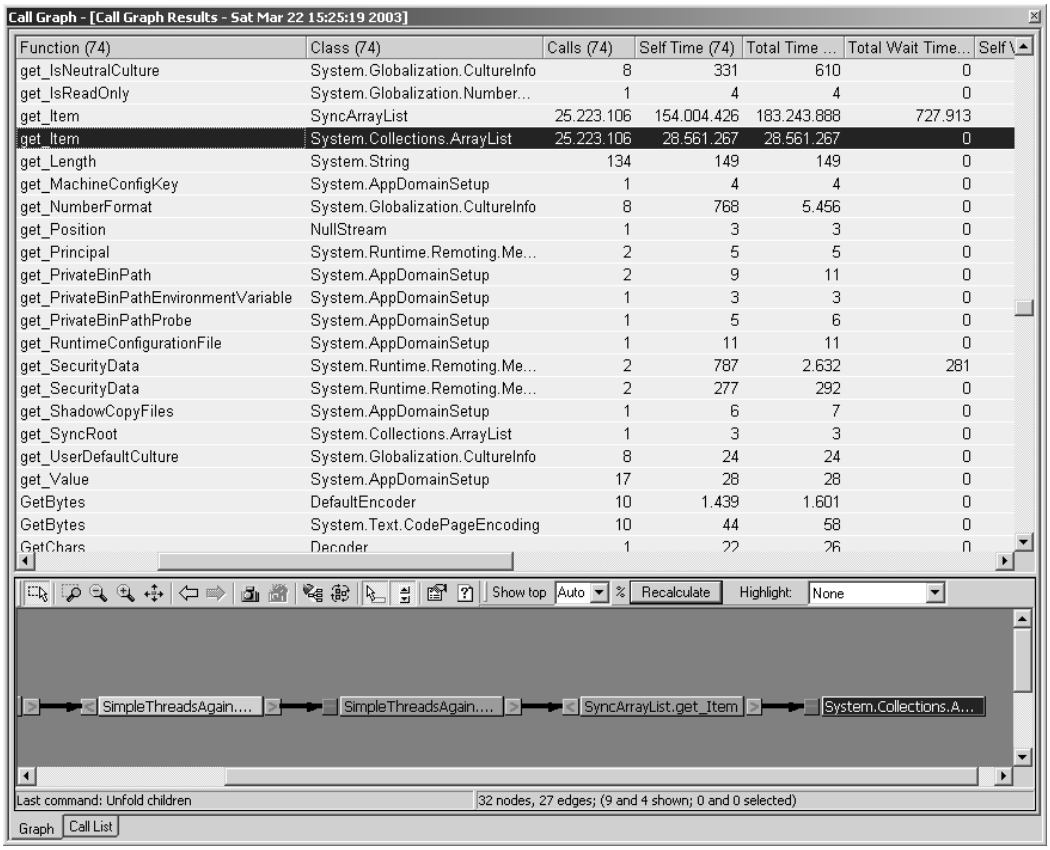

Abbildung 10.14: Der Funktionsgraph von `Simplethreadsagain.exe`

Die Wartezeit innerhalb von `get_Item` ist mit 0.7 von 183 Sekunden zu vernachlässigen. Die Synchronisation zwischen den beiden Threads ist gering, es kommt fast nie zu einer Wartezeit, obwohl die Funktion rund 25 Millionen Mal aufgerufen wird. Das Problem ist der Code der Synchronisation selbst. Der muss ja auch ausgeführt werden, wenn es zu gar keiner Wartezeit kommt. Am Unterschied der Self Time der `get_Item`-Funktion des Wrappers und von `ArrayList`, der bei 154 zu 27 Sekunden liegt, wird deutlich, wie teuer der Synchronisationscode ist. Die Iteration innerhalb der Liste war ja im C++-Code der mit Abstand aufwändigste Teil. Auch hier ist er sehr aufwändig, wird aber noch um den Faktor 5.7 übertroffen. Und dieser Faktor begrenzt gleichzeitig die Effizienz der parallelen Version des Programms auf maximal 17 %. Die tatsächlich erreichten 13 % (Abschnitt 2.5 *Die Parallelisierung einer Anwendung*) kommen dem schon nahe.

Der Profiler zeigt also ganz klar, was zuvor nur vermutet werden konnte. Der Aufwand für die Synchronisation ist hoch, auch wenn sie nur sehr selten tatsächlich benötigt wird, was die geringe Wartezeit andeutet. Auch eine effiziente Synchronisierung kann nicht schnell sein, wenn sie 25 Millionen Mal durchgeführt werden muss. Eine Verbesserung der Performance kann also nur darin bestehen, einen Container zu verwenden, der sehr viel seltener oder gar nicht synchronisiert werden muss.