Санкт-Петербургский Государственный Университет

Математико-Механический факультет

Кафедра системного программирования

Исследование и тестирование семплирующего метода профайлинга на примере профилировщика производительности Intel VTune Amplifier XE 2011

Курсовая работа студента 345 группы

Одерова Романа Сергеевича

Научный руководитель. . . . . Баклановский М.В.

старший преподаватель

кафедры системного программирования

Санкт-Петербург

2012

Оглавление

ВВЕДЕНИЕ

1. Профилирование и анализаторы производительности

В последнее время наблюдается тенденция повышения производительности программного обеспечения. Возможности совершенствования процессоров ограничиваются существующими технологиями, тактовая частота почти достигла предельного значения, и улучшения проводятся экстенсивным путем [9]. В связи с этим все большее внимание уделяется оптимизации программ. Высокопроизводительные программные продукты пользуются огромной популярностью, например, в сфере хранения и управления большими объемами данных. Такие системы должны оптимально распоряжаться имеющимися ресурсами. Можно вспомнить закон Парето («принцип 20/80»), из которого следует, что примерно 20% кода работает около 80% времени. Поэтому следует искать эти участки и попытаться их улучшить.

Для оптимизации программного обеспечения необходимы специальные средства, позволяющие оценить его производительность, выявить причины замедления и найти непосредственно те части кода, которые необходимо переписать более рационально. Этим и занимается профайлер, или профилировщик/анализатор производительности, – программа, способная подсказать программисту, что пагубно влияет на исполнение написанного кода, и найти «горячие» участки (места, требующие сравнительно большего количества памяти, времени и других ресурсов).

Профилировщики способны замерять и фиксировать различные параметры системы, например:

- время исполнения программы или функции
  (в тактах процессора, секундах, миллисекундах...)
- количество промахов кэша
- критические секции (Locks and Waits)
- количество неверно угаданных переходов
- объем использующейся памяти
- стеки вызовов функций
  и другие... [7,8]

2. Существующие решения

Рынок продуктов, занимающихся анализом производительности программ, довольно разнообразен. Конкуренты могут отличаться между собой элементами функциональности, возможностью работы под разными операционными системами или на процессорах различной архитектуры.

Среди наиболее известных и распространенных можно выделить:

- **Intel® VTune™ Amplifier XE.** [6]
  Создан компанией Intel. Эволюционировал из продукта Intel® VTune™ Performance Analyzer с добавлением функциональности Intel® Thread Profiler. Ориентирован на процессоры Intel. Полный набор функциональности доступен под ОС Microsoft Windows, под ОС Linux – с ограничениями.
- **Windows Performance Analysis Toolkit.** [20]
  Создан компанией Microsoft. Специализирован под ОС Microsoft Windows. Включает в себя Windows Performance Recorder [WPR] (основан на Event Tracing for Windows), Windows Performance Analyzer (инструмент для обработки информации, собранной WPR) [21,22]. Xperf – предыдущий аналог WPR – также поставляется в данном пакете утилит [16].
- **AMD CodeAnalyst Performance Analyzer.** [4]
  Создан компанией AMD. Ориентирован на процессоры AMD. Работает и в ОС Microsoft Windows, и в ОС Linux.
- **AQtime Pro.**[5]
  Создан компанией SmartBear. Работает на процессорах фирм AMD и Intel под руководством ОС Microsoft Windows.

Несмотря на обилие различных анализаторов производительности, существует два основополагающих подхода к профилированию программ: инструментирование и семплирование. Суть метода инструментирования заключается в том, что в исходный или скомпилированный код вставляются дополнительные команды, которые, в свою очередь, собирают информацию о ходе исполнения программы. Семплирование – метод, при котором производятся замеры системных счетчиков программы через определенный интервал времени, например, по какому-либо прерыванию [10]. На основе полученной информации анализируется работа приложения. Оба подхода имеют свои достоинства и недостатки. Более подробное описание этих методов будет дано позже.

Стоит отметить, что все анализаторы производительности, упомянутые выше, нацелены на профилирование программ, работающих с уровнем привилегий User-mode [18], хотя некоторые и имеют необходимую функциональность для считывания данных из ядра операционной системы. Таким образом, не существует подходящего инструмента для профилирования кода, исполняющегося в ядре ОС с уровнем привилегий Kernel-mode.

Летом 2011 года компания EMC [3] выставила на летнюю школу научно-исследовательский проект, связанный с разработкой профайлера ядра ОС Microsoft Windows Server 2008 R2 x64 Edition. Команда студентов, состоявшая из 3 человек, познакомилась с темой и проделала первые шаги в исследовании. Были освоены базовые приемы работы на «низком уровне»: написание драйверов, способы их установки в систему, логирование из драйвера в файл, получение информации о работающих процессах и измерение времени в ядре [8]. После успешного завершения запланированных работ подписан контракт на непосредственную разработку многофункционального профилировщика, способного с учетом недостатков конкурентов выполнять профилирование в ядре операционной системы. В течение года мы занимались изучением подходов к анализу производительности и справлялись с многочисленными возникающими проблемами. В апреле 2012 года на ежегодном отчетном семинаре научно-исследовательских проектов в EMC был успешно представлен результат работ по проекту в целом. Данная курсовая работа является частью проведенных исследований, некоторая часть которых опубликована на конференции «СПИСОК-2012»[1,2]. В ней рассмотрена технология семплирования и описывается набор стресс-тестов, направленных на ее изучение на основе семплирующего профайлера Intel VTune Amplifier XE.

.

1. Предварительное описание и постановка задачи

Для реализации поставленного компанией EMC задания необходимо было провести ряд исследований. Как упомянуто выше, некоторые из целей научно-исследовательского проекта были взяты в качестве темы курсовой работы, а именно: рассмотрение двух методов анализа производительности программного обеспечения (семплирование и инструментирование), их достоинств и недостатков, анализ возможностей существующих профайлеров на примере Intel VTune Amplifier XE. В частности, одним из самых приоритетных стартовых направлений стало определение подхода к профилированию программного обеспечения. Для этого было проведено сравнение известных методов (семплирования и инструментирования).

1.1 Семплирование

В данном методе профайлер начинает работу по какому-либо прерыванию (например, по специальному прерыванию IRQL 27 Profile) [19]. Запустившись в качестве обработчика, он снимает показания системных счетчиков, получает стек вызовов функции на текущий момент и т.п. Реализация описываемого подхода относительно несложная. Важно отметить, что в ходе семплирования не происходит модификация кода анализируемой программы, и, следовательно, не нужно контролировать ее поведение и следить за корректностью работы. Метод статистический, поэтому главный его недостаток – аппроксимирующий подход: основываясь на периодическом сборе информации, необходимо построить «приближение» реальной работы программы. При этом не исключается возможность пропустить вызовы функции, попадающие строго между семплами – моментами времени, в течение которых работает профайлер. Однако стоит учитывать и то, что это довольно редкое и маловероятное событие, исходя из закона больших чисел. Таким образом, нужно следить за корректностью полученных результатов, и применение этого метода зависит от преследуемых разработчиком целей.

1.2 Инструментирование

Как уже упоминалось выше, инструментирование заключается во внедрении непосредственно в код программы определенных инструкций, которые будут собирать необходимую информацию о процессе исполнения. Внедрение может происходить как статически (в исходный код), так и динамически, или «на лету» (в откомпилированный код с использованием дизассемблера длин). В результате можно достичь хорошей точности измерений, однако не стоит забывать о сопутствующих недостатках. Главный из

них – необходимость модификации кода программного продукта, в результате которой можно получить неожиданные ошибки в исполнении программы. Чтобы не наткнуться на различные «подводные камни», нужно корректно выполнять инъекции своего кода в исходную программу, а это требует сравнительно больших затрат на реализацию.

В виду описанных выше достоинств и недостатков существующих подходов к профилированию ПО, разработчики последних версий профилировщиков все больше и больше склоняются к реализации, основанной на семплировании. Анализатор производительности Intel VTune Amplifier XE 2011 не стал исключением. Как заявил инженер-консультант по техническим вопросам Intel Performance and Analysis Tools Lab Peter (Zhen Yu) Wang [11], с недавнего времени технология сбора информации поменялась, и теперь Intel VTune тоже использует статистический подход. Результат такой «эволюции» - снижение накладных расходов во время работы.

Intel VTune Amplifier XE был выбран в качестве «подопытного» продукта для изучения технологии семплирования.

1.3 Intel VTune Amplifier XE

Intel VTune Amplifier XE, ранее известный как Intel VTune Performance Analyzer с дополнительной функциональностью Intel Thread Profiler, представляет собой новейшую утилиту, способную проводить комплексный анализ программного обеспечения [6]. Поставляется как отдельно, так и в комплекте с другими инструментами (например, Intel Parallel Studio XE, Intel C++ Studio XE, Intel Fortran Studio XE, Intel Cluster Studio XE) [12]. Является платным продуктом. Один Intel VTune Amplifier XE (без дополнительного пакета утилит) стоит от $899 и для ОС Windows, и для ОС Linux. Для сравнения ниже приведены цены некоторых пакетов, включающих в себя VTune [14].

(W)=Windows (L)=Linux	**Intel Parallel Studio XE Fortran and C++**		**Intel C++ Studio XE**	
	New License	**Renewal**	**New License**	**Renewal**
Single User	$1,899 (W) $2,249 (L)	$759 (W) $899 (L)	$1,499 (W & L)	$599 (W & L)
Floating - 2 users	$9,999 (W & L)	$3,999 (W & L)	$6,499 (W & L)	$2,599 (W & L)
Floating - 5 users	$19,999 (W & L)	$7,999 (W & L)	$12,999 (W & L)	$5,199 (W & L)

Таблица 1

Возможности Intel VTune очень обширны и во многих случаях предусматривают настройку со стороны пользователя [15]. Ниже приведены лишь некоторые из них:

- нахождение функций, работающих наибольшее время
- просмотр исходного кода программы с указанием конкретного «горячего» места, найденного в результате профилирования
- построение дерева вызовов
- получение значений специальных процессорных счетчиков с использованием технологии Performance Monitoring Unit [13,17] (например, количество промахов кэша, тактов процессора и т.д.).
- удобная и понятная визуализация собранных данных
- большой набор различных типов анализа производительности, которые позволяют оптимально выбирать нужное направление оптимизации и предоставляют различные возможности по ее осуществлению. Анализы разбиты на группы:
    - алгоритмические анализы
        a. *Lightweight Hotspots*

           Ищет участки кода, на которые тратится больше всего времени выполнения; не собирает стеки вызовов.
        b. *Hotspots*

           Ищет участки кода, на которые тратится больше всего времени выполнения; собирает стеки вызовов.
        c. *Concurrency*

           Анализирует, как используются доступные логические CPU, и предлагает набор точек в программе для возможного внедрения или усовершенствования параллелизма программы.
        d. *Locks and Waits*

           Определяет, где программа вынуждена ждать операций синхронизации или ввода/вывода и как это влияет на производительность.
    - анализы, зависящие от семейства используемых процессоров

      Получение необходимой для профилирования информации определяется особенностями структур и технологий соответствующего семейства процессоров Intel (Sandy Bridge, Nehalem, Atom, Core 2…).

1.4 Цель

Итак, в рамках проекта, предложенного компанией EMC, необходимо выявить слабые места современных подходов к профилированию программ и непосредственно самих анализаторов производительности. Таким образом, целью работы стало более детальное исследование возможностей семплирующего метода профайлинга на примере одного из наиболее известных инструментов Intel VTune Amplifier XE 2011. Запланировано написание набора тестов, которые позволят проследить преимущества упомянутого выше подхода и выявить его недостатки на определенных примерах. Стоит отметить, что основным направлением тестирования стали анализы типа «Hotspots» и «Lightweight Hotspots», т.к. именно их реализация основана на семплировании.

2. Решение поставленной задачи: описание тестов и комментарии

Хотелось бы сразу отметить, что все приведенные результаты тестов получены в ходе многочисленных прогонов; в работе показаны усредненные данные. Если какой-то результат представляет интерес, как единичный частный случай, это будет отдельно оговариваться в тексте работы. Каждая описываемая программа имеет возможность логирования в файл времени своего исполнения, что помогает сравнивать реальное время работы без профилирования с результатами, выданными анализатором производительности.

2.1 Empty Function Test

Первым тестом стала программа под названием «Empty Function». В ней реализована одна «почти пустая» функция: запускается цикл, внутри которого происходит инкрементирование переменной.

```
int main(int argc, char **argv){
    int a = 0;
    for (int i = 0; i < 10; i++)
         a += i;
    return a;
}
```

Стоит отметить, что количество итераций цикла не очень большое, из-за чего Intel VTune Amplifier XE выдал результат «No data to show» и программа не была отражена в результатах анализа. Т.е. по мнению профайлера, функция main занимает 0 мс времени, что, естественно, невозможно. Причиной тому метод семплирования, на котором основана деятельность Intel VTune: программа успевает отработать до того, как сгенерируется первое прерывание и его обработчик начнет собирать информацию о потоке исполнения.

Итак, рождается первая идея: определить ситуации, когда профайлер выдает ложные результаты, при попытках детектирования работающих функций и таким образом выявить недостатки метода семплирования.

2.2 Simple Asm Test

Теперь убедимся в том, что профайлер выдает корректный результат с точки зрения действий, производимых анализируемой программой. Т.е. хочется проверить, что VTune не выдает ничего лишнего, не имеющего отношения к профилируемому коду. Для этого реализована простейшая программа на ассемблере, которая выполняет элементарное

действие (заполнение регистра некоторым значением). Учитывая предыдущий опыт, необходимо повторять операцию с регистром много раз (здесь `10000000000`) так, чтобы как минимум один семпл был зафиксирован во время выполнения теста. Ниже представлен фрагмент кода функции `Simple()`.

```
    mov       rcx, 10000000000
l1:
    mov       rax,rbp
    loop          l1
```

Результат, выданный профилировщиком:

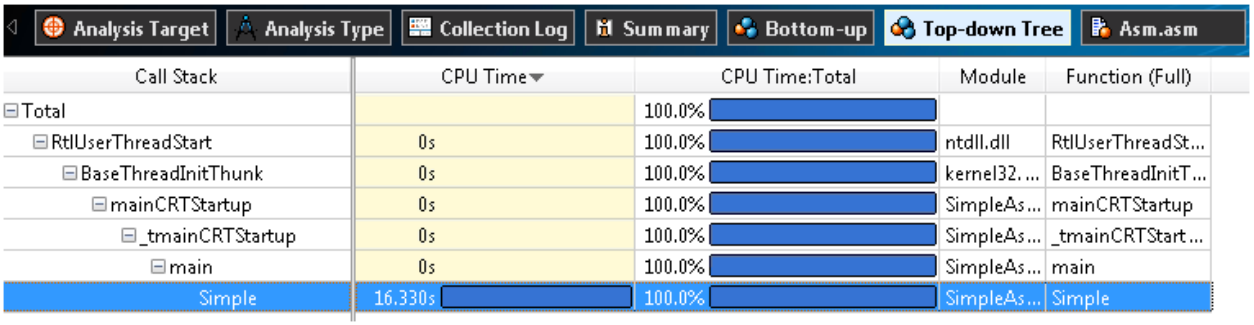

Рисунок 1

2.3 Function Stack Test

Данный тест реализует цепочку вызовов функций. С его помощью можно понять, ошибается ли семплирование при анализе вложенных функций. Схематично код программы (точнее говоря, стек вызовов) можно изобразить так:

```
main
---> ExternalFunc
------> MiddleFunc
---------> InternalFunc
```

Происходит последовательный вызов трех вложенных функций. Функция `main` вызывает `ExternalFunc`, `ExternalFunc` вызывает `MiddleFunc`, `MiddleFunc` вызывает `InternalFunc`. Сначала тело каждой из них не содержало никаких действий, и, как уже должно быть очевидно, профайлер не заметил их. Но это неинтересный случай, поэтому функции были наполнены примитивными арифметическими действиями (вычисление факториала числа), чтобы время их работы стало ощутимым. После этого результат анализа оказался странным, на первый взгляд, но объясняемым при более

детальном рассмотрении. Профайлер распознал только `main` и `MiddleFunc` функции. Было выдвинуто предположение, что в дело вмешалась оптимизация компилятора. Действительно, при отключении всех оптимизирующих опций получен ожидаемый результат:

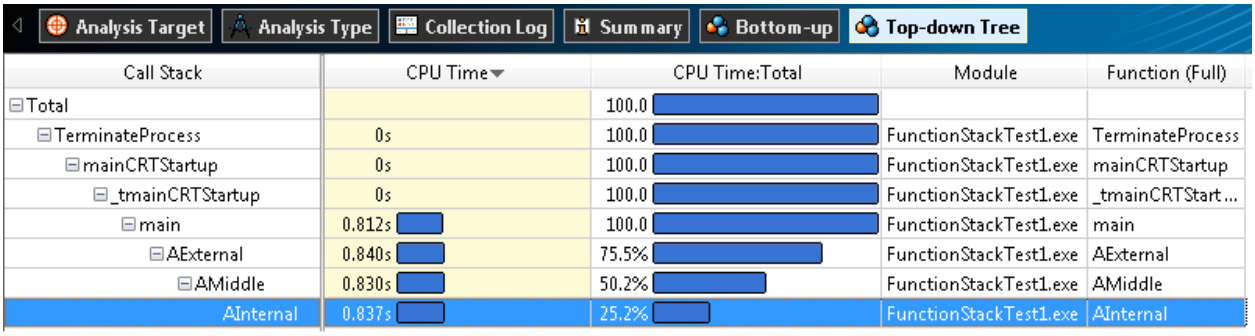

Рисунок 2

Важное замечание: так как внутри `main`, `ExternalFunc`, `MiddleFunc` и `InternalFunc` выполнялись одни и те же действия, они должны были работать приблизительно одинаковое время. Профайлер корректно отобразил и эту часть анализа, сопоставив каждой из функций примерно 25% от всего времени работы теста (см. рис.2).

Итак, профайлер смог раскрыть стек и не потерять ни одной функции.

2.4 Create Process Test

Тест направлен на проверку того, как Intel VTune Amplifier XE 2011 справляется с отслеживанием различных процессов, используемых в профилируемой программе. Анализируемое приложение порождает N новых процессов из программы EmptyFunction, использованной в тесте №1. Процесс завершается сразу же после порождения.

```
NewProcessCreateAndExit(){
    <...>
    CreateProcess("EmptyFunction.exe",...);
    <...>
}
<...>
void Main(){
    for (i = 0; i < N; i++)
        NewProcessCreateAndExit();

}
```

Профайлер достойно справился с этим тестом, показав следующие результаты:

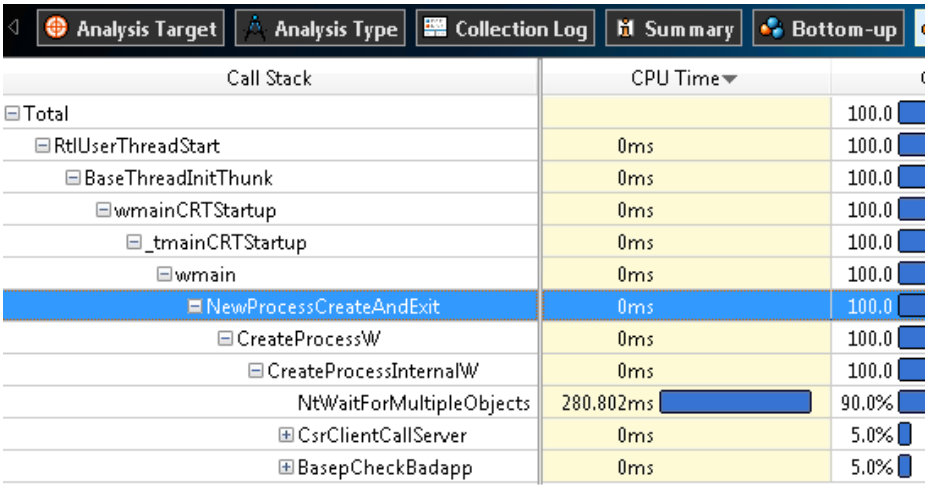

Рисунок 3

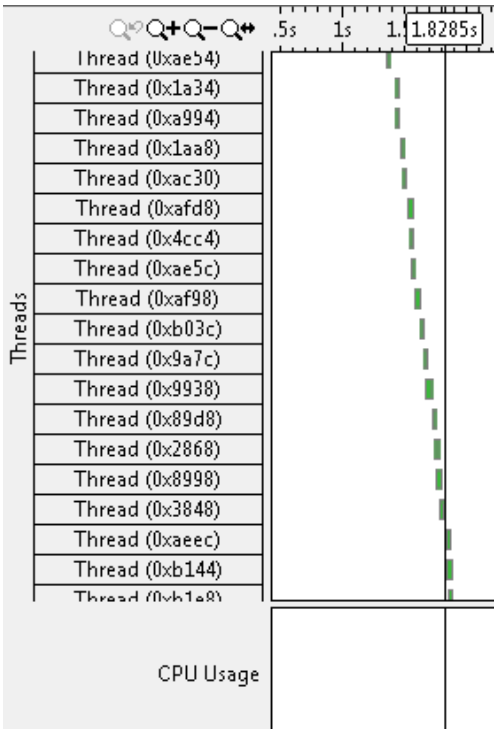

Рисунок 4

На рисунке 4 отчетливо видно задуманное поведение программы: последовательное открытие и закрытие порожденных процессов. Но не все так безоблачно, как кажется. В данном тесте проявились огромные накладные расходы на профилирование. Без профайлинга программа работала порядка 15 секунд при параметре N равном 500, а с Intel VTune Amplifier XE в режиме Hotspots Analysis – около 1000 секунд. Т.е. в результате действий анализатора время работы увеличилось более чем в 60 раз. Сразу же стоит оговорить следующее: не у всех тестов наблюдались настолько повышенные накладные расходы. Все зависит от выбора типа анализа в меню VTune и от «содержимого» программы. Например, Lightweight Hotspots анализ почти не влияет на работу приложения, в отличие от Hotspots. Также можно отметить чрезмерно длительную обработку собранных результатов после завершения работы программы, длившуюся порядка 30 минут.

2.5 Counter Losses Test

Цель теста – оценить количество пропущенных вызовов функции в результате семплирования.

Идея: вызывать в циклах (в разных местах программы П1) функцию `IncCounter()`, замерить время работы функции с профайлером/без профайлера, оценить замедление (накладные расходы по времени) и количество пропущенных вызовов. Функция `IncCounter()` увеличивает значение заранее выделенного счетчика на единицу. В конце программы П1 значение счетчика логируется в файл. Таким образом становится возможным определить реальное количество вызовов функции `IncCounter()` и сравнить с тем, сколько вызовов зафиксировал профайлер.

Важное замечание: как уже упоминалось ранее, Intel VTune перешел на новую технологию работы [Peter Wang] и потерял возможность подсчета точного количества вызовов функций. Поэтому для реализации задуманной идеи теста необходимо каким-то другим способом подсчитать количество вызовов, замеченных профилировщиком.

Для замера «реального» времени работы `IncCounter()` без влияния профайлера написана отдельная программа (П2), реализующая подсчет времени ее работы. При этом в программах П1 и П2 для увеличения точности проводилось большое количество вызовов рассматриваемой функции.

Времена работы программ и реализованной внутри них функции `IncCounter()` в зависимости от типа анализа представлены в Таблице 2.

Анализ	**HotSpots (HS)**	**Lightweight HS**
IncCounter в программе П2 (итераций $75*10^8$)	575 054 ms	25 980 ms
Программа П2	628 673 ms	652 538 ms
IncCounter в программе П1 (итераций $3*10^8$)	24 708 ms	1 158 ms
Программа П1	27 121 ms	27 893 ms

Таблица 2

Замечание:

интервал семплирования – 10 мс

`IncCounter()` без профайлера 21585 мс (количество итераций $3*10^8$)

Введем обозначения:

It' - количество итераций, замеченных профайлером

It - общее кол-во итераций

T_1 - время работы программы П1 без профайлинга

T_2 - время работы программы П1 с профайлингом

T_0 - время работы $3*10^8$ `IncCounter()` без профайлинга

T_x - время работы $3*10^8$ `IncCounter()` с профайлингом

$K = \frac{T_2}{T_1}$ - коэффициент замедления всей программы

T_2.IncCounter – время работы функции `IncCounter()` в программе П1 с профайлингом.

Были выдвинуты две гипотезы (формулы подсчета), касающиеся вычисления процента потерь вызовов функции `IncCounter()` при профилировании:

- It'/It = T_2.IncCounter/T_x
- It'/It = T_2.IncCounter/(T_0 * K)

Считаем, что всевозможные конструкции языка (например, циклы for(…)) выполняются за пренебрежимо малое время относительно времени работы функций и не влияют на показания профилировщика.

Тогда имеем следующий противоречивый результат:

	T_2.IncCounter/T_x	**T_2.IncCounter/(T_0 * K)**	**K**
Lightweight HS	1.1143	0.048084	1.11572
HS	1.0742	1.055163	1.08484

Таблица 3

Т.о. из таблицы можно увидеть, что коэффициент замедления K получился больше 1, однако в описанном выше теоретическом обосновании было запланировано $K \leq 1$. Это говорит о том, что профайлер «заметил» больше функций, чем есть на самом деле. Итак, мы экспериментально доказали, что Intel VTune Amplifier XE 2011 использует статистический метод анализа, аппроксимируя реальную работу программы.

2.6 Saw Test

Для более детального исследования метода семплирования был придуман ещё один тест под названием «Пила» («Saw»).

Идея: соорудить из стека вызовов функций «пилу». Т.к. семплирование – аппроксимирующий метод, то можно подобрать такую программу, при профилировании которой семплы будут попадать строго между вызовами некоторой функции. Следовательно, профайлер не сможет заметить никаких следов этой функции.

Модель реализации: вызывать различные функции N раз «в глубину», причем N можно выбирать случайным образом.

Графически тест можно пояснить так (оранжевые точки – семплы, во время которых происходит сбор стека вызовов):

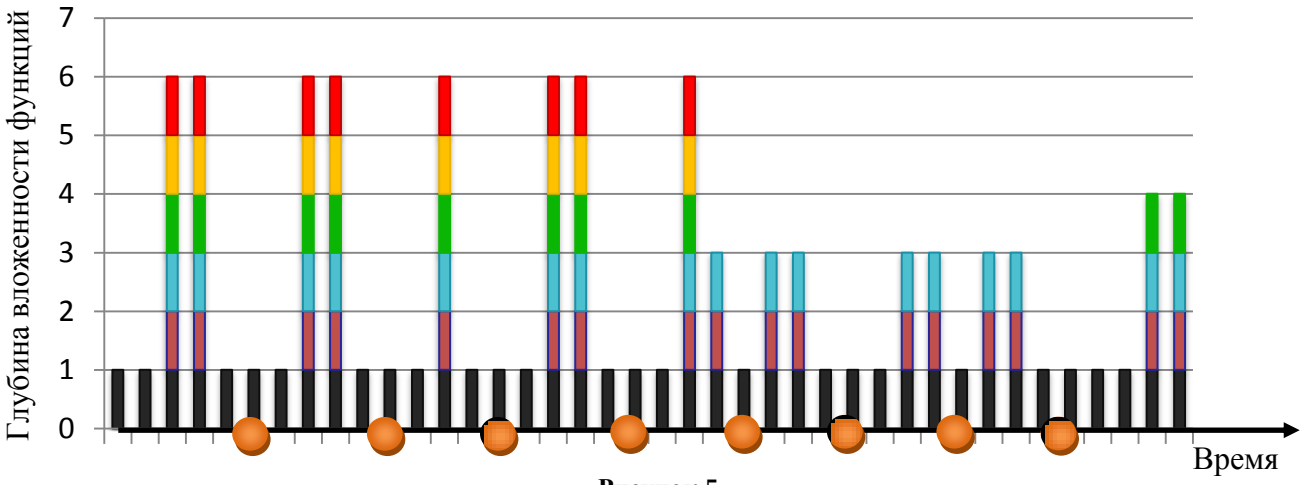

Рисунок 5

Первая версия теста представляет собой функцию `main()` с одним большим циклом, внутри которого происходят вызовы функций `f1()` и `GoRandDeep(int rand)`. `Main()` наполнен вычислениями факториалов некоторых чисел. Функция `f1()` вызывает функцию `f2()` и так далее до функции `f10()`:

```
f1() -> f2() -> f3() -> … -> f9() -> f10()
```

Последняя функция цепочки производит вычисление факториала небольшого числа (для того, чтоб время работы было малым, но не равным нулю). Все остальные содержат лишь вызовы следующих функций. Функция `GoRandDeep(int rand)` вызывает себя `rand` раз и при последнем вызове содержит те же действия, что и `f10()`.

Схема программы:

```
int main(){
    for(experiment = 10000; experiment >= 0; experiment--){
        <…>//здесь вычисляются факториалы каких-то чисел
        GoRandDeep(int rand);//производится на каждой 100
    итерации
        <…>//здесь вычисляются факториалы каких-то чисел
        f1();//производится на каждой 75 итерации
        <…>//здесь вычисляются факториалы каких-то чисел
        f1();//производится на каждой 125 итерации
    }
}
```

Результат анализа оправдал ожидания: функция `f10()`, находящаяся на вершине цепочки функций f, как и функция `GoRandDeep()`, не была замечена профилировщиком вообще.

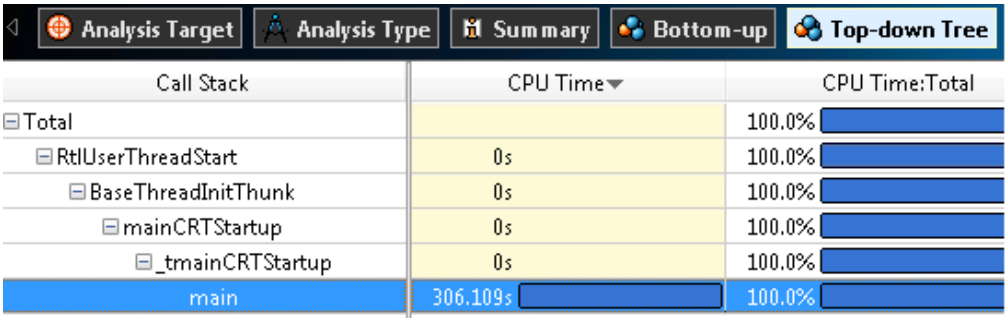

Рисунок 6

Хотя несколько раз в результатах фигурировала функция `GoRandDeep()`, что, вероятно, связано с ее «самовызовами»:

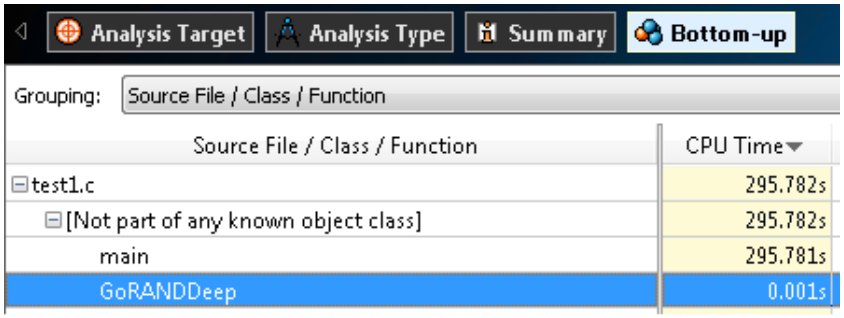

Рисунок 7

Затем решено было поменять местами в коде программы вызовы функций `GoRandDeep()` и `f1()`. Результат не изменился:

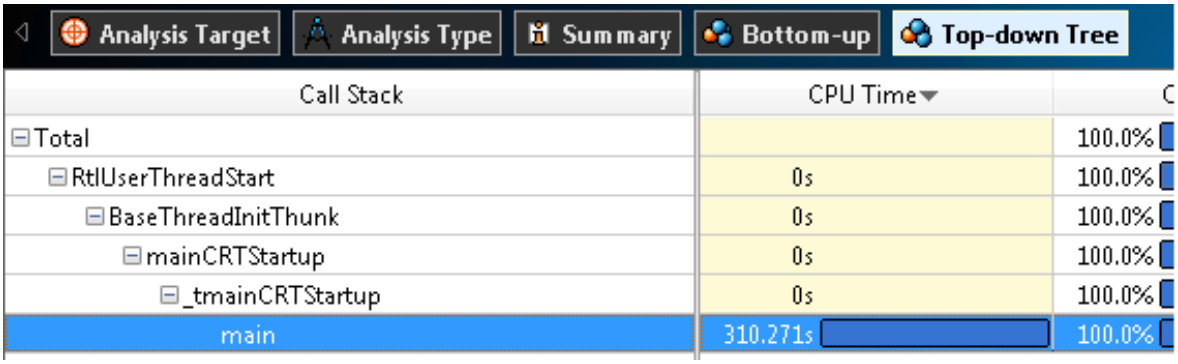

Рисунок 8

Зафиксированной осталась по-прежнему только функция `main()`. Хотя теперь редко проскакивала в результатах функция `f10()`.

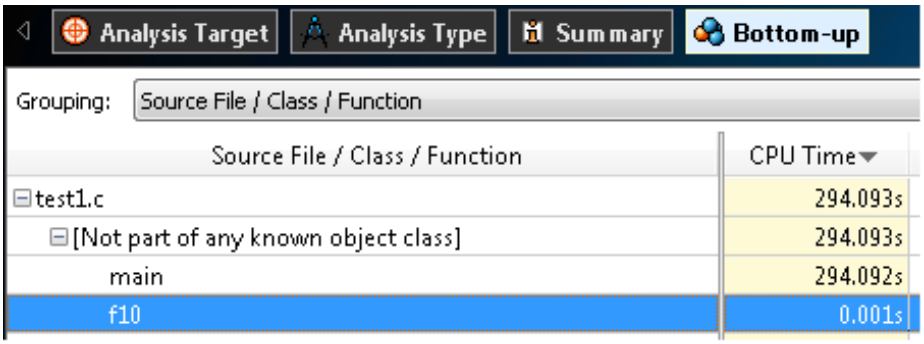

Рисунок 9

Затем была произведена попытка провести тест следующим образом:

Было высчитано время работы цикла по подсчету факториала числа 250. Оно составило около 2 мс. Далее все факториалы были заменены на вычисление одного и того же:

```
for (i_count = 250; i_count >= 1; i_count--)
    fact *= i_count;
```

Затем код заполнялся этими факториалами, причем между каждыми двумя стоял вызов функции `GoRandDeep()` или `f1()` (внутри них вычислялся такой же факториал). Это сделано из соображений, что интервал семплирования равен 10 мс, и тогда в предположении, что первый семпл приходится на начало программы, вызовы приходятся в середину семплирующего интервала.

Один из полученных результатов:

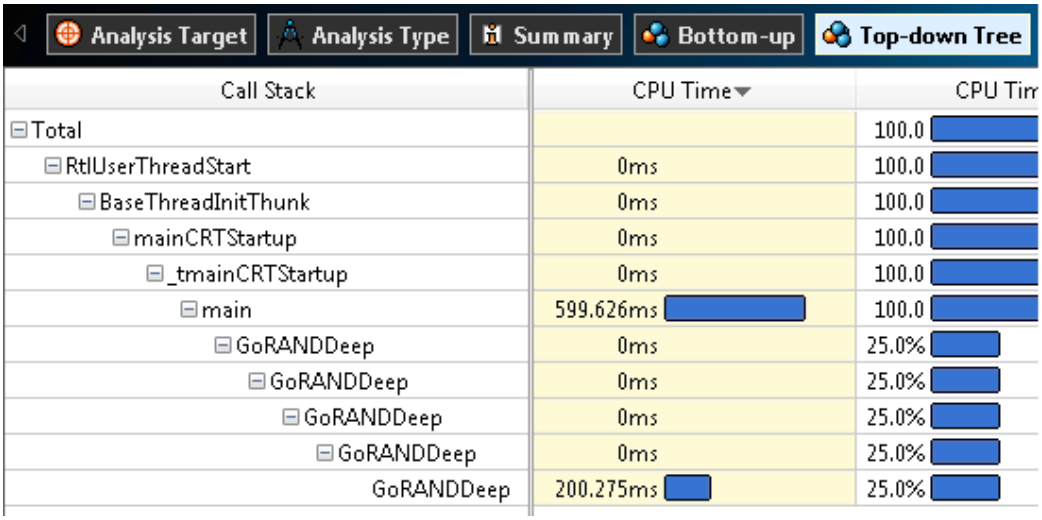

Рисунок 10

Видно, что иногда идея теста срабатывала, т.к. были моменты, когда функция `f10()`, содержащая арифметику, не была замечена (здесь отображена только `GoRandDeep()`). Но почти всегда складывалось так, что семпл попадал в середину цепочки вызовов функции `f()` и в функцию `f10()`. Следовательно, отображались некоторые пустые функции `f()` с номерами от 1 до 9. Разброс результатов означает, что простейшими методами невозможно предсказать поведения семплирования при новом запуске и высчитать временные интервалы.

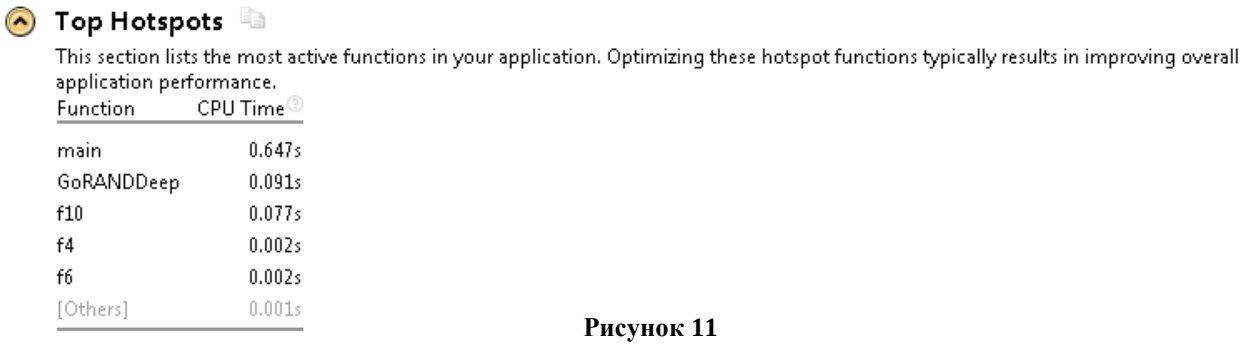

Рисунок 11

Заметим, что время функции `f10()` относительно всего времени работы программы составляло существенную часть – порядка 10%. Т.о. существуют частные случаи, когда «горячие» функции могут быть пропущены профайлером в ходе анализа, что приводит к серьезным трудностям в применении оптимизации к тестируемому коду.

Очевидно, подобная ситуация будет происходить довольно редко в реальных системах, но сам факт несовершенства метода семплирования все-таки был установлен.

Заключение

Итак, в результате проведенных тестов был изучен и протестирован семплирующий подход к профилированию приложений, найдены его слабые места. Семплирование выдает довольно точные результаты в большинстве ситуаций, однако существуют примеры, на которых оно может сработать некорректно. Суть метода – периодические замеры основных показателей производительности системы и составление из полученных данных приближенной статистики о ходе исполнения программы. Несмотря на то, что в большинстве случаев семплирование работает с приемлемой точностью, разработчики должны понимать, что решение о выборе подхода к анализу кода зависит от тех целей, которые они преследуют при попытках оптимизации.

Важно упомянуть о том, что весь полученный опыт и довольно обширный набор сведений о технике семплирования, безусловно, пригодится при реализации собственного многофункционального профайлера ядра операционной системы Microsoft Windows Server 2008 R2 x64 в рамках продолжения работы над проектом компании EMC.

Список использованной литературы

1. Исследование и тестирование семплирующего метода профайлинга на примере профилировщика производительности Intel VTune Amplifier XE 2011 / Одеров Р.С., Серко С.А. – конференция «СПИСОК-2012», СПбГУ, 2011 год. (В ПЕЧАТИ)
2. Верификация дизассемблера x86-64 / Тенсин Е. – конференция «СПИСОК-2012», СПбГУ, 2011 год. (В ПЕЧАТИ)
3. Компания EMC^2 // http://russia.emc.com
4. Описание AMD CodeAnalyst Performance Analyzer// http://developer.amd.com/tools/codeanalyst/Pages/default.aspx
5. Описание AQTime Pro Performance Profiling // http://smartbear.com/products/qa-tools/application-performance-profiling
6. Описание Intel® VTune™ Amplifier XE // http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/
7. Профилирование программ / Алексей А. Романенко // http://ccfit.nsu.ru/arom/data/PP_ICaG/06_Profiling_txt.pdf
8. Способы размещения своего кода в ядре ОС Microsoft Windows Server 2008 / Одеров Р.С., Тенсин Е.Д. – сборник трудов межвузовской научно-практической конференции «Актуальные проблемы организации и технологии защиты информации». – СПбНИУ ИТМО, Санкт-Петербург, 2011 год // стр. 100-102
9. Теоретический анализ и разработка методик оценки достоверности информации, получаемой современными профайлерами» / Булычев И.Д., дипломная работа. – СПбГУ, Мат-Мех ф-т, кафедра системного программирования, 2011 год
10. Технологии Интел для разработки параллельных программ / Глазкова Е.А. – студенческая исследовательская лаборатория Intel в МГУ // http://intel.cmc.msu.ru/sites/default/files/PP_Intel_Technologies_1.pdf
11. Упоминание о переходе на новую технологию сбора данных при профилировании./ Peter Wang, Intel Corporation// http://software.intel.com/en-us/forums/showthread.php?t=81390
12. Intel® 64 and IA-32 Architectures Optimization Reference Manual / Intel Corporation.– June 2011
13. Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3: System Programming Guide / Intel Corporation . – March 2012 // Chapter 18 – 19
14. Intel® Parallel Studio XE 2011 – Purchase // http://software.intel.com/en-us/articles/intel-parallel-studio-xe-purchase/

15. Intel® VTune™ Amplifier XE 2011 – Documentation // http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe-documentation/#win
16. Microsoft Xperf Options // http://msdn.microsoft.com/ru-ru/library/ff191081
17. Performance Monitoring Unit Sharing Guide / Peggy Irelan and Shihjong Kuo, Intel Corporation
18. User mode and kernel mode // http://msdn.microsoft.com/en-us/library/windows/hardware/ff554836(v=vs.85).aspx
19. Windows Internals (5th edition) / Mark E. Russinovich and David A. Solomon : Microsoft Press . – 2009 год// стр. 92.
20. Windows Performance Analysis Toolkit (WPT) for Windows. Developer Preview// http://msdn.microsoft.com/en-us/performance/cc709422
21. Windows Performance Analyzer // http://msdn.microsoft.com/en-us/library/hh448170.aspx
22. Windows Performance Recorder // http://msdn.microsoft.com/en-us/library/hh448205.aspx