ОПТИМИЗАЦИЯ ПРОГРАММНОГО КОДА ПО КРИТЕРИЮ БЫСТРОДЕЙСТВИЯ

А.В. Сабокарь, Д.В. Ермусевич, Н.Н. Ивашко, А.В. Лисниченко
Институт проблем математических машин и систем НАН Украины
E-mail: ermusevich@ukrsoft.net

В данной работе рассматриваются проблемы оптимизации программ реального времени по критерию быстродействия.

В системах реального времени вообще, и в системах мобильной связи в частности, показатель быстродействия является критическим. В нашем случае такая система обеспечивает вход в защищенную сессию между двумя абонентами. Рамки того времени, которое допустимо для входа в сессию, трактуется здравым смыслом и комфортностью для абонентов. И когда это время составляет больше 20 секунд между нажатием на кнопку начала связи и первым услышанным словом от собеседника – вопрос оптимизации такой системы в плане быстродействия становится сверхактуальным.

В качестве примера для оптимизации в работе рассмотрен код программы для получения секретного ключа для входа в защищенную телефонную сессию между двумя абонентами по протоколу Diffie-Hellman.

Принцип работы Diffie-Hellman таков:
если обозначить секретный ключ абонента как X, открытый ключ как Y и установить соответствие между ними как Y=A^X (A в степени X, где A является константой), то приближенно получение общего секретного ключа можно записать так.

Каждый участник вычисляет K на основе своего секретного ключа и открытого ключа другого абонента:

K=Y2^X1=(A^X2)^X1= A^X2X1

K=Y1^X2=(A^X1)^X2= A^X1X2

Как видим, вычисленное на обоих концах канала связи значение K одинаково, при этом по каналу передавались только значения Y1 и Y2, на основании которых потенциальный злоумышленник, прослушивавший канал, не сможет вычислить X1 и X2, равно как и общий секрет K.

Программа реализована на языке Assembler сигнального процессора ADSP-2188. Семейство процессоров ADSP-21xx представляет из себя набор программируемых однокристальных микропроцессоров с общей базовой архитектурой, оптимизированной для DSP и других приложений, требующих быстрых вычислений. Каждый из процессоров объединяет в себе ядро DSP архитектуры - вычислительные устройства, генераторы адресов данных и генератор адресов инструкций - с различными дополнительными модулями, такими как внутрикристальная память программ и память данных (RAM), программируемый таймер, один или два последовательных порта и порт интерфейса с хост-процессором. [1]

Предлагается процесс оптимизации программ разбить на несколько шагов:

- выявление узких мест,
- выбор стратегии оптимизации кода,
- реализацию стратегии оптимизации.

Перед тем как приступить к выявлению узких мест, необходимо составить представление о программе как таковой. Для этого мы выписали все функции программы и пересмотрели, как каждая из них вызывается, и какие функции каждая из них вызывает. В результате мы получили диаграмму иерархии всех функций программы. За основу мы взяли язык UML и как инструмент – пакет Rational Rose. В результате получилась следующая диаграмма (рис. 1).

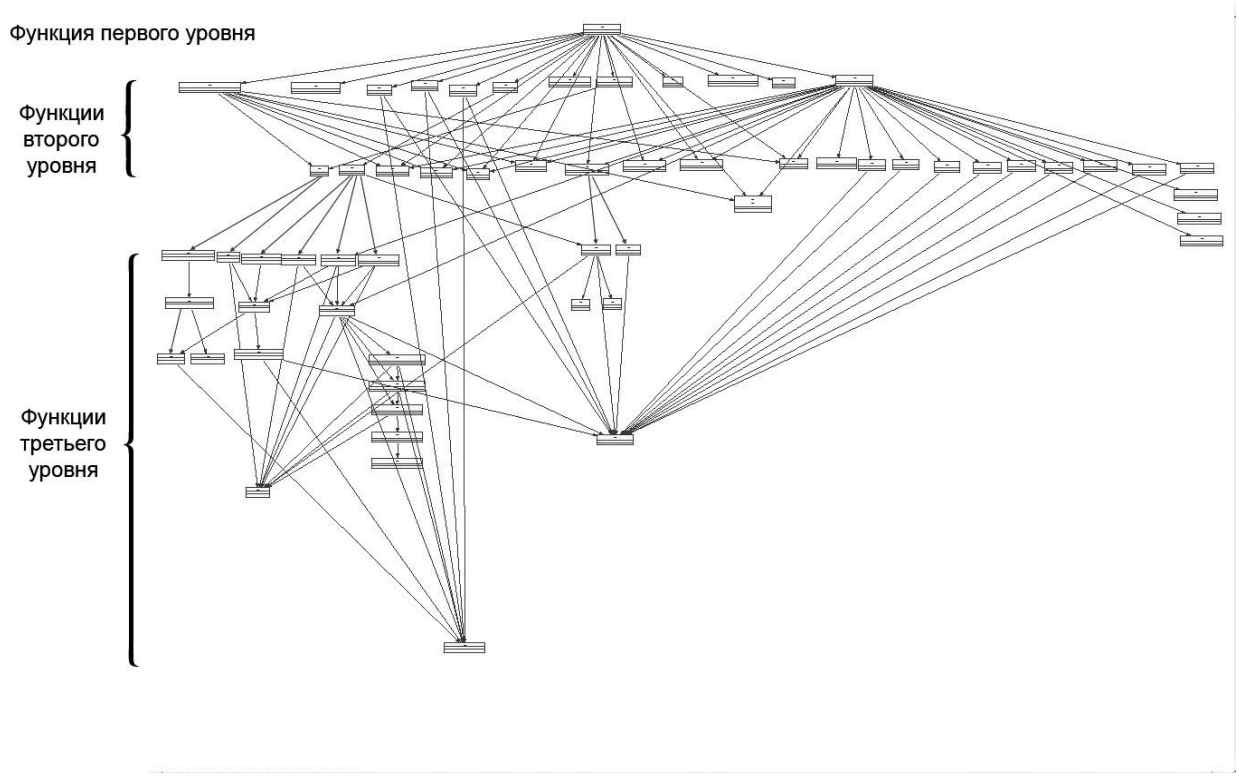

Рис. 1 Иерархия всех функций программы

Проанализировав составленную диаграмму, мы смогли выделить три условных уровня функций.

1– Главная

2– Вызываемые непосредственно главной

3– Все остальные (в основном – математические)

Далее, чтобы иметь представление о порядке вызовов этих функции и о ходе программы в целом, мы создали еще одну модель – Activity диаграмму (Рис 2).

Где по горизонтали расположены функции второго уровня, а по вертикали – все функции, вызываемые каждой из этих второуровневых. Диаграмма содержит вложенности. для каждой функции третьего уровня со сложной системой вызовов есть отдельная диаграмма.

Далее нам предстояло провести процесс тестирования согласно нашей диаграмме.

Существует два метода проверки того, сколько занимает времени тот или иной участок программы: с помощью программного отладчика или с помощью аппаратного (ICE).

Только первый метод может дать точные результаты, хотя и занимает такой процесс измерения немало машинного времени.

Все результаты тестирования заносились на диаграмму. После проведения этого этапа были обнаружены те функции и циклы, которые и представляют собой ту «математику», занимающую большую часть времени. Итак, стали известны те «узкие» места, из-за которых программа столько работает, и на которых и стоит сконцентрировать все доступные методы оптимизации.

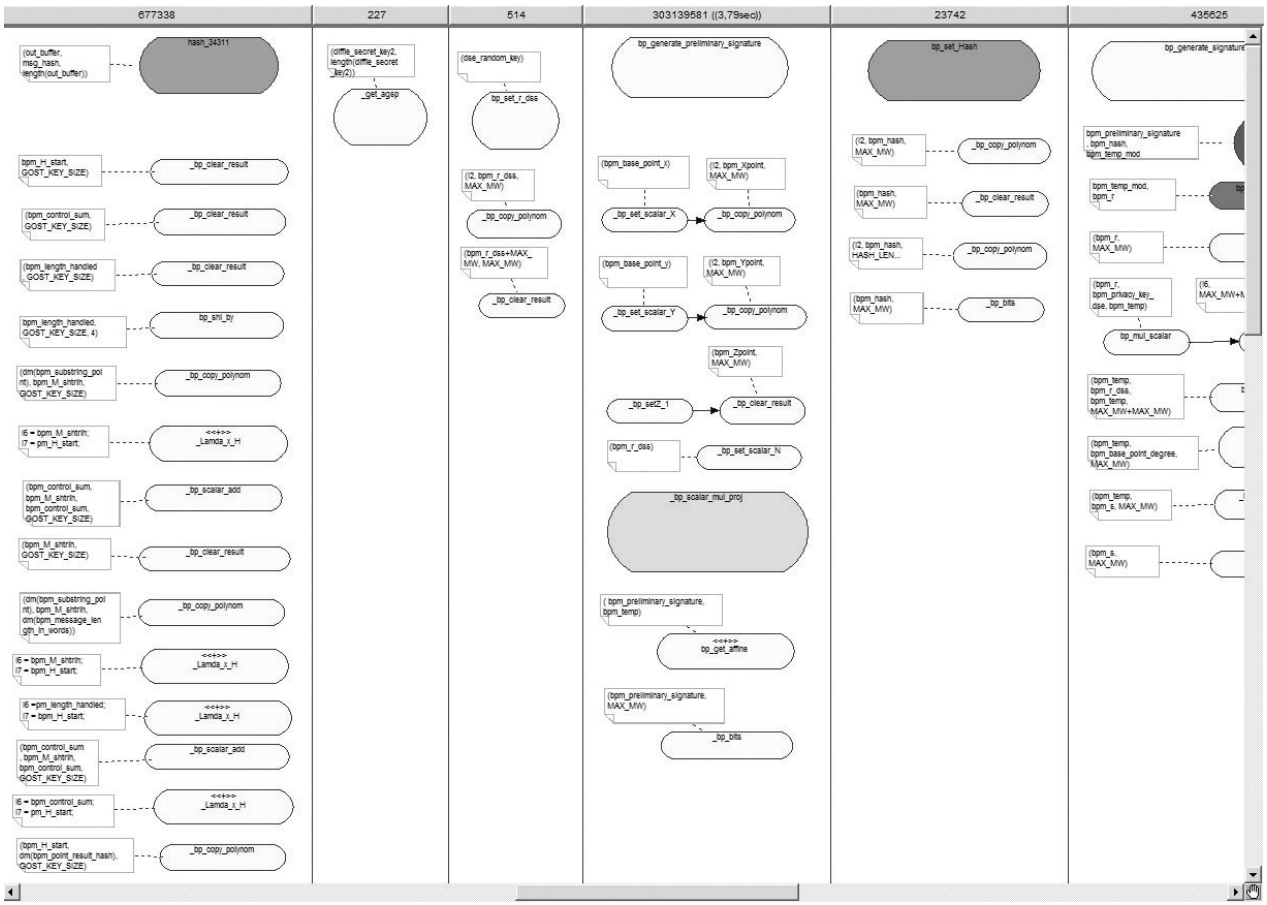

Рис. 2 Activity диаграмма

Под оптимизацией понимают последовательность эквивалентных преобразований исходной программы, уменьшающих ее стоимость. Как набор, так и порядок выполнения этих преобразований зависят от того, что считается стоимостью программы. В качестве такой стоимости могут выступать, например, среднее время работы, объем кода и т.д. Эффективность оптимизации также зависит от отношения эквивалентности и от размера участка экономии, на котором эта оптимизация проводится. За счет оптимизации невозможно добиться существенного улучшения *алгоритма* программы, можно только говорить об улучшении *реализации* этого алгоритма. В удачных случаях оптимизация может ускорить программу в несколько раз.

Предлагается использовать следующие способы оптимизации программы: кодовый, алгоритмический и системный.

Под кодовым способом оптимизации следует понимать оптимизацию арифметических операций, минимизацию количества команд пересылки (особенно между процессором и памятью), наиболее эффективное использование регистров.

Основа оптимизации арифметических операций это мультифункциональные инструкции – объединение арифметической операции с передачей данных. Проанализировав программный код мы заметили, что мультифункциональные инструкции используются здесь по максимуму.

После реализации данного метода было проведено измерение времени выполнения программы. Результаты показали, что используя этот метод, мы ускорили программу лишь на ≈4%

Было решено перейти к следующему методу – алгоритмическому.

Рассмотрим некоторые методы алгоритмической оптимизации программного кода.[2]

- **Развертка циклов**. Суть этого преобразования заключается в том, что тело цикла дублируется *n* раз, а число повторений соответственно сокращается во столько же раз. Число *n* называется коэффициентом развертки цикла.
- **Слияние циклов**. Два расположенных последовательно цикла можно слить, если они имеют одинаковое число итераций и отсутствуют зависимости по данным, препятствующие объединению. Если тела сливаемых циклов не зависят друг от друга появляется возможность спланировать параллельное выполнение команд, относящихся к разным циклам.
- **Разбивка циклов**. – преобразование обратное слиянию. Его целесообразно осуществлять, например, если тело цикла слишком длинное, и имеющееся число регистров недостаточно для размещения всех используемых в теле цикла переменных. Благодаря разбивке цикла можно избежать дефицита регистров и выталкивания значений в память.

Основа оптимизации арифметических операций это Multifunctional Instructions – объединение арифметической операции с передачей данных. Например:
AR = AX0 + AY0, AX0 = DM (I0, M0). Теперь две операции (AR = AX0 + AY0; и AX0 = DM (I0, M0);) будут выполняться как одна (за один такт). Более детально Multifunctional Instructions показаны на рисунке(multifunctional1.bmp).

Если говорить об алгоритмическом подходе к оптимизации программного кода, то нужно сказать, что он применяется для кардинального изменения хода выполнения любой части проекта. Например, при вычислении площади под заданной кривой можно использовать метод «трапеций» или метод «треугольников»: один из них, возможно, даст лучшие результаты по критерию быстродействия.

Применив вышеперечисленные методы алгоритмического преобразования и произведя измерения, мы увидели, что программа ускорилась еще на ≈11%

После всех проведенных нами работ над этим проектом, мы получили общий прирост около ≈15% в быстродействии. Время входа в закрытую сессию все еще было слишком велико, чтобы не отказаться от последнего, наиболее кардинального метода оптимизации – системного.

Системный подход заключается в изменении всего или части общего алгоритма работы программы. Его используют в тех случаях, когда оптимизация арифметических операций и алгоритма не дает нужных результатов.

Было решено, что необходимо уменьшить количество вычислений публичного ключа в программном модуле. Это возможно, если сгенерированные ключи будут храниться во флэш-карте, откуда их всегда можно быстро прочитать.

Далее, реализовав этот подход, мы увидели, что при доставании ключа из флеш – все время, необходимое для организации защищенного канала связи составило теперь ≈5 секунд, что в ≈4 раз быстрее, чем до оптимизации.

Итак, проведя данную работу, Мы увидели, что в нашем случае наибольший прирост в быстродействии программы дает наиболее кардинальный метод оптимизации – системный.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. *DSP LAB*. Введение в DSP, http://dsp.iu4.bmstu.ru

2. *Шумаков С.М*. Обзор методов оптимизации кода для процессоров с поддержкой параллелизма на уровне команд. сентябрь 2003,
http://www.citforum.ru/programming/digest/20030430/