САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Теоретический анализ и разработка методик оценки достоверности информации, получаемой современными профайлерами

Дипломная работа студента 545 группы

Булычева Ивана Дмитриевича

Научный руководитель	…… / подпись /	старший преподаватель Баклановский М.В.
Рецензент	…… / подпись /	к.ф.-м.н., доцент Булычев Д.Ю.
“Допустить к защите” заведующий кафедрой	…… / подпись /	д.ф.-м.н., проф. Терехов А.Н.

Санкт-Петербург
2011

SAINT PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

Theoretical analysis and development of techniques for estimating the reliability of information obtained by modern profilers

by
Ivan, Bulychev

Master's thesis

Supervisor	……	associate teacher M.V. Baklanovsky
Reviewer	……	associate professor D.U. Bulychev
"Approved by" Head of Department	……	professor A.N. Terekhov

Saint Petersburg
2011

Содержание

Содержание ... **3**

Вступление ... **6**

1. Обзор ... **8**

1.1. Определение профайлера ... 8

1.2. История профайлера ... 10

1.3. Многообразие профайлеров ... 11

1.4. Современные профайлеры ... 12

1.5. Принципы работы современных профайлеров ... 13

1.5.1. Семплирование ... 13

1.5.2. Инструментирование ... 14

1.6. Обзор современных профайлеров ... 15

1.6.1. Intel VTune performance analyzer ... 15

1.6.1.1. Описание ... 15

1.6.1.2. Принцип работы ... 16

1.6.2. AMD CodeAnalyst ... 18

1.6.3. AQTime ... 19

1.6.4. Метод подсчета инструкций ... 19

1.7. Используемые единицы измерений ... 20

2. Постановка задачи ... **22**

3. Описание разработанного инструмента ... **23**

3.1. Курсовая работа ... 23

3.1.1. Идея метода подсчета инструкций ... 23

3.1.2. Инициализация ... 24

3.1.3. Исполнение ... 25

3.2. Дальнейшая разработка ... 26

3.2.1. Временные оценки для отдельных инструкций ... 26

3.2.2. Расширен набор обрабатываемых инструкций ... 27

3.2.3. Внедрение и профайлинг ... 27

3.2.4. Подсчет времени работы приложения ... 28

4. Анализ ... **29**

4.1. Проблемы измерений ... 29

4.2. Причины погрешностей и особенностей в измерениях ... 30

4.3. Задачи, с которыми работает профайлер....31
5. Тестирование....32
5.1. Перечень проводимых экспериментов....32
5.2. Область применимости....32
5.2.1. Результат....32
5.2.2. Комментарии к результатам....32
5.3. Порядок проведения экспериментов....33
5.3.1. Автоматизация тестирования....34
5.4. Единые единицы измерений....35
5.4.1. AMD CodeAnalyst. Timer Samples в миллисекунды....35
5.4.2. Intel VTune. Unhalted cycles в миллисекунды....35
5.4.3. Калибровка метода подсчета инструкций....36
5.5. Эксперимент “Использование оперативной памяти”....36
5.5.1. Использование ”нагрузки”....37
5.5.2. Минимальное использование оперативной памяти....37
5.5.3. Активное использование оперативной памяти....37
5.5.4. Использование оперативной памяти с частыми промахами кеша....38
5.6. Эксперимент “Предсказания ветвлений”....38
5.6.1. Эксперимент “Вариация длины цикла”....38
5.6.2. Эксперимент “Периодически выполняемые операции”....39
5.7. Эксперимент “Потеря контекста”....39
5.8. Эксперимент “Выравнивание данных”....39
5.9. Эксперимент “Вызовы процедур”....40
6. Результаты....41
6.1. Комментарии к результатам тестирования....41
6.1.1. Эксперимент “Использование оперативной памяти”....41
6.1.2. Эксперимент “Предсказания ветвлений”....41
6.1.3. Эксперимент “Потеря контекста”....42
6.1.4. Эксперимент “Выравнивание данных”....42
6.1.5. Эксперимент “Вызовы процедур”....43
6.2. Современные методы профайлинга....43
6.2.1. Выявленные ошибки....43
6.2.2. Выявленные недостатки....43
6.2.3. Сравнение профайлеров между собой....44
6.2.4. Проблемы, возникшие при тестировании....44

6.3. Преимущества метода подсчета инструкций 45
6.4. Рекомендации по использованию профайлеров 46
7. Выводы 48
8. Заключение 49
9. Глоссарий 51
10. Литература 52
Приложение 56
Приложение 1. “Минимальное использование оперативной памяти” 56
Приложение 2. “Активное использование оперативной памяти” 57
Приложение 3. “Использование оперативной памяти с частыми промахами кеша” 58
Приложение 4. “Вариация длины цикла” 60
Приложение 5. “Периодически выполняемые операции” 61
Приложение 6. “Вынужденная потеря контекста” 62
Приложение 7. “Выравнивание данных” 63
Приложение 8. “Вызовы процедур” 64
Приложение 9. “Результаты тестирования” 65

Вступление

В этой работе будут рассматриваться, тестироваться и анализироваться современные инструменты профайлинга приложений. Данная работа является продолжением работы [40], в которой предпринималась попытка разработать альтернативный метод тестирования производительности программ.

В настоящее время можно наблюдать повышение интереса к производительности программ. Из-за того, что предельная тактовая частота процессора почти достигнута и совершенствование процессора совершается путем внедрения новых технологий оптимизации исполнения и простым экстенсивным путем, много внимания уделяется эффективности исполнения программы. Поэтому профайлеры и другие инструменты оптимизации приложений становятся все более и более востребованными. Очень важно уметь классифицировать и сравнивать их между собой.

В Главе 1 “Обзор” будет приведен исторический обзор этапов развития средств профайлинга, обзор наиболее известных и широко используемых в настоящее время инструментов. Также будут подробно рассмотрены основные методы профайлинга и прокомментирована обоснованность выбора профайлерами тех или иных единиц измерений.

В Главе 3 “Описание разработанного инструмента” представлено краткое описание утилиты, разработанной в рамках курсовой работы, и ряд доработок, которые были внесены в рамках дипломной работы.

В Главе 4 “Анализ” описаны некоторые размышления на тему возможности точных и идеальных измерений, а также задач, которые должны решаться профайлером.

В Главе 5 “Тестирование” будут представлены тесты, на которых осуществлялось тестирование профайлеров. Также описаны проблемы

тестирования и их решения. К таким проблемам можно отнести автоматизацию тестирования, перевод единиц, случайные шумы в показаниях.

В Главе 6 “Результаты” перечислены некоторые значимые выводы, полученные из результатов тестирования профайлеров. Также вынесены некоторые оценки профайлерам, которые оценивались по качеству, удобству использования и универсальности.

В Главе 7 “Выводы” представлены главные выводы, которые были сделаны после проделанной работы.

Все материалы доступны по адресу:

http://willzyx-edu-project.googlecode.com/svn/trunk/Proj2

1. Обзор

1.1. Определение профайлера

Под профайлером в данной работе мы будем понимать инструмент для оценки производительности программ и планирования работ по ее оптимизации. Это может быть как фон-Неймановская [18], так и Гарвардская архитектура [38]. Там программа рассматривается как последовательность команд, которая может содержать такие структуры как циклы, условные переходы, процедуры и другие. Из-за этих структур команды в программе исполняются неодинаковое число раз. Почти в любой программе можно найти некоторую последовательность команд, время и частота исполнения которых значительно выше, чем у других участков в программе. Такие участки будем называть “горячими” – на их исполнение тратится сравнительно больше тех или иных ресурсов компьютера. Очевидно, если удастся заменить этот “горячий” участок программы на аналогичный, результат выполнения которого тот же, но выполняющийся быстрее, то производительность программы возрастет значительно больше, чем при попытках замены “холодных” участков.

Главная задача профайлера – поиск таких “горячих” участков в программе. Ниже перечислены некоторые из ресурсов, расход которых способны измерять многие современные профайлеры:

- количество тактов процессора,
- количество выполненных инструкций,
- количество промахов кеша,
- количество неверно угаданных переходов,
- количество обращений к кешу,
- объем используемой памяти,
- и многие другие.

Анализ информации о “нагретости” исполняемого кода помогает в изучении и анализе программы, планировании работ по оптимизации [19]. Большие и сложные программы изучаются быстрее и качественнее. Разработчик также может сравнивать результаты профайлинга с ожиданиями для соответствующих участков в программе и, тем самым, выявлять целый класс ошибок.

Для полноценного анализа приложений необходимо производить замеры по всем ключевым и используемым ресурсам. Только так возможно достижение наилучшего быстродействия. Если использовать профайлер в процессе разработки программы, то, например, возможно своевременно избежать архитектурных ошибок, связанных с использованием памяти. Аналогично анализируются прочие измерения. Большинство из них имеет смысл производить на стадии прототипирования и разработки.

Часто профайлер выступают как экономически выгодный подход к оптимизации программ. Многие профессиональные профайлеры являются платными системами, и их цена колеблется в больших интервалах [13]. Эта цена определяется, главным образом, набором предлагаемых средств по оптимизации приложений, интерактивности и удобству использования. Многие предприятия готовы тратить немалые средства для повышения эффективности разрабатываемых ими систем.

В некоторых ситуациях использование профайлера становится необходимостью. Например, довольно распространена ситуация, когда необходимо повысить эффективность системы при отсутствии людей хоть как-нибудь разбирающихся в системе. Без профайлера эта задача становится крайне сложной и практически невыполнимой. Поэтому предприятия ставятся перед выбором – приобрести профайлер, который удовлетворит всем потребностям разработчика, или набрать более дешевых профайлеров, которым придется обучать работников.

1.2. История профайлера

Первые полноценные системы для профайлинга начали появляться в начале 70-х годов. Для платформ IBM/360 и IBM/370 были созданы средства, которые использовали прерывания программы по таймеру для сохранения их состояния [9]. Эти средства профайлинга можно назвать одними из первых, которые можно было отнести к методу семплирования.

Позже были созданы эмуляторы инструкций процессора, которые трассируют машинные команды и сами их исполняют. Так программа полностью находится под контролем профайлера, и становится возможно отслеживание всех событий, происходящих в ней. SIMON, SIMMON, OLIVER – некоторые из тех профайлеров.

В конце 70-х – начале 80-х в операционной системе Unix появляются инструменты и расширения к компиляторам, которые инструментируют программу еще при компиляции. К таким можно отнести профайлеры prof, gprof.

Следующим этапом развития профайлера стало инструментирование скомпилированного бинарного кода. Первым из таких инструментов был ATOM [4], который появился в 1994 году. Он умел уже работать с объектными модулями.

В настоящее время с появлением байт-кода и множества виртуальных машин, таких как JVM, CLR, создаются инструменты профайлинга приложений, которые на них работают. В данном случае все решения схожи между собой, т.к. каждая из таких виртуальных машин предоставляет специальные интерфейсы для профайлинга. К таким инструментам относятся jTracert, CHESS.

Большинство из упомянутых подходов находят применение в профайлинге и по сей день. Единственно устаревший вариант – эмуляция исполнения программы. Тут не достаточно имитировать только процессор. Для качественного анализа нужно имитировать всю систему вплоть до

контроллеров, оперативной памяти и внешних устройств. Из-за многообразия систем и архитектур, процессоров, которые становятся с каждым годом все более сложными, очевидно, что создать такой эмулятор почти невозможно.

1.3. Многообразие профайлеров

Из-за разнообразия [43] языков программирования, платформ, операционных систем, компиляторов и интерпретаторов в настоящее время существуют и продолжают появляться множество различных профайлеров. Все их можно разбить на классы по следующим признакам:

- Статистические. Большинство из них используют метод семплирования. Профайлер работает уже скомпилированной программы.
- Инструментирование. В исполняемый код тем или иным способом добавляются дополнительные команды, которые занимаются сбором информации.
- Симуляция. Программа фактически не исполняется на процессоре. Его заменяет специальная программа – эмулятор.

Ниже приведена таблица с представителями каждого из классов:

Статистические	
Intel VTune [17]	ориентирован на работу на процессорах Intel, ОС – Windows и Linux
AMD CodeAnalyst [2]	ориентирован на работу на процессорах AMD, ОС – Windows и Linux
AQTime [5]	ОС Windows, любой процессор
Shark [30]	ОС MacOS
Paraller Amplifier [12]	дополнительный инструмент для Microsoft Visual Studio, профайлинг многопоточных приложений
Инструментальные	
Gprof [44] , Quantify [34]	дополнения к GCC
ATOM [4]	статическое инструментирование

Pin [32]	динамическое инструментирование, ОС Windows, Linux, MacOS
Valgrind [35] [1]	фреймворк для инструментов динамического анализа, ОС Linux и MacOS
DynInst [7]	динамическое инструментирование, мультиплатформенная библиотека
Dmalloc [6]	библиотека языка Си, отслеживает выделение и освобождение памяти
Turbo Profiler	устаревший профайлер компании Borland, который был разработан как дополнение для продуктов Turbo Pascal, Turbo C, Turbo Assembler. ОС DOS
Эмуляция	
SIMMON, SIMON, OLIVER	устаревшие симуляторы инструкций на архитектуре IBM System 360/370/390

1.4. Современные профайлеры

Многие из созданных когда-либо профайлеров могут использоваться и в настоящее время, т.к. до сих пор производится поддержка того многочисленного ПО, которое было написано десятилетия назад. Поэтому профайлер умрет окончательно только тогда, когда соответствующее программное обеспечение выйдет из использования. В любом случае, популярность таких профайлеров неуклонно падает.

В данной работе будут рассматриваться инструменты, которые активно используются и в настоящее время, а программное обеспечение, которое они способны тестировать, по-прежнему выпускается. Также наложим еще одно ограничение на профайлер: инструмент должен уметь считать или оценивать время исполнения программы или ее участка. Профайлеры, которые работают только с памятью, процессорным кешом или другими показателями, которые не являются оценкой времени, в этой работе рассматриваться не будут.

Далее будет рассмотрен ряд систем, которые в настоящее время пользуются наибольшей популярностью. К таким средствам можно отнести:

- Intel VTune – набор инструментальных средств, разработанный компанией Intel. Некоторая часть функциональности работает только на процессорах этой компании. Более подробное описание в соответствующем разделе.
- AMD CodeAnalyst – продукт компании AMD. Также как и для Intel VTune, полный набор функциональности можно использовать лишь на процессорах компании AMD.
- AQTime – универсальный инструмент, работающий на процессорах обеих фирм.

1.5. Принципы работы современных профайлеров

Все перечисленные профайлеры имеют схожие принципы работы. Основные методы:

- семплирование – анализ программы на основе периодических замеров без изменения исполняемого кода,
- инструментирование – здесь в исходные или машинные коды внедряются дополнительные команды, которые и производят необходимые замеры (времени, счетчиков процессора и т.д.).

1.5.1. Семплирование

Суть метода семплирования заключается в том, что профайлер делает замеры системных счетчиков программы через постоянный интервал времени с использованием системных прерываний. Изменения исполняемого кода при этом не происходит. Так как этот метод является статистическим, то сильных погрешностей в измерениях не происходит. Семплированием можно искать участки программы, которые больше всего процессорного времени тратят на свое исполнение, которые больше всего подвержены кеш-промахам и ошибкам предсказаний ветвлений.

Этот метод применяется в большинстве коммерческих инструментов тестирования производительности. Он прост в реализации, универсален. Его используют для первичного анализа программы, чтобы определить возможные аспекты оптимизации ее кода.

Проблема, с которой часто сталкиваются современные профайлеры – многообразие процессоров. Не существует стандартов, которые определяют их архитектуру. Это сказалось и на встроенных средствах профилирования – специальных счетчиках, которые считают количество тактов процессора, кеш-промахи и т.п. Для разных архитектур и для разных процессоров набор счетчиков и способ доступа к ним совершенно различны. Поэтому и возникает несовместимость между, например, процессором Intel и профайлером AMD CodeAnalyst, когда большинство счетчиков недоступны для семплирования.

1.5.2. Инструментирование

Представляет собой широкий класс приемов профайлинга. Метод инструментирования применяется в программирования для:

- осуществления мониторинга и измерения уровня производительности приложения,
- отладки и диагностики ошибок,
- записи трассировочной информации.

Суть метода заключается в том, что в исходный или скомпилированный код вставляются дополнительные команды, которые, в свою очередь собирают информацию о ходе исполнения программы. В отличие от метода семплирования, внедрение инструкций может вносить значительную погрешность в измерение производительности.

Можно классифицировать множество способов внедрения в процесс исполнения приложения следующим образом:

1. Статическое инструментирование

a. добавление дополнительных команд в программу при компиляции,
b. добавление команд в скомпилированные объектные файлы,
c. добавление команд в исполняемые модули,

2. Динамическое инструментирование
   a. перехват вызовов импортируемых и экспортируемых функций,
   b. анализ всей программы по мере ее исполнения.

1.6. Обзор современных профайлеров

Прежде чем тестировать профайлеры, необходимо их описать, перечислить их функциональность и ограничить область их применимости.

1.6.1. Intel VTune performance analyzer

Intel VTune performance analyzer представляет из себя набор инструментов профилирования, который позволяет:

- строить дерево вызовов,
- производить статистический анализ кода (семплирование),
- просматривать измерения на строчках исходного кода и на дизассемблированных инструкциях скомпилированного кода,
- оптимизировать приложение в соответствии с рекомендациями, которые будет давать Intel VTune.

1.6.1.1. Описание

Intel VTune performance analyser является профессиональным платным коммерческим продуктом. Также компания Intel выпускает такие средства тестирования производительности, как Intel Thread Profiler, Intel Parallel Studio, специализирующиеся на тестировании и анализе активности исполняемых потоков. Все эти инструменты взаимно дополняют друг друга и составляют систему, которая не имеет аналогов в мире.

1.6.1.2. Принцип работы

В своих измерениях Intel VTune использует методы семплирования и инструментирования.

Сам Intel VTune performance analyzer предлагает сбор трех видов информации:

1. Collect sampling data,
2. Collect counter monitor data,
3. Collect call graph data.

Collect sampling data.

Как упоминалось ранее, большинство массово выпускаемых на данный момент процессоров снабжаются встроенными средствами тестирования производительности. Наиболее важные и используемые из этих средств – встроенные счетчики.

Сейчас существуют десятки типов встроенных счетчиков, которые умеют подсчитывать:

- количество тактов процессора,
- количество промахов и попаданий в кеш первого и второго уровней,
- количество выполненных инструкций,
- количество верно и ошибочно предсказанных ветвлений,
- количество прерываний,
- другие.

Полный список событий, с которыми работают счетчики процессоров фирмы Intel, можно смотреть тут [14].

Существует два способа использования счетчиков [15]:

1. простой подсчет событий,
2. семплирование – процессор генерирует прерывание, когда показатель счетчика превышает определенное значение (при этом счетчик сбрасывается).

Настраивать и считывать счетчики производительности можно только в режиме ядра, поэтому Intel VTune, используя собственные драйвера, настраивает нужные счетчики, исходя из настроек профилирования. При возникновении прерывания, профайлер обрабатывает его, считывает все необходимые показания, анализирует контекст потока, в котором произошло прерывание. Собрав достаточное количество семплов, VTune производит их анализ: находит функции и участки кода, исполнение которых происходит чаще всего, или те участки, которые чаще всего вызывают соответствующие события в процессоре (промах кеша, ошибка предсказания ветвления и прочее).

Collect counter monitor data.

Этот вид сбора данных использует счетчики по второму их назначению, а именно, обычный подсчет событий в системе.

Такой способ не подходит для вычисления производительности самой программы. Он предназначен для оценки работы программы внутри системы: оценивается, насколько эффективно используется процессор, не простаивают ли процессы в ожидании тех или иных ресурсов и т.д.

Collect call graph data.

Этот вид сбора статистики основан на инструментировании скомпилированного кода программы. При этом используется информация из файла с отладочными данными или информация из таблицы импорта для того, чтобы локализовать положение каждой функции в скомпилированной программе. Вычислив необходимые адреса, VTune производит замену первых байт каждой функции инструкцией безусловного перехода. Поэтому, когда будет вызываться та или иная функция, тут же будет осуществлен переход в код динамической библиотеки VTune, которая предварительно загружается в память профилируемого процесса.

Для полноценного инструментирования программы, построения дерева вызовов и подсчета необходимой статистики, необходимо иметь файл

отладочной информации. Таким образом, инструментирование производится статически, во время подготовки процесса к профайлингу. Без отладочного файла дерево вызовов не будет отображать большую часть информации, которая присуща профилируемой программе.

1.6.2. AMD CodeAnalyst

AMD CodeAnalyst – фирменный профайлер компании AMD. Он ориентирован на платформу x86 и разрабатывается для операционных систем Windows и Linux. Является бесплатным.

Позволяет производить следующие виды анализа [31]:

- Access performance – первоначальный анализ приложения, чтобы определить дальнейшие направления для анализа,
- Investigate L2 cache access – подробный анализ использования кеша второго уровня,
- Investigate branching – подробный анализ использования модуля предсказания ветвлений,
- Investigate data access – анализ использования памяти,
- Investigate instruction access – анализ процесса чтения и исполнения инструкций,
- Thread profile – анализ многопоточности и параллельного использования многоядерной или многопроцессорной системы.

Каждый из видов профайлинга использует метод семплирования. После анализа выводится сводная таблица – количество семплов соответствующего события на каждый процесс. В дальнейшем можно детально рассмотреть те участки программы, которые больше всего вызывали те или иные события-семплы. Смотреть можно как исходный код, так и ассемблерный листинг.

1.6.3. AQTime

AQTime – профайлер компании AutomatedQA с проприетарной лицензией. Он работает на операционной системе Windows. Умеет встраиваться в популярные среды разработки такие как Microsoft Visual Studio, Borland Developer Studio, Embarcadero RAD Studio. Стоимость – от 600$.

Профайлер поддерживает следующие режимы:

- Performance Profiler – считает время работы приложения и ее частей, единственный интересующий нас режим,
- Platform Compliance – определяет совместимость программы с операционными системами,
- Static Analysys – различные статистики программы, вычисляемые с использованием отладочной информации,
- Load Library Tracer – поиск используемых динамических библиотек,
- многие другие, ориентированные на dot.net приложения.

Режим «Performance Profiler». Этот режим профайлинга позволяет вычислять время работы, как исполняемых модулей, так и отдельных функций. Профайлер предлагает следующие единицы измерений: секунды, миллисекунды, микросекунды, циклы процессора. Вместе с этим для каждой функции считается количество их вызовов.

1.6.4. Метод подсчета инструкций

В рамках этой работы был разработан альтернативный подход к анализу производительности приложений. Здесь не будет производиться попытки точного вычисления времени работы программы или ее части. Вместо этого метод анализирует все выполненные инструкции. Если для каждой инструкции оценить количество тактов, которое тратится на ее исполнение, то возможно посчитать и количество тактов, необходимые для

работы целого приложения. Поделив получившуюся величину на тактовую частоту процессора, можно оценить время работы уже в секундах.

Более подробное описание инструмента будет приведено в Главе 3. “Описание инструмента”.

1.7. Используемые единицы измерений

Важно отметить некоторые особенности у выбираемых профайлерами единиц измерений.

Если вернуться к системам, использующим процессоры i8086, i8088, i186, i286, i386, i486, то для них использовались единицы Clock [39]. Для всех инструкций было известно количество клоков, которое требовалось для их исполнения. Просуммировав эти значения для всех исполненных инструкций программы, можно тривиальным образом перевести значение в секунды и получить время исполнение программы в секундах.

В настоящее время секунды и клоки не так широко распространены как раньше. Вместо этого используется множество других единиц, которые, зачастую, далеки от секунд и не переводимы однозначно. Почему произошли такие изменения? Причины этому многократное усовершенствование процессора:

- конвейерная обработка команд,
- кеши различных уровней,
- предсказания ветвлений,
- технологии Hyper-threading и Super-threading.

Все эти изменения направлены на повышение производительности процессоров. С другой стороны, теперь время исполнения, как инструкции, так и целой программы не фиксировано и может сильно варьироваться, если использовать в качестве единиц измерений секунды или пропорциональные им единицы. Это объясняет многообразие используемых в настоящее время единиц: профайлеры используют те единицы, которые им удобно использовать, или те, которые получают используемые ими методы.

Если возникает необходимость измерить время работы программы именно в секундах, то зачастую приходится проделать дополнительную работу по калибровке профайлера. После этого можно переводить получаемые значения в секунды. Также не стоит забывать о погрешности, которая естественным образом возникнет при переводе единиц, а именно из-за того, что нет строгой зависимости между единицами.

Если раньше измерение времени работы программы можно было производить “секундомером”, то сейчас данный подход подвержен сильным погрешностям, и применять его следует с осторожностью. Вместо этого современные профайлеры используют другие методы, которые оперируют отличными от секунд единицами.

2. Постановка задачи

Главная задача этой работы – проверить насколько точны измерения, производимые современными профайлерами. Одновременно с этим будет происходить их сравнение с разработанным инструментом, который можно по праву назвать профайлером.

Итак, первая задача, которую нужно выполнить, – доработать инструмент таким образом, чтобы он выдавал свою оценку времени работы приложения. Это нужно для того, чтобы иметь возможность сравнить его показания с показаниями других современных промышленных профайлеров.

Вторая задача – разработать способ сравнения между собой качества результатов для различных профайлеров. Разработанный способ должен быть применим к большинству из существующих профайлеров.

Третья задача – используя разработанный метод сравнить следующие профайлеры: Intel VTune, AMD CodeAnalyst, AQTime, разработанный метод подсчета инструкций. В результате должна быть представлена оценка каждому из профайлеров и предложены рекомендации по их использованию.

3. Описание разработанного инструмента

3.1. Курсовая работа

Ранее в рамках курсовой работы была создана динамическая библиотека, которую можно было подключить к приложению, и с помощью экспортируемых из нее функций управлять процессом профайлинга – начать и остановить профайлинг, сохранить статистику в файл.

3.1.1. Идея метода подсчета инструкций

Суть метода заключается в том, что исполняемый код во время выполнения будет анализироваться и перестраиваться налету. Для этого необходимо выполнить шаги, указанные ниже.

При инициализации процесса профайлинга:

- устанавливается глобальный обработчик исключений – сплайсинг [29] функции KiUserExceptionDispatcher;
- все секции кода выбранного исполняемого модуля будут заблокированы – соответствующим страницам памяти будет выставлены права NO_ACCESS [22].

Процесс исполнения программы:

1. попытка исполнения инструкций из недоступных страниц памяти будет вызывать исключение Access Violation;
2. исключение перехватывается прежде, чем оно достигает пользовательских и стандартных обработчиков;
3. сформированная при возникновении исключения структура данных содержит информацию об адресе возникновения исключения и ее типе;
4. зная адрес возникновения исключения, определяем, какой код требуется выполнить;

5. анализируем код вплоть до переходов, которые нельзя вычислить (для их определения нужно знать содержимое регистров);
6. строим аналогичный проанализированному код в специально выделенной памяти;
7. продолжаем исполнение программы, но уже перестроенного участка;
8. при переходе по абсолютному адресу (все такие переходы вычисляемые) исполнение вернется в недоступные секции кода;
9. возвращаемся к шагу 1.

3.1.2. Инициализация

Инициализация процесса профайлинга приложений состоит из двух шагов: установка глобального обработчика исключений и блокировка секций кода.

Все исключения в системе инициализируются в ядре операционной системы. Причины их инициализации может быть две: исключение инициализировано процессором (например, деление на ноль или обращение к недоступной части памяти) или самой операционной системой (например, функция RaiseException [24]). При этом каждое исключение обрабатывается ядром операционной системы, и некоторая часть из них даже не доходит до обработчиков в самой программе. Для исключений, которые остались необработанными, в стеке на уровне приложений формируется фрейм, который содержит полную информацию об исключении. Затем управление передается функции KiUserExceptionDispatcher из системной динамической библиотеки ntdll.dll, которая загружена в память к каждому приложению. Далее исключение обрабатывается одним из механизмов – Structured Exception Handling (SEH [26]) или Vectored Exception Handling (VEH [27]). Дальнейшая судьба исключений нас не очень интересует. Главное, мы знаем процедуру, через которую проходит каждое исключение, обрабатываемое на пользовательском уровне, следовательно, можно их все контролировать.

Второй шаг – блокировка секций кода загруженного в память PE файла. Для этого нужно решить следующие задачи:

1. найти, по каким адресам был загружен исполняемый модуль,
2. найти адреса секций PE файла, которые содержат исполняемый код,
3. применить функцию VirtualProtect.

Первая задача решается использованием функций CreateToolhelp32Snapshot, Module32First и Module32Next. С помощью них можно перебрать все исполняемые модули, загруженные в память, и выбрать подходящий. Зная формат PE-файлов [42] можно решить вторую задачу. Третья задача решается тривиальным образом.

3.1.3. Исполнение

Во время исполнения динамически будет строиться новый код, который и будет исполняться вместо оригинального.

Любое обращение к заблокированным секциям кода будет вызывать исключение Access Violation. Перехватив его, определяем адрес возникновения исключения – это и будет адрес инструкции, с которой следует начать анализ. Следует заметить, что чтение памяти из заблокированных секций запрещено, поэтому предварительно необходимо сделать копии всех секций кода.

Анализ производится последовательно, просматривается инструкция за инструкцией. Анализ заканчивается тогда, когда достигается инструкция перехода или инструкция уже анализировалась ранее. Если достигается инструкция перехода по относительному адресу (такой адрес фиксирован и не зависит от значений в регистрах), то пытаемся проанализировать код, что находится по адресу перехода.

Одновременно с анализом будет строиться новый исполняемый код. Если встречается инструкция, которая нарушает последовательное исполнение команд, например, инструкции перехода или инструкции с

префиксом REP, то их необходимо обрабатывать специальным образом, а именно, добавлять дополнительные управляющие инструкции. Если же инструкция обычная и не нарушает последовательность исполнения, то она просто копируется.

Так анализируется весь код до переходов по абсолютным адресам. Такие переходы зависят от значений регистров, поэтому продолжение анализа невозможно. После переноса таких инструкций в сгенерированный код, адреса переходов не изменятся, поэтому переходы будут происходить снова в заблокированную секцию.

3.2. Дальнейшая разработка

Эта утилита была доработана до приложения, которому в качестве параметра при запуске указывается исполняемый файл, подвергаемый профайлингу. Файл запускается под управлением профайлера. После окончания работы создается текстовый файл, содержащий статистическую информацию о процессе исполнения. Также в этом файле содержится два числа – общее число выполненных инструкций и временную оценку работы приложения. Эти временные оценки и будут сравниваться с показателями популярных профайлеров.

3.2.1. Временные оценки для отдельных инструкций

Компания Intel в своих документациях [10] приводит таблицу оценок времени исполнения инструкций в циклах процессора. В таблицах указаны следующие оценки:

- Latency – количество циклов процессора, которое требуется для исполнения всех μops, из которых формируется инструкция,
- Throughput – количество циклов процессора, которое ему нужно ожидать, чтобы приступить к выполнению инструкции того же типа.

Эти оценки приблизительны и зачастую на порядки отличаются от истины, о чем в документации также упоминается. Поэтому результирующие показания разработанной утилиты также могут сильно отличаться от реальных.

3.2.2. Расширен набор обрабатываемых инструкций

Ключевая часть разрабатываемой утилиты – дизассемблер длин инструкций [41].

Ранее использовался модуль, выложенный на форуме сайта `rsdn.ru` [33]. Он умел обрабатывать только инструкции из набора x86. Для многих программ этого оказывается недостаточно: часто в обычных программах встречаются инструкции из набора MMX и SSE – это малая часть всех расширений набора инструкций, которые широко распространены.

Поэтому был задействован дизассемблер длин Catchy32 [36]. MMX, SSE, SSE2, 3DNow! – наборы инструкций, которые поддерживаются новым дизассемблером. В большинстве случаев этого оказывается вполне достаточным.

3.2.3. Внедрение и профайлинг

Утилита, разработанная в рамках курсовой работы, представляла собой динамическую библиотеку, которую тестируемые программы должны были сами подключать. Такой подход не всегда применим из-за того, что приходится вносить изменения в исходный код программы.

Динамическая библиотека была дополнена программой, которой в качестве параметров передаются имя исполняемого файла, который необходимо протестировать, и соответствующие параметры запуска. Программа производит следующие шаги:

1. производит запуск соответствующего файла с указанными параметрами; в функции CreateProcess [20] указывается параметр

CREATE_SUSPENDED [23] –главный поток приложения после создания будет остановлен;

2. выделяет память в тестируемой программе с помощью функции VirtualAllocEx [28];
3. записывает в выделенную память команды, необходимые для
    a. загрузки динамической библиотеки в память,
    b. инициализации процесса профайлинга;
4. для исполнения этих команд создает поток с помощью функции CreateRemoteThread [21];
5. возобновляет работу главного потока с помощью функции ResumeThread [25].

3.2.4. Подсчет времени работы приложения

При вычислении времени работы, как уже упоминалось ранее, будут использоваться оценки для отдельных инструкций, представленных в документации фирмы Intel [10].

Оценки для отдельных инструкций подсчитываются во время анализа исполняемого кода и заносятся в специальный ассоциативный массив. После окончания исполнения программы просматривается каждый из ее участков. Зная количество раз, которое они были исполнены, и, умножив на вычисленный вес входящих в них инструкций, получим искомую величину.

4. Анализ

4.1. Проблемы измерений

Первый вопрос, который возникает при оптимизации программных продуктов, – при каких конфигурациях компьютера осуществлять профайлинг приложения. Тип и модель процессора, размер кеша, даже объем оперативной памяти и наличие файла подкачки – все это может влиять на показания профайлера. Оптимизированная на одном компьютере программа может не быть оптимальной на другом.

Другая частая проблема для большинства инструментов измерений – любое измерение вносит погрешность в саму измеряемую величину. Это относится и к профайлерам – при измерениях им приходится тем или иным образом вторгаться в процесс исполнения программы. И, конечно, это влияет на показания профайлера. Различные методы используют разнообразные способы внедрения: некоторые из них делают свои специальные вставки в программу, некоторые внедряются в систему и используют ее возможности. Но профайлер это такая же программа. Она работает в этой же системе, и ее код должен периодически выполняться для произведения необходимых измерений и их обработки.

Третья проблема процесса профайлинга – многообразие единиц измерения времени. Зачастую различные профайлеры оперируют различными единицами при выводе результатов. К таким единицам можно отнести:

- секунды,
- такты процессора,
- количество семплов,
- количество выполненных инструкций.

И, как правило, у каждых из них есть свои существенные недостатки. Секунды и такты процессора – равносильные единицы, потому что одни

можно однозначно перевести в другие для конкретного процессора. Эти единицы были распространены ранее, но сейчас после значительного увеличения сложности процессоров качественно измерить время исполнения программы в этих единицах становится крайне проблематично. Количество семплов – единицы, которые получаются на выходе у статистических методов. Таким методам свойственно получать приблизительные показания. Количество выполненных инструкций – одни из немногих единиц, которые можно подсчитать с абсолютной точностью. Проблема состоит в том, что из полученных показаний трудно делать выводы, если в конечном итоге стремимся к оптимизации секунд.

4.2. Причины погрешностей и особенностей в измерениях

Все эти проблемы профайлинга происходят из-за того, что в современных процессорах исполнение команд это очень сложный процесс. Частично он описан в многотомных документациях [11], но значительная доля все же остается известна только разработчикам процессоров и другим высококвалифицированным специалистам.

На время исполнения участков программ влияют следующие факторы:

- кеши первого и второго уровней процессора,
- предсказания ветвлений,
- конвейерное выполнение инструкций,
- технологии Hyper-threading, Super-threading и прочие,
- параллельно запущенные потоки и приложения.

С каждой новой версией процессора в нем реализуются всё новые технологии, добавляются новые методы оптимизации инструкций. Наличие всех этих факторов не позволит процессору исполнять один и даже тот же участок программы всегда одно и то же время.

4.3. Задачи, с которыми работает профайлер

В этом пункте будет приведен список задач, с которыми приходится работать современным средствам тестирования производительности программ, учитывая контекст этой работы. Профайлеры вычисляют:

- время исполнения всей программы,
- время исполнения определенной процедуры,
- время исполнения произвольного участка кода,
- время, потраченное на выполнение команд определенного исполняемого модуля,
- время исполнения определенной инструкции.

А также полезные функции, наличие которых приветствуется в профайлере:

- возможность тестирования кода на платформе, отличной от платформы, на которой код был скомпилирован,
- возможность тестирования кода на виртуальных машинах,
- пакетное (batch) тестирование.

5. Тестирование

5.1. Перечень проводимых экспериментов

Сначала будут проведены тесты, которые определят область применимости выделенных профайлеров. Каждый профайлер будет вручную запускаться в различных условиях, чтобы понять, где тестирование производить можно, а где нет.

После этого каждым из профайлеров будут протестированы ряд специально подобранных программ-тестов с различными параметрами. В частности, у тестов будет варьироваться активность использования оперативной памяти и кешей, а также нагрузка прочих программ на систему. Основное назначение этих тестов – максимизировать погрешность, которая вносится профайлерами.

5.2. Область применимости

Проведено первичное тестирование профайлеров, которое определяло область применимости каждого из них.

5.2.1. Результат

Ниже приведена таблица результатов тестирования области применимости профайлеров:

	Intel VTune	**AMD CodeAnalyst**	**AQTime**	**Метод подсчета инструкций**
Процессоры Intel	Все функции доступны	Функциональность частично ограничена	Все функции доступны	Функциональность не зависит от версии и модели процессора
Процессоры AMD	Профайлинг недоступен	Все функции доступны	Все функции доступны	
Единицы измерения производительности	unhalted cycles	CPU clocks, Timer samples	секунды	сумма взвешенных инструкций

5.2.2. Комментарии к результатам

Как упоминалось ранее, профайлер Intel VTune не работает корректно на процессорах фирмы AMD. Действительно, проведя запуск на

элементарной программе, можно в этом убедиться: при любых конфигурациях профайлер выдает сообщение об ошибке «The CPU architecture can't be identified properly; data collection is not available» (осуществлялась проверка версии 9.1 на процессоре AMD Turion 64 TL-60).

И наоборот, профайлер AMD CodeAnalyst на процессорах семейства Intel имеет ограниченную функциональность [8]. Большая часть функциональности недоступна. Одна из функций, которая продолжает работать это Time-Based Sampling. Этого нам будет достаточно, чтобы осуществлять тестирование.

Рассмотрим единицы измерения, которыми оперируют соответствующие профайлеры. AQTime в этом отношении выглядят более привлекательно, чем Intel VTune и AMD CodeAnalyst, потому что заявляемые секунды это более естественная величина, чем CPU clock и unhalted cycles. Естественные величины – секунды или другие единицы, строго пропорциональные им, хороши тем, что именно их стремятся уменьшить в процессе оптимизации приложения.

5.3. Порядок проведения экспериментов

Для тестирования профайлеров будет создан набор тестов. Тест представляет собой программу, которая в качестве параметра принимает одно число. Профайлер будет запускаться на этой программе, при этом, параметр будет варьироваться в некотором интервале с некоторым шагом. Запуски производятся в пакетном режиме для оптимизации процесса проведения тестов. Сбор информации осуществляется таким образом, чтобы в дальнейшем ее можно было вставить в таблицу Excel.

Каждый эксперимент проводится несколько раз. Это нужно для того, чтобы для каждой неровности в графике знать, является ли это некоторым “шумом” многозадачной системы или какой-то особенностью работы процессора, памяти, операционной системы или других факторов.

Многократные замеры и усреднения позволят сгладить получаемые графики измерений.

5.3.1. Автоматизация тестирования

Ни один из рассматриваемых профайлеров не поддерживает пакетное тестирование программы для изменяющегося параметра через GUI. Так же этого напрямую не позволяют делать предлагаемые ими средства для командной строки. Но изменение параметра можно организовать средствами стандартного пакетного bat файла.

AMD CodeAnalyst. Для автоматизации тестирования этого профайлера была создана утилита, которая последовательно запускает три утилиты из его поставки: `caprofile`, `cadataanalyze`, `careport`. Они проводят профайлинг, анализируют полученные данные и строят отчет соответственно. Далее из отчета извлекается необходимое число – время работы приложения и выводится в стандартный поток. Все остальные действия делаются с помощью команд в bat файле.

Intel VTune. Для этого профайлера организовать автоматизацию было несколько проще, чем для предыдущего. Достаточно использовать две команды: “`vtl activity`” и “`vtl view`”. Для построенного отчета используется специальный скрипт для извлечения данных: извлечение сводится к поиску строк, начинающихся с определенной последовательности символов.

AQTime. Этот профайлер не имеет средств для серьезных автоматизаций. Разрешен лишь запуск заранее сконфигурированного проекта. При этом изменять параметр запуска программы не представляется возможным. Поэтому параметр будет передаваться через файл, и перед запуском этот файл будет обновляться. Запуск производится командой “`aqtime project.aqt /r /silentmode /e`”, где `project.aqt` – имя файла проекта. Множественные запуски можно организовать с помощью пакетного файла. Результаты будут сохранены в файле проекта. Далее

необходимо выполнить две операции в GUI под названием Merge (объединение результатов нескольких запусков) и Compare (сравнение результатов тестирования для нескольких запусков). В конечном итоге получится таблица, содержащая в себе результаты для всех запусков. Далее их можно копировать в буфер обмена и вставлять в электронную таблицу.

5.4. Единые единицы измерений

Помимо того, что мы будем анализировать результаты запусков отдельно для каждого профайлера, важно, в конечном итоге, попытаться сравнить их между собой. Понятно, что сравнение запусков должно происходить на одном компьютере.

Проблема, с которой мы столкнемся, – разные единицы измерений для различных профайлеров. Поэтому необходимо научиться переводить одни единицы в другие.

5.4.1. AMD CodeAnalyst. Timer Samples в миллисекунды

Из документации [3] можно узнать, что при Timer-Based профайлинге профайлером делаются замеры состояния системы через определенный интервал времени. По умолчанию, используя команду “`caprofile /s`”, замеры будут производиться каждые 100мкс. То есть секунда равна 10000 timer samples.

5.4.2. Intel VTune. Unhalted cycles в миллисекунды

Профайлер Intel VTune измеряет производительность методом семплирования. Основные единицы измерения профайлера Intel VTune это количество семплов для событий CPU_CLK_UNHALTED.CORE и INST_RETIRED.ANY – количество unhalted циклов процессора и количество выполненных инструкций соответственно [16]. Работать будем с первым из них.

Часто в работе процессора возникают ситуации, когда ему приходится ожидать некоторые данные извне. Это может быть, например, чтение данных

от внешних устройств. В это время процессор совершает холостые циклы. Профайлер Intel VTune не учитывает такие циклы, хотя реальное время при этом продолжает идти.

В связи с этим, калибровка будет проходить следующим образом. Мы напишем простую тестовую программу, которая будет работать исключительно с процессором для того, чтобы минимизировать количество холостых циклов. Затем измерим реальное время работы программы и показания профайлера. Поделив друг на друга эти числа, мы получим коэффициент перевода одних единиц в другие.

На самом деле этот коэффициент зависит от тактовой частоты процессора и от количества unhalted циклов на семпл. Первая величина постоянна для процессора, и ее нетрудно узнать. Вторая настраивается непосредственно в профайлере.

5.4.3. Калибровка метода подсчета инструкций

Так как для каждой инструкции статически определяется количество тактов, которое тратится на ее исполнение, то можно просуммировать эти значения для каждой выполненной инструкции и получить общее количество тактов, которое потребуется для выполнения программы. Такое количество, конечно, является оценочным. Далее эту величину можно поделить на тактовую частоту процессора и получить уже оценочное время работы программы.

5.5. Эксперимент “Использование оперативной памяти”

Проведем серию экспериментов, в которых попытаемся разнообразить использование оперативной памяти. Будет три типа тестов:

1. оперативная память минимально используется,
2. оперативная память активно используется,
3. неоптимальное использование оперативной памяти (многочисленные промахи кеша памяти).

Программы, которые будут представлены в виде тестов, должны иметь возможность быть сконфигурированными двумя параметрами. Первый параметр – целое число, при произведении серии тестов, варьироваться будет именно этот параметр. Второй параметр – также целое число, но оно будет изменяться только для различных профайлеров, чтобы в некоторых случаях скомпенсировать накладные расходы – замедление работы программы, которые могут быть значительными.

5.5.1. Использование ”нагрузки”

Одновременно с обычными измерениями будем производить измерения с параллельно работающей “нагрузкой” – программой, которая активно использует оперативную память и процессор. Такие измерения необходимы для того, чтобы симулировать систему, на которой полноценно работают другие приложения.

5.5.2. Минимальное использование оперативной памяти

Данный набор тестов должен показать, как соответствующие профайлеры измеряют время простейшей программы, которая должна производить некоторые вычисления, не использующие оперативную память. Это нужно, чтобы избежать некоторых погрешностей, которые могут ей вызываться.

В качестве тестируемой программы можно выбрать программу, которой в качестве параметра передается некоторое число, и которая должна посчитать и вывести сумму от единицы до этого числа. Код программы представлен в Приложении 1 “Минимальное использование оперативной памяти”.

5.5.3. Активное использование оперативной памяти

Здесь мы будем тестировать программу, которая активно использует оперативную память.

Тестируемая программа создаст массив, который заполнится числами. Потом эти числа будут складываться в некоторой переменной. Сложение будет происходить несколько раз, количество определяется вторым постоянным параметром. Код программы представлен в Приложении 2 “Активное использование оперативной памяти”.

5.5.4. Использование оперативной памяти с частыми промахами кеша

В этой программе, как и в предыдущем тесте, будем создавать массив и производить сложение элементов, предварительно заполнив его произвольными числами. Только в этом тесте массив должен быть двумерным, и сложение будет происходить поперек выделенной памяти (внутренний цикл перебирает строчки в массиве). В этом случае промахи кеша будут происходить настолько часто, насколько это возможно. Код программы представлен в Приложении 3 “Использование оперативной памяти с частыми промахами кеша”.

5.6. Эксперимент “Предсказания ветвлений”

Вторая серия экспериментов будет относиться к предсказаниям ветвлений. Они, как и кеш памяти, вносят определенную неоднородность в процесс исполнения машинного кода.

5.6.1. Эксперимент “Вариация длины цикла”

Суть программы-теста будет такая: фиксируются N операций и разделяются на M групп по K операций ($M * K = N$). Программа состоит из двух вложенных циклов: внутренний цикл – K итераций, внешний – M итераций. Одна итерация вложенного цикла – одна операция.

Зафиксировав N операций и варьируя K, можно наблюдать некоторые неоднородности во времени исполнения теста. Совершенно очевидно, что они вызваны конвейерностью обработки инструкций. Ключевая инструкция в этой программе – инструкция условного перехода внутреннего цикла.

Именно на ней процессор чаще всего обращается к предсказателю ветвлений. Для очень маленьких значений *K* процессор будет знать, что, скорее всего, повторять цикл не придется. Для больших *K* – скорее всего, цикл повторится. Код программы представлен в Приложении 4 “Вариация длины цикла”.

5.6.2. Эксперимент “Периодически выполняемые операции”

В этом тесте мы упростим подход и будем просто выполнять некоторые действия периодически, в нашем случае действия будут выполняться, когда некоторый счетчик будет делиться на некоторое число *K*, которое будет варьироваться. Поэтому программа структурно будет содержать лишь цикл и одно условие. Код программы представлен в Приложении 5 “Периодически выполняемые операции”.

5.7. Эксперимент “Потеря контекста”

Этот эксперимент предназначен для того, чтобы обнаружить некоторые из особенностей метода семплирования. Т.к. нам известно, что суть этого метода в периодических замерах, то против этого и будем работать.

В качестве входных данных тесту передается одно число – оно будет определять количество операций между принудительными потерями контекста. Потеря контекста будет осуществляться посредством API функции Sleep с параметром 1. Количество итераций “Операции-Sleep” фиксировано. Код программы представлен в Приложении 6 “Вынужденная потеря контекста”.

5.8. Эксперимент “Выравнивание данных”

Еще одна из проблем производительности, которая может возникать в программе – ошибка выравнивания данных [37]. Некоторые из профайлеров отслеживают такого рода проблемы.

Этот тест похож на тест “4.5.2. Активное использование оперативной памяти”. Отличие заключается в том, что четырехбайтные значения для сложения будут извлекаться из массива с шагом один байт.

Код программы представлен в Приложении 7 “Выравнивание данных”.

5.9. Эксперимент “Вызовы процедур”

Этот эксперимент должен выявлять некоторые из случаев непосредственного внедрения вспомогательных инструкций в исполняющуюся программу. Конкретно, отследить инструментирование вызовов процедур тестируемой программы.

Тест содержит в себе процедуру, которая принимает четырехбайтное число, производит с ним некоторые простые операции и возвращает получившийся результат – также четырехбайтное число. Во время исполнения этого теста процедура исполняется большое количество раз. Если перед исполнением каждой процедуры происходит выполнение вспомогательных инструкций, то исполнение программы должно несколько замедлиться. Код программы представлен в Приложении 8 “Вызовы процедур”.

6. Результаты

6.1. Комментарии к результатам тестирования

Ссылка на полную таблицу результатов можно найти в Приложении 9 “Результаты тестирования”.

6.1.1. Эксперимент “Использование оперативной памяти”

Первые три теста были призваны определить, как использование программой памяти влияет на показания профайлера. Во всех тестах алгоритм имеет линейную сложность.

В случае, когда оперативная память не используется, все популярные профайлеры показывают одинаковые результаты в независимости от того, работает ли параллельно нагрузка или нет. Графики функций представляют собой прямую линию.

В двух последних тестах, когда память активно используется, характер графиков функций для популярных профайлеров уже не является прямой линией – на показания начинает влиять кеш памяти. При этом параллельно работающая нагрузка увеличивает показания профайлеров в несколько раз. Без нагрузки показания профайлеров Intel VTune и AMD CodeAnalyst совпадают с реальным временем работы программы.

Результаты метода подсчета инструкций во всех случаях растут линейно и не зависят от использования нагрузки. Отличаются от реального времени не больше чем в три раза.

6.1.2. Эксперимент “Предсказания ветвлений”

Этот эксперимент был призван показать, как сильно может различаться процесс исполнения программ для различных процессоров.

В обоих тестах показания профайлеров Intel VTune и AMD CodeAnalyst почти совпадают с реальным временем работы приложения. Результаты

AQTime для процессоров AMD несколько отклонены от этих верных значений.

Наиболее нагляден тест “Периодически выполняемые операции”. Из постановки теста следует, что чем меньше значение переданного параметра, тем чаще выполняется некоторая операция, и, казалось бы, время выполнения программы должно быть больше. Но для процессоров Intel характер зависимости времени от параметра имеет совершенно противоположный вид: для меньших значений параметра время работы программы меньше. Метод подсчета инструкций показывает более ожидаемые результаты (Рис 1). Для процессора AMD характер функции реального времени, в общем, ожидаемый, но для очень малых значении (меньше 10) график функции испытывает резкий рост (Рис 2).

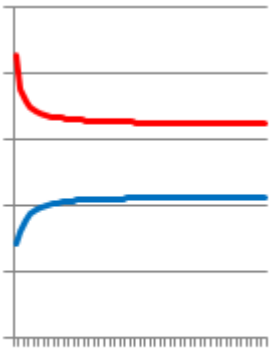

Рис 1

Рис 2

6.1.3. Эксперимент “Потеря контекста”

Тест достиг поставленной цели – выявить одну из главных проблем метода семплирования. График результатов имеет ступенчатый вид. Объясняется это тем, что значительная часть семплов не успевает попасть в необходимый процесс.

6.1.4. Эксперимент “Выравнивание данных”

Тест показал, что ошибка выравнивания данных никак не влияет на погрешность рассматриваемых профайлеров. Все результаты, как и ожидалось, линейно зависят от параметра и совпадают с реальным временем работы программы.

6.1.5. Эксперимент “Вызовы процедур”

Тест показал, что профайлер AQTime неспособен справиться с поставленной задачей. Результаты значительно превышают реальные значения. Характер графика функции представляет собой случайную ломаную линию (Рис 3). Время профайлинга возрастает в 300 раз.

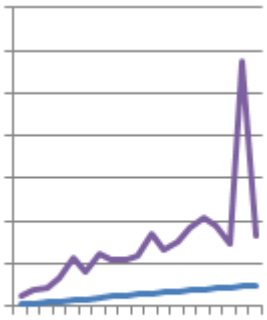

Рис 3

6.2. Современные методы профайлинга

После проведенных тестов можно выделить некоторые проблемы и ошибки, которые происходят при профайлинге приложений.

6.2.1. Выявленные ошибки

Единственная ошибка, которая была допущена одним из исследуемых профайлеров это попытка добавить вспомогательный исполняемый код в тестируемую программу. Если при этом продолжать делать замеры времени стандартными способами, то это неизбежно влечет за собой появление погрешностей в измеряемых значениях. В настоящее время не существует способов, которые помогут избежать этой проблемы.

6.2.2. Выявленные недостатки

Первая из проблем, которая была выявлена набором тестов ”5.5. Использование оперативной памяти”, – возникновение погрешностей при активном использовании памяти. Если при профайлинге приложения параллельно запускается другое приложение, использующее оперативную память, то показания профайлера несколько отличаются от того времени, которое требуется программе для работы в “чистой” системе.

Вторая проблема, выявленная набором тестов “5.6. Предсказания ветвлений” – различие процессов выполнения программ на процессорах различных архитектур. Тестирование производилось на процессорах фирмы AMD и Intel. На графиках видны значительные отличия в поведении

функции времени исполнения программы от входного параметра. Из этого можно сделать вывод, что для различных процессоров оптимальные программы могут отличаться.

Третья проблема, которая была выявлена, связана с применением метода семплирования. Метод производит периодические замеры состояния программы, а затем, собрав все эти данные, находит “горячие” участки в ней. Из-за того, что метод статистический, то возможен случай, когда семплы не будут покрывать весь исполняемый код. В тесте ”5.7. Потеря контекста” для профайлера AMD CodeAnalyst был выявлен случай, когда график времени работы программы от параметра имеет ступенчатый вид. Это объясняется тем, что для несильно отличающихся параметров семпл не успевает попасть в нужный процесс из-за принудительной потери контекста.

6.2.3. Сравнение профайлеров между собой

Если использовать профайлер как инструмент для измерения времени работы приложения, то, оценив все полученные результаты, можно сказать, что профайлер AMD CodeAnalyst проявил себя лучше всех. От профайлера компании Intel его отличает лишь то, что он работает на процессорах обеих фирм. Intel VTune же специализирован только на процессорах Intel.

AQTime оценивается как самый худший из рассматриваемых фирменных профайлеров. В его показаниях всегда присутствует некоторое постоянное отклонение от реального времени исполнения приложений. В тесте “5.9. Вызовы процедур” из-за ошибки в способе измерения, профайлер давал на выход неадекватные показания, которые значительно отличаются от реального времени работы приложения.

6.2.4. Проблемы, возникшие при тестировании

Каждый из рассматриваемых профайлеров не был приспособлен к такому роду тестирования. Для каждого инструмента пришлось создавать небольшие утилиты или скрипты для того, чтобы пакетные запуски стали

возможны. В нашем случае при обработке тестов необходимо перебирать некоторое количество чисел и передавать их в качестве параметров.

Профайлер AQTime и в этом случае проявил себя не с лучшей стороны, не предоставив полноценных консольных средств для работы с ним. Из командной строки возможно осуществлять только запуск уже настроенной конфигурации. При этом отсутствует экспорт полученных результатов в обычный текстовый файл.

AMD CodeAnalyst и Intel VTune имеют достаточно средств для работы с ними через командную строку. Для полной оптимизации процесса тестирования было достаточно написать скрипты для извлечения необходимых данных из сгенерированных отчетов.

6.3. Преимущества метода подсчета инструкций

Ниже перечислены преимущества метода подсчета инструкций, выявленные после проведения всех тестов:

1. независимость показаний от архитектуры компьютера, версии и модели процессора,
2. независимость показаний от внутренних оптимизаций процессора (кеширование, предсказания ветвлений и другие),
3. постоянные показания, независящие от окружения, параллельных процессов и нагрузки на систему,
4. результирующие показания пропорциональны реальному времени работы программы,
5. высокая точность производимых измерений.

За все такие преимущества приходится платить замедлением работы программы в среднем в 50-100 раз в зависимости от плотности ветвлений и переходов в программе.

6.4. Рекомендации по использованию профайлеров

Во время проведения тестов был подтвержден тот факт, что измерения профайлеров не могут быть точными. Часто при оптимизации приложений с использованием профайлеров рекомендуют:

- производить многократные запуски, чтобы исключить возможность случайных ошибок в измерениях;
- тестировать на различных процессорах и конфигурациях, чтобы получить оптимальную на большинстве компьютеров программу.

В процессе выполнения работы эти утверждения были полностью подтверждены.

Теперь рассмотрим выбранные в работе профайлеры: Intel VTune, AMD CodeAnalyst, AQTime. Профайлеры фирм Intel и AMD для большинства тестов показывают почти одинаковые результаты. Но профайлер Intel VTune не работает на процессорах своего конкурента, поэтому для решения нашей проблемы его можно не использовать. Тут важно отметить, что ценность профайлера фирмы Intel заключается в другой его функциональности, связанной с анализом кода и рекомендациями по его оптимизации.

При использовании профайлера AQTime следует быть очень осторожным, т.к. было показано, что в его измерениях часто присутствует отклонение от реальных значений. В отдельных случаях это отклонение бывает очень значительным. Поэтому рекомендуется дополнительно применять другие инструменты для проверки результатов профайлинга.

Для всех трех профайлеров можно сказать, что они являются промышленными, т.к. все они в настоящее время широко используются. И для всех них свойственен анализ программы в контексте той системы, в которой работает профайлер. Разработанный метод подсчета инструкций позволяет производить анализ вне такого контекста, т.е. его результаты не зависят от той системы, в которой работает программа. В частности, на показания никак не влияют многочисленные оптимизации процессора,

операции ввода-вывода, обращение к оперативной памяти и другие. Иногда бывают случаи, что из-за таких факторов теряется часть “горячих” точек: либо они становятся холоднее из-за оптимизаций процессора, либо появляется много других точек, которые несколько “горячее” рассматриваемых. В дальнейшем, при изменении характеристик окружения “горячие” точки могут проявляться. Поэтому иногда бывает полезно произвести анализ программы, не учитывая все эти факторы.

Метод подсчета инструкций позволяет находить те участки программы, которые выполняются чаще других. В этой работе было показано, что в некоторых случаях ни один из промышленных профайлеров не может справиться с этой задачей. В вычислениях метода используются веса, предварительно подсчитанные для каждой инструкции, поэтому возможно сравнение времени для различных участков программ, например, двух процедур. Так как на показания не влияют естественные задержки, связанные с операциями ввода-вывода и оптимизациями процессора, то можно сказать, что осуществляется профайлинг вычислительной составляющей программы.

Из всего сказанного можно сделать вывод, что разработанный профайлер стоит рассматривать как дополнительный инструмент поиска “горячих” точек в приложении, и его следует использовать совместно с другими профайлерами.

7. Выводы

После проделанных тестов утверждение о том, что измерения профайлера никогда не бывают точными, полностью подтвердилось. Действительно:

- все проделанные измерения для профайлеров имеют собственную погрешность,
- используемые методы не идеальны, и возможно подобрать программу, для которой они работают неверно,
- показания профайлера могут качественно отличаться для архитектур различных процессоров.

Впрочем, измерения, получающиеся после работы современных профайлеров, вполне удовлетворяют потребностям разработчиков, и обнаруженные погрешности в большинстве случаев настолько незначительны, что их просто оставляют без внимания.

Если рассмотреть все характеристики показаний разработанного метода подсчета инструкций, то можно сказать, что получаемая оценка – метрика на множестве программ. Эту метрику можно использовать для сравнения производительности различных приложений.

Подобно тому, как при оценке времени работы алгоритма в О-нотации отбрасывается константа, так и оценка, вычисленная разработанным инструментом, не учитывает влияние системы.

8. Заключение

Предложенный в работе метод можно использовать гораздо шире. Так можно оценивать время исполнения не только программы, скомпилированной под платформу x86, а, например, время работы программ на виртуальных машинах или на интерпретаторах. Разработка таких программ сейчас довольно распространена из-за высокой скорости их разработки. Для проведения профайлинга достаточно собрать достаточную статистику для каждой инструкции байт-кода или другой элементарной операции: например, сколько тактов процессора в среднем тратится на исполнение той или иной команды.

Разработанный в рамках дипломной и курсовой работы инструмент имеет свои сильные и слабые стороны, как и остальные разработанные на данный момент профайлеры. Многие из них используют предоставляемую процессорами функциональность для профайлинга. Добавление этой функциональности было вынужденной мерой. Без нее осуществлять быстрый и качественный профайлинг было невозможно. Также в этой работе было доказано, что производить точные измерения времени работы приложения также невозможно по причине сложности архитектуры процессора и самого процесса исполнения программ.

Как видим, существенную роль в профайлинге занимает центральный процессор. Из этого можно сделать предположение, что в будущем профайлингом приложений будет заниматься целиком только сам процессор. Например, будет создан специальный режим работы процессора, в котором измерения не будут влиять на процесс исполнения самой программы. За это придется заплатить некоторым снижением скорости работы программы.

Еще один способ получения идеальных измерений – создание программы-эмулятора, точной копии процессора. Такой эмулятор позволит анализировать код программы настолько всесторонне, насколько это

возможно. Но такой подход крайне сложен в реализации и, скорее всего, создан не будет никогда.

9. Глоссарий

1. Инструментирование – внедрение в исходные или машинные коды дополнительных команд, которые производят необходимые замеры (времени, счетчиков процессора и т.д.).
2. Профайлинг приложения – измерение количества ресурсов потребляемой той или иной программой и поиск мест, в которых потребление больше чем в остальных.
3. Семплирование – анализ программы на основе периодических замеров.
4. Сплайсинг функции – замена первых пяти байт функции инструкцией безусловного перехода. Необходим для перехвата вызовов этой функции [29].

10. Литература

1. Alexey Ott, “Что такое valgrind и зачем он нужен”. http://www.alexott.net/ru/linux/valgrind/Valgrind.html;
2. “AMD CodeAnalyst Performance Analyzer. Product page”. http://developer.amd.com/cpu/CodeAnalyst/Pages/default.aspx;
3. AMD Developer Central, CodeAnalyst, “Time-Based Profiling”. http://developer.amd.com/cpu/CodeAnalyst/codeanalystlinux/Documents/CodeAnalyst-Linux-help/analysis_TBP.htm;
4. Amitabh Srivastava, Alan Eustace, “Atom: A system for building customized program analysis tools”, 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.119.8540;
5. “AQtime Pro. Product page”. http://smartbear.com/products/development-tools/performance-profiling/;
6. “Dmalloc – Debug Malloc Library”. http://dmalloc.com/;
7. “Dyninst. An Application Program Interface (API) for Runtime Code Generation”. http://www.dyninst.org/;
8. “How I ran AMD CodeAnalyst on an Intel CPU”. http://www.virtualdub.org/blog/pivot/entry.php?id=288;
9. “IBM S/360 System Calls”. http://www.cs.clemson.edu/~mark/syscall/s360.html;
10. Intel® 64 and IA-32 Architectures, Optimization Reference Manual, Appendix C, “Instruction latency and throughput”;
11. “Intel® 64 and IA-32 Architectures Software Developer's Manuals”. http://www.intel.com/products/processor/manuals/;
12. “Intel® Parallel Amplifier 2011. Product page”. http://software.intel.com/en-us/articles/intel-parallel-amplifier/;
13. “Intel® Parallel Studio XE 2011 – Purchase”. http://software.intel.com/en-us/articles/intel-parallel-studio-xe-purchase/;

14. Intel Software Developer’s Manual, 3 том, Приложение A, “Performance-monitoring events”;
15. Intel Software Developer’s Manual, 3 том, глава 30, 30.8. ”Performance Monitoring (Processors based on Intel NetBurst® Microarchitecture)”, стр 328.
16. Intel® Software Network, “Using Intel® VTune™ Performance Analyzer Events/ Ratios & Optimizing Applications”. http://software.intel.com/en-us/articles/using-intel-vtune-performance-analyzer-events-ratios-optimizing-applications/;
17. “Intel® VTune™ Amplifier XE 2011. Product page”. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/;
18. John von Neumann, "First Draft of a Report on the EDVAC". http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf;
19. Maxim Chirkov, “Для чего нужно профилирование”. http://www.opennet.ru/docs/RUS/gprof/gprof-1.html;
20. MSDN, “CreateProcess Function”. http://msdn.microsoft.com/en-us/library/ms682425(v=vs.85).aspx;
21. MSDN, “CreateRemoteThread Function”. http://msdn.microsoft.com/en-us/library/ms682437(v=vs.85).aspx;
22. MSDN, “Memory Protection Constants”. http://msdn.microsoft.com/en-us/library/aa366786(v=vs.85).aspx;
23. MSDN, “Process Creation Flags”. http://msdn.microsoft.com/en-us/library/ms684863(v=vs.85).aspx;
24. MSDN, “RaiseException Function”. http://msdn.microsoft.com/en-us/library/ms680552(v=vs.85).aspx;
25. MSDN, “ResumeThread Function”. http://msdn.microsoft.com/en-us/library/ms685086(v=vs.85).aspx;
26. MSDN, “Structured Exception Handling”. http://msdn.microsoft.com/en-us/library/ms680657(v=vs.85).aspx;

27. MSDN, “Vectored Exception Handling”. http://msdn.microsoft.com/en-us/library/ms681420(v=vs.85).aspx;
28. MSDN, “VirtualAllocEx Function”. http://msdn.microsoft.com/en-us/library/aa366890(v=vs.85).aspx;
29. Ms-Rem, “Перехват API функций в Windows NT (часть 1). Основы перехвата”, 2005 год. http://www.wasm.ru/article.php?article=apihook_1;
30. “Optimizing your Application with Shark 4”. http://developer.apple.com/tools/sharkoptimize.html;
31. Paul J. Drongowski, “Increased performance with AMD CodeAnalyst™ software and Instruction-Based Sampling”. http://developer.amd.com/Assets/amd_ca_linux_june_2008.pdf;
32. “Pin – A Dynamic Binary Instrumentation Tool”. http://www.pintool.org/;
33. RSDN, “Дизассемблер длин инструкций (x86)”. http://www.rsdn.ru/forum/src/3120789.1.aspx;
34. Steve Mansour, “Profile code with Quantify 1.1”, 1994. http://portal.acm.org/citation.cfm?id=188277;
35. “Valgrind. Product site”. http://www.valgrind.org/;
36. VX Heavens, “Catchy32”. http://vx.netlux.org/vx.php?id=ec11;
37. Wikipedia, “Data structure alignment”. http://en.wikipedia.org/wiki/Data_structure_alignment;
38. Wikipedia, “Harvard Architecture”. http://en.wikipedia.org/wiki/Harvard_architecture;
39. Zack Smith, “The Intel 8086 / 8088/ 80186 / 80286 / 80386 / 80486 Instruction Set”. http://home.comcast.net/~fbui/intel.html;
40. Булычев Иван, ”Разработка метода сбора информации о ходе исполнения программы, который использует возможность модификации памяти процесса”.

http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/doc/coursework.pdf;
41. Википедия, “Дизассемблер длин”. http://ru.wikipedia.org/wiki/Дизассемблер_длин;
42. Гумеров Максим, “Загрузчик PE-файлов”. http://www.rsdn.ru/article/baseserv/peloader.xml;
43. “Деревья эволюции операционных систем и языков программирования”. http://shannon.usu.edu.ru/history/trees.htm;
44. Чирков Максим,“Профилятор gprof”. http://www.opennet.ru/docs/RUS/gprof/;

Приложение

Приложение 1. “Минимальное использование оперативной памяти”

Ниже представлен текст программы-теста, которая должна производить некоторые вычисления, не использующие оперативную память. Промежуточные результаты будут храниться в регистрах или кеше.

В качестве параметра программе передается целое число – количество итераций, которое должен совершить главный цикл в программе. Встроенный параметр `z` – постоянный параметр для отдельного профайлера. Он нужен для того, чтобы иметь возможность скомпенсировать накладные расходы, возникающие при работе с профайлером.

Исходный код программы:

```
#include <iostream>
const int z = 1024 << 8; // ratio factor

int main(int argc, char *argv[]){
    int n = atoi(argv[1]);
    int s = 0;
    for (int j = 0; j < z; j++) {
        for (int i = 0; i < n; i++) {
            s += i;
        }
    }
    std::cout << s;
    return 0;
}
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj2/tests/test01-no-memory/Main.cpp

Приложение 2. “Активное использование оперативной памяти”

Ниже представлен текст программы-теста, которая должна производить некоторые вычисления, использующие оперативную память.

В качестве параметра программе передается целое число `n`. Сначала в программе создается массив размерности `n` и заполняется произвольными числами. Затем с этим массивом производятся некоторые вычисления. Также в программе определен встроенный параметр `z`. Он нужен для того, чтобы иметь возможность скомпенсировать накладные расходы, возникающие при работе с профайлером.

Исходный код программы:

```
#include <iostream>
const int z = 2048; // ratio factor

int main(int argc, char *argv[]){
    int n = atoi(argv[1]);
    int *a = new int[n];
    for (int i = 0; i < n; i++) a[i] = i;
    int s = 0;
    for (int j = 0; j < z; j++) {
        for (int i = 0; i < n; i++) {
            s += a[i];
            a[i] += s;
        }
    }
    std::cout << s;
    return 0;
}
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj2/tests/test02-memory/Main.cpp

Приложение 3. “Использование оперативной памяти с частыми промахами кеша”

Ниже представлен текст программы-теста, которая должна производить некоторые вычисления, использующие оперативную память. При этом должны происходить частые промахи кеша.

В качестве параметра программе передается целое число `n`. Программа создает прямоугольный массив из `n` строк, для каждой из которых выделено по странице оперативной памяти. Массив заполняется произвольными числами, и производятся некоторые вычисления над ним так, что каждая операция обращается к новой странице памяти. Также в программе определен встроенный параметр `z`, который нужен для того, чтобы иметь возможность скомпенсировать накладные расходы, возникающие при работе с профайлером.

Исходный код программы:

```
#include <iostream>
const int m = 1024;
const int z = 64;

int main(int argc, char *argv[]){
    int n = atoi(argv[1]);
    int **a = new int*[n];
    for (int i = 0; i < n; i++){
        a[i] = new int[m];
        for (int j = 0; j < m; j++) {
            a[i][j] = i + j;
        }
    }
    int s = 0;
    for (int t = 0; t < z; t++) {
        for (int j = 0; j < m; j++) {
            for (int i = 0; i < n; i++) {
                s += a[i][j];
                a[i][j] += s;
            }
        }
    }
    std::cout << s;
    return 0;
}
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj2/tests/test03-extra-memory/Main.cpp

Приложение 4. “Вариация длины цикла”

Ниже представлен код программы-теста, суть которой заключена в двух циклах. Общее количество итераций вложенного цикла фиксировано. Варьируется количество итераций вложенного цикла за одну итерацию внешнего.

Также в программе определен встроенный параметр `z`. Он нужен для того, чтобы иметь возможность скомпенсировать накладные расходы, возникающие при работе с профайлером.

Исходный код программы:

```
#include <iostream>
const int n = 1 << 24;
const int z = 256; // ratio factor

void main(int argc, char *argv[]) {
    int k = atoi(argv[1]);
    int m = n / k;
    int s = 0;
    for (int t = 0; t < z; t++) {
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < k; j++) {
                s++;
            }
        }
    }
    std::cout << s;
}
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj2/tests/test11-cycle/Main.cpp

Приложение 5. “Периодически выполняемые операции”

Ниже представлен код программы-теста, смысл которого схож с предыдущим тестом. Реализация использует один цикл и счетчик, обнуление которого влечет за собой выполнение некоторого действия.

В программе определен встроенный параметр `z`. Он нужен для того, чтобы иметь возможность скомпенсировать накладные расходы, возникающие при работе с профайлером.

Исходный код программы:

```
#include <iostream>
const int n = 1 << 22;
const int z = 256; // ratio factor

void main(int argc, char *argv[]) {
    int k = atoi(argv[1]);
    int s = 0;
    for (int t = 0; t < z; t++) {
        int j = k;
        for (int i = 0; i < n; i++) {
            if (!j) {
                s += i;
                j = k;
            }
            j--;
        }
    }
    std::cout << s;
}
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj2/tests/test12-divisibility/Main.cpp

Приложение 6. “Вынужденная потеря контекста”

Ниже представлен код программы-теста, цель которого будет найти проблемы в методе семплирования. Функия Sleep выступает здесь как команда передать контекст.

Исходный код программы:

```
#include <iostream>
#include <windows.h>
const int z = 1000;

void main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    int s = 0;
    for (int t = 0; t < z; t++) {
        for (int i = 0; i < n; i++) {
            s += i;
        }
        Sleep(1);
    }
    std::cout << s;
}
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj2/tests/test21-sleep/Main.cpp

Приложение 7. “Выравнивание данных”

Ниже представлен код программы-теста, цель которого проверить как ведет себя профайлер на программе, в которой часто возникают проблемы с выравниванием данных.

В программе определен встроенный параметр `z`. Он нужен для того, чтобы иметь возможность скомпенсировать накладные расходы, возникающие при работе с профайлером.

Исходный код программы:

```
#include <iostream>
const int z = 1024;

int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    char *a = new char[n * 4];
    for (int i = 0; i < n; i++) ((int *)a)[i] = i;
    int s = 0;
    n = n * 4 - 3;
    for (int j = 0; j < z; j++) {
        for (int i = 0; i < n; i++) {
            int *p = (int *)(a + i);
            s += *p;
            *p += s;
        }
    }
    std::cout << s;
    return 0;
}
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj2/tests/test31-mem-misalign/Main.cpp

Приложение 8. “Вызовы процедур”

Ниже представлен код программы-теста, цель которого проверить, производит ли профайлер внедрение собственного кода в функции исследуемой программы.

В программе определен встроенный параметр `z`. Он нужен для того, чтобы иметь возможность скомпенсировать накладные расходы, возникающие при работе с профайлером.

Исходый код программы:

```
#include <iostream>
const int z = 1024 << 4;

#pragma auto_inline(off)
int foo(int a) {
    return a + (a >> 8) + (a << 1);
}
#pragma auto_inline()

void main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    int s = 0;
    for (int t = 0; t < z; t++) {
        for (int i = 0; i < n; i++) {
            s += i ^ foo(s);
        }
    }
    std::cout << s;
}
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj2/tests/test41-proc-call/Main.cpp

Приложение 9. “Результаты тестирования”

Полную таблицу результатов тестирования можно найти по адресу:

http://willzyx-edu-project.googlecode.com/svn/trunk/Proj2/tests/results.xls

Файл представляет собой электронную талицу. Для каждого теста создан отдельный лист. В верхней части каждого листа собраны в таблицу полученные в результате тестирования показания. Ниже таблица дублируется только уже с приведенными к секундам единицами измерений. В нижней части листа добавлены графики, построенные по приведенным данным.