**Профилировщик Quantify**

**Введение**

Quantify, вставляя отладочный код в бинарный текст тестируемого модуля, замеряет временные интервалы, которые прошли между предыдущим и текущими запусками. Полученная информация отображается в нескольких видах: табличном, графическом, комбинированном.

Статистическая информация от Quantify позволит узнать какие dll библиотеки участвовали в работе приложения, узнать список всех вызванных функций с их именами, формальными параметрами вызова и с статистическим анализатором, показывающим сколько каждая функция исполнялась.

Гибкая система фильтров Quantify позволяет, не загромождая экран лишними выводами (например, системными вызовами) позволяет получать необходимую информацию либо только о внутренних, программных вызовах либо только о внешних, либо комбинируя оба подхода.

Вооружившись полученной статистикой, разработчик без труда выявит узкие места в производительности тестируемого приложения и устранит их в кратчайшие сроки.

**Запуск приложений**

Рисунок 1 показывает действия после выбора «File->Run», в результате которого можно выбрать имя внешнего модуля и аргументы его вызова.

В качестве параметров настройки можно выбрать метод вставки отладочного кода:

Line. Наилучший способ вставки отладочного кода. Замеряется время исполнения каждой строки тестируемого приложения.

Function. То же самое, что и для «line», но с замером для времени исполнения вызываемых функций.

**Time.** Осуществляет сбор временной информации и преобразует ее вы машинные циклы.

По умолчанию Quantify собирает статистическую информацию в модуле тестируемого продукта и во всех внешних библиотеках.

Начало насыщения тестируемого приложения сопровождается появлением окна инструментирования, в котором построчно отображаются все модули, вызываемые основным. Данные модули, как говорилось выше, насыщаются отладочным кодом и помещаются в специальную директорию «cache» по адресу «\rational\quantify\cache». Отметим, что первоначальный запуск инструментирвания процесс длительный, но каждый последующий вызов сокращает общее время ожидания в силу того, что вся необходимая информация уже есть в Кеше.

С точки зрения дисковой емкости, файл (кэшируемый) с отладочной информацией от Quantify вдвое длиннее своего собрата без отладочной информации.

**Анализ информации**

**«Run Summary»**

Плавно переходим к следующей стадии тестирования, собственно, к сбору информации. По окончании процесса насыщения отладочным кодом модулей тестируемого приложения, Quantify переходит к его исполнению, производимым, обычным образом, за одним исключением: запись состояний тестируемого приложения продолжает проводиться в фоновом режиме.

Рисунок 3 демонстрирует фрагмент окна «Summary», в котором производится запись состояния тестируемого приложения. Причем, что очень примечательно, тестирование производится не только для простого приложения, но и для много поточного. В последнем случае (см. рисунок 3), тестируется каждый поток отдельно. В любом случае, даже если приложение однопоточное, то имя основного (единственного) потока именуется как «.main\_0», что представляется вполне логичным.

Информационный график постепенно наполняется квадратами разного цвета, демонстрирующими текущее состояние тестируемого приложения.

Отметим некоторые из них:

Running. Начало исполнения потока;

**Waiting I\O.** Ожидание действий по вводу\выводу;

**Blocked.** Блокирование исполнения потока;

**Quantify.** Ожидание вызова модуля Quantify;

**Exited.** Окончание исполнения потока.

Важный аспект при тестировании — получение статистической информации о количестве внешних библиотек, которые вызывало основное приложение, а также элементарное описание машины, на которой проводилось тестирование. Последнее особенно важно, так как бывает, что ошибка проявляется только на процессорах строго определенной серии или производителя. Все статистические аспекты решаются внутри окна «Summary».

Следующие два примера показывают статистическую информацию:

(1) Общий отчет — «Details»:

Program Name: C:\projects\aa\Debug\aa.exe

Program Arguments:

Working Directory: C:\projects\aa\Debug

User Name: Alex

Product Version: 2002.05.00 4113

Host Name: ALEX-GOLDER

Machine Type: Intel Pentium Pro Model 8 Stepping 10

# Processors: 1

Clock Rate: 847 MHz

O/S Version: Windows NT 5.1.2600

Physical Memory: 382 MBytes

PID: 0xfbc

Start Time: 24.04.2002 14:17:38

Stop Time: 24.04.2002 14:17:52

Elapsed Time: 13330 ms

# Measured Cycles: 191748 (0 ms)

# Timed Cycles: 2489329 (2 ms)

Dataset Size (bytes): 0x4a0001.

(2) «Log»

Quantify for Windows,

Copyright (C) 1993-2001 Rational Software Corporation All rights reserved.

Version 2002.05.00; Build: 4113;

WinNT 5.1 2600 Uniprocessor Free

Instrumenting:

Far.exe 620032 bytes

ADVAPI32.DLL 549888 bytes

ADVAPI32.DLL 549888 bytes

USER32.DLL 561152 bytes

USER32.DLL 561152 bytes

SHELL32.DLL 8322560 bytes

SHELL32.DLL 8322560 bytes

WINSPOOL.DRV 131584 bytes

WINSPOOL.DRV 131584 bytes

MPR.DLL 55808 bytes

MPR.DLL 55808 bytes

RPCRT4.DLL 463872 bytes

RPCRT4.DLL 463872 bytes

GDI32.DLL 250880 bytes

GDI32.DLL 250880 bytes

MSVCRT.DLL 322560 bytes

MSVCRT.DLL 322560 bytes

SHLWAPI.DLL 397824 bytes

SHLWAPI.DLL 397824 bytes

Для разработчика или тестера информация (информационный отчет), представленная выше, способна пролить свет на те статистические данные, которые сопровождали, а точнее, формировали среду тестирования.

**Дерево вызовов «Call Graph»**

Следующий интереснейший способ анализа конструкции приложения — это просмотр дерева вызовов. Окно, показанное на 4 рисунке, показывает только фрагмент окна с диаграммой вызовов.

Обратите внимание на количество и последовательность вызова различных модулей потока «main». Жирная линия показывает наиболее длительные ветви (содержащие либо часто вызываемые функции, либо функции, выполнявшиеся дольше остальных). Для демонстрации возможностей Quantify было сконструировано простое приложение, состоящее из функции «main» и двух дополнительных «recursive» и «outside» (см. листинг 1).

**Листинг 1.** Пример тестируемого приложения, сконструированном в виде консольного приложения из Visual Studio 6.0. Язык реализации «С».

#include "stdafx.h"

//Создаем функцию-заглушку

void outside (void)

{

static int v=0;

v++;

}

//Создаем рекурсивную функцию, исполняющуюся 100 раз

int recursive(void)

{

static int i=0;

int oo;

outside();//Вызываем функцию заглушку

if(i==100){i=1;return 0;}//Обнуляем счетчик и выходим

i++;

recursive();

}

int main(int argc, char\* argv[])

{

int i;

for(i=0;i<100;i++)recursive();

//Вызываем 100 раз рекурсивную функцию 100х100

return 0;

}

Приложение простое по сути, но очень содержательное, так как эффективно демонстрирует основные возможности Quantify.

В самом начале статьи мы выдвигали требование, по которому разработчикам не рекомендуется пользоваться рекурсивными функциями.

Тестеры или разработчики, увидев диаграмму вызовов, выделят функцию, находящуюся в полукруге, что является признаком рекурсивного вызова (см. рисунок).

В зависимости от того, на какой из ветвей дерева, находится курсор, выводится дополнительная статистическая информация о временном доступе к выделенной функции и к дочерним, идущим ниже.

Следующий рисунок демонстрирует статистику по функции «recursive».

Более подробно о статистике будет рассказано в следующем материале.

**Список вызовов функций «Function List»**

Одно из наиболее важных статистических окон. Здесь в табличном виде выводится статистическая информация по числу и времени работы каждой функции. Рисунок 6 демонстрирует окно с включенной сортировкой по числу вызовов каждой функции. В качестве дополнительной информации включен список формальных параметров вызовов функций. Подобную информацию можно получить только в том случае, когда тестируется модуль с отладочным кодом, к которому прилагается исходный текст.

Единицы измерения длительности работы функций могут быть следующими:

Микросекунды;

Миллисекунды;

Секунды;

Машинные циклы.

На рисунке приведены цифры соответствующие машинным циклам.

Полученная таблица вызовов анализируется тестером или разработчиком для выяснения узких мест в производительности приложения.

К сожалению, для принятия решения о производительности приложения, а точнее, производительности отдельных его функций можно принимать только рассматривая данный вопрос в комплексном разрезе. А именно, принимается во внимание и число вызовов каждой функции, и среднее время доступа к функции и общее время работы функции, и, наконец, то использовались ли при компиляции определенные специфические настройки компилятора.

Это комплексный подход, не предполагающий однозначного совета.

Сначала рассмотрим описание столбцов в появившейся таблице. Хотя многие из пунктов и являются интуитивно понятными, все же попробуем дать им короткое описание:

Function. Наименование функции. Можно высвечивать число и тип формальных параметров вызова данной функции.

**Calls.** Число вызовов. Величина абсолютная.

**Function Time.** Общее время исполнения всех вызовов данной функции

**Max F Time.** Максимальное время функции

**Module.** Полный путь до модуля с функцией (бинарного)

**Min F Time.** Минимальное время работы функции

**Source File.** Полный путь до исходных текстов модуля.

По любому из предложенных полей можно провести сортировку в прямом и обратном порядке, а также наложить фильтр на определенные модули, например, для проверки только внутренних модулей или только внешних.

Выделить из списка узкую функцию трудно, поскольку для правильного расчета нужно принимать во внимание и время работы функции и число вызовов. Причем, число вызовов не всегда может быть показателем медлительности функции (вызывается часто, а работает быстро).

Трудно давать какие либо советы по оптимизации кода, тем более, что в этой редакции мы не ставили перед собой подобных целей. По теории оптимизации написаны громадные труды, к которым мы с радостью и отправляем читателей.

Можно, конечно, дать общие рекомендации по улучшению производительности кода и его эффективности:

Использовать конструкцию «I++» вместо «I=I+1», так как компиляторы транслируют первую конструкцию в более эффективный код. К сожалению, этот эффективность примера ограничена настройками используемого компилятора, и иногда бывает равнозначной по быстродействию;

Использовать прием развертывания циклов. Такой же старый прием оптимизации работы относительно простых циклов. Эффект заключается в сокращении числа проверок условия счетчика, так при проверке выполняются очень медленные функции микропроцессора (функции перехода). То есть вместо кода:

For(i=0;i<100;i++)sr=sr+1;

Лучше писать:

For(i=0;i<100;i+=2)

{

sr++;

sr++;

}

Использовать тактику отказа от использования вычисляющих конструкций внутри простых циклов. То есть, если иметь подобный фрагмент кода:

for (sr = 0; sr < 1000; sr++)

{

a[sr] = x \* y;

}

то его лучше преобразовать в такой:

mn= x \* y;

for (sr = 0; sr < 1000; sr++)

{

a[sr] = mn;

}

поскольку мы избавляемся от лишней операции умножения в простом цикле;

Там где возможно при работе с многомерными массивами, обращаться с ними как с одномерными. То есть, если есть необходимость в копировании или инициализации, например, двумерного массива, то вместо кода:

int sr[400][400];

int j, i;

for (i = 0; i < 400; i++)

for (j = 0; j < 400; j++)

sr[j][i] = 0;

лучше использовать конструкцию, в которой нет вложенного цикла:

int sr[400][400];

int \*p = &sr[0][0];

for (int i = 0; i < 400\*400; i++)

p[sr] = 0; // или \*p++=0, **что для большинства компиляторов одно и тоже**

Также при работе с циклами выгодно использовать слияния, когда несколько коротких однотипных циклов сливаются вместе. Подобный подход также дает прирост в производительности кода;

В математических приложениях, требующих больших вычислений с плавающей точкой, или с большим количеством вызовов тригонометрических функций, удобно не производить все вычисления непосредственно, а использовать подготовленные значения в различных таблицах, обращаясь к ним как к индексам в массиве. Подход очень эффективен, но, к сожалению, как и многие эффективные подходы применим не всегда;

Короткие функции в классах лучше оформлять встроенными (inline);

В строковых операциях, в операциях копирования массивов лучше пользоваться не собственными функциями, а применять для этого стандартные библиотеки компилятора, так как эти функции, как правило, уже оптимизированы по быстродействию;

Использовать команды SSL и MMX, поскольку в достаточно большом круге задач они способны дать ускорение работы приложений в разы. Под такие задачи попадают задачи по работе с матрицами и векторами (арифметические операции над матрицами и векторами);

Использовать инструкции сдвига вместо умножений и делений там, где это позволяет делать логика программы. Например, инструкция S=S<<1 всегда эффективнее, чем S=S\*2;

Конечно, это далеко не полный список приемов оптимизации кода по производительности и качеству. Для этого есть масса других книг. Примеры здесь имеют чисто утилитарный подход: демонстрация возможностей Quantify в плане исследования временных характеристик кода.

Используя все средства сбора и отображения, разработчик постепенно сможет использовать только эффективные конструкции, что поднимет производительность на недосягаемую ранее высоту.

По любой функции можно вывести более детальный отчет (см. рисунок). Из него можно почерпнуть информацию о числе дочерних функций и то, откуда они были произведены. Следующий рисунок демонстрирует данную возможность.

Переход к просмотру исходного текста.

Если тестируемый модуль сопровождается исходным текстом, то в Quantify имеется возможность по переходу на уровень просмотра исходного текста. По контекстному меню можно осуществить данный переход. Вызывать функцию перехода имеет смысл только в том случае, когда Quantify работает в независимом режиме, в отрыве от среды разработки. Рисунок демонстрирует данный режим.

**Сравнивание запусков «Compare Runs»**

В большинстве случаев требуется иметь не только сведения об отдельных запусках, но и сравнения разных запусков в различных комбинациях для прогнозирования и анализа. Ведь всегда интересно знать быстро работает исправленная функция или медленно, по сравнению с тем, что было до этого.

Подобная аналитическая информация позволить иметь достаточно четкое представление о том находятся ли функции в прогрессирующем или в регрессирующем состоянии.

Для вызова модуля сравнения необходимо воспользоваться кнопкой (Compare Runs), выделив один из запусков, и указав на любой другой (каждый новый запуск отображается в истории запусков на левой части рабочего поля Quantify).

Для осуществления не пустого сравнения, в пример, рассмотренный выше, намеренно были внесены изменения, увеличившие число вызовов функций. Данные были сохранены и перекомпилированы и снова исполнены в Quantify. Результат представлен на рисунке:

Сравнение запусков позволяет проводить сравнительный анализ между базовым запуском (base – том, с которого все началось) и новым (new).

Результаты сравнения также остаются в проекте Quantify и сохраняются на протяжении жизненного цикла разработки проектов.

Наравне со сравнением запуск можно воспользоваться суммированием, нажав на кнопку . Эта функция вызывает простое суммирование чисел от двух запусков.

Можно сравнивать и складывать также запуски вместе со слепками (snapshot), которые позволяют оформить текущее количественное состояние в работе приложения в виде отдельного запуска. В дальнейшем над ним можно провести любую логическую операцию.

Ценность слепков проявляется тогда, когда необходимо узнать число вызовов каждой функции до свершения определенного события, например, до входа в определенный пункт меню в тестируемом приложении.

**API**

Это дополнительная возможность, предоставляемая Quantify по полному управлению процессом тестирования. API представляет собой набор функций, которые можно вызывать из тестируемого приложения по усмотрению разработчика.

Для получения доступа к API необходимо выполнить ряд действий по подключению «puri.h» файла с определением функций и с включением «pure\_api.c» файла в состав проекта. Единственное ограничение, накладываемое API — рекомендации по постановке точек останова после вызовов Quantify при исполнении приложения под отладчиком.

**Рассмотрим имеющиеся функции API Quantify:**

QuantifyAddAnnotation. Позволяет задавать словесное описание, сопровождающее тестирование кода. Информация, заданная разработчиком этой функцией может быть извлечена из пункта «details» меню тестирования и доступна в LOG-файле. На ее основе, тестер может впоследствии использовать особые условия тестирования;

**QuantifyClearData.** Очищает все несохраненные данные;

**QuantifyDisableRecordingData.** Запрещает дальнейшую запись;

**QuantifyIsRecordingData.** Возвращает значение 1 или 0 в зависимости от того производится ли запись свойств или нет;

**QuantifyIsRunning.** Возвращает значение 1 или 0 в зависимости от того проходит тестируемое приложение исполнение в обычном режиме или под Quantify;

**QuantifySaveData.** Данная функция позволяет сохранять текущее состояние — делать снимок (snapshot);

**QuantifySetThreadName.** Функция позволяет разработчикам именовать потоки в произвольном именном поле. По умолчанию Quantify дает имена, наподобие «thread\_1», что может не всегда положительно сказываться на читаемости получаемой информации;

**QuantifyStartRecordingData.** Начинает запись свойств. По умолчанию, данная функция автоматически вызывается Quantify при исполнении;

**QuantifyStopRecordingData.** Останавливает запись свойств.

Если модифицировать наше тестовое приложение так, чтобы оно использовало преимущества интерфейса API, то может получиться нечто нижеследующее

int main(int argc, char\* argv[])

{

int i;

QuantifyAddAnnotation("Тестирование проводится под Quantify с использованием API");

QuantifySetThreadName("Основной поток приложения");

for(i=0;i<12;i++){

QuantifySaveData();

recursive();

}

return 0;

}

В листинге показано как можно использовать основные функции API для извлечения максимального статистического набора данных.

Рисунки показывают слепки фрагментов экрана Quantify по окончании тестирования.

Рис. Данный рисунок демонстрирует вид окна после выполнения команды снятия слепка или вызова функции API QuantifySaveData()

Рис. Обратите внимание на поле аннотации

Рис. У Quantify отсутствуют сложности с русскими буквами

**Сохранение данных и экспорт**

Традиционные операции над файлами присущи и программе Quantify. Дополнительные особенности заключаются в том, что сохранять данные можно как во встроенном формате (qfy), для каждого отдельного запуска, так и в текстовом виде, для последующего использования в текстовых редакторах, либо для дальнейшей обработки скриптовыми языками типа perl (более подробно смотрите об этом в разделах по ClearCase и ClearQuest).

Quantify позволит переносить таблицы через буфер обмена в Microsoft Excel, что открывает безграничные возможности по множественному сравнению запусков, по построению графиков и различных форм. Все что необходимо сделать — это только скопировать данные из одной программы и поместить в другую известным способом.