Разработка преобразования code factoring для ПЛИС

А.П. Баглий

Южный Федеральный Университет

Аннотация. Приводятся доводы в пользу высокоуровневого анализа текста на частичные дубликаты для оптимизации синтеза на ПЛИС из программы на языке С. Описываются возможные подходы к такому анализу. Приводится выбранный способ поиска важных фрагментов с фильтрацией результатов через статическое профилирование. Для поиска частично повторяющихся фрагментов используется комбинация методов анализа текста и поиска шаблонов в дереве программы после синтаксического разбора. Статическое профилирование позволяет выбрать фрагменты кода, которые выгодно выполнять на ПЛИС. Представлены результаты анализа кода некоторых тестовых программ. На основе сделанного анализа выбраны требования к разрабатываемому преобразованию программ code factoring для внедрения в состав компилятора с высокоуровневого языка на ПЛИС.

Ключевые слова: ПЛИС, code factoring, дублирование кода, компилятор, высокоуровневый синтез, оптимизирующая компиляция.

Developing code factoring transformation for FPGA

A.P. Bagly

Southern Federal University

Abstract. This work proposes using high-level source analysis for finding partial code duplicates to optimize FPGA synthesis from C language programs. Possible approaches to such analysis are described. Technique to filter useful fragments using static profiling is presented. Combination of program text analysis and syntax tree template search methods is used to find partially repeated code fragments. Static profiling allows to pick code fragments that would be beneficial to run on FPGA. Results of code analysis for a set of test programs are shown. Based on presented analysis results requirements for high-level code factoring program transformation are selected for this transformation to be added to high-level language compiler for FPGA.

Keywords: FPGA, code factoring, code duplication, compiler, high-level synthesis, optimizing compilation.

Введение

Синтез программ для ПЛИС по программам на высокоуровневом языке программирования (например, C) – это в общем случае сложная задача, которая на данный момент не решается полностью автоматически.

На основе Оптимизирующей распараллеливающей системы [1] разрабатывается компилятор на ПЛИС с языка C [2]. Поскольку ОРС обладает набором оптимизирующих преобразований программ, типичных для современных компиляторов, то интересной выглядит идея их использования для улучшения параметров получаемых схем, например их размера (для возможности размещения на ПЛИС).

Целью высокоуровневых оптимизирующих преобразований программ обычно являются:

1. Уменьшение времени работы программы
2. Уменьшения объема ее кода
3. Ускорение работы программы с памятью

Эти параметры для каждой программы влияют друг на друга нетривиальным образом. В это время, оптимизация синтеза на ПЛИС может включать большее число параметров, например:

1. Число задействованных элементов разных типов
2. Максимальная частота работы схемы
3. Задействованная пропускная способность памяти
4. Энергопотребление

Каким образом преобразования высокоуровневой программы повлияют на параметры созданной по ней схемы для ПЛИС заранее не известно и зависит во многом от используемых инструментов.

Постановка задачи

Возможно провести прямые параллели между оптимизацией параметров схем для ПЛИС и оптимизацией программ компиляторами на примере встраивания процедур в место вызова (инлайнинга). Частая задача для компиляторов – поиск баланса между инлайнингом и использованием вызовов подпрограмм в неизменном виде. Инлайнинг устраняет накладные расходы на вызов подпрограммы, но увеличивает объем кода за счет его дублирования. Рост объема кода может привести к промахам в кэше инструкций и уменьшению производительности. Методам выбора функций для инлайнинга посвящено множество работ, например [3], где кроме этого используется обратное преобразование – outlining, которое преобразует участок кода в функцию.

Дополнительная проблема, усложняющая решение задачи по инлайнингу и аутлайнингу кода – наличие полностью или частично повторяющихся фрагментов кода, которые не выделены в отдельные подпрограммы.

При переходе к ПЛИС указанные проблемы сохраняются, но проблема повторяющихся фрагментов кода встает более остро, так как они напрямую влияют на размер полученной схемы. Поэтому стоит задача проверки подхода оптимизации параметров ПЛИС за счет поиска частично повторяющихся фрагментов кода и их преобразований.

Целью исследований является проверка актуальности создания преобразования code factoring для высокоуровневого внутреннего представления, аналогичного преобразованию, используемому в современных компиляторах в низкоуровневом представлении [4].

Методы

Если стоит задача уменьшения объема используемых ресурсов ПЛИС, то потенциальные источники кода, подлежащего оптимизации - это:

1. Часто вызываемые функции,
2. Полностью продублированные фрагменты кода
3. Частично продублированные фрагменты кода, и отдельные выражения

В случае 3 возможно использование ранее описанных методов [5].

Полное или частичное дублирование исходного кода программ - неизбежное явление. Дублированию могут подвергаться целые файлы, функции или отдельные фрагменты кода. Исследование [6] демонстрирует возможность анализа очень большого объема исходного кода на дублирование.

В применении к высокоуровневому синтезу на ПЛИС может быть интересно частичное или полное дублирование **коротких участков кода**, выполняющих относительно **большой объем вычислений**.

Выбор участков кода для отображения на ПЛИС может производиться по результатам профилирования программы [7] или статического анализа [8] (статического профилирования). Профилирование на реальных входных данных дает точные данные о времени, уходящем на выполнение каждой функции или оператора программы, но оно может быть ресурсоемким и не всегда доступным способом анализа времени выполнения. Статическое профилирование может дать удовлетворительные результаты для некоторого подкласса программ за меньшее время.

Предлагаемое решение

Для проверки перспективности предлагаемого подхода были скомбинированы методы поиска похожих фрагментов [9], [10] и статического профилирования. Статическое профилирование используется нами для

отсеивания неинтересных для оптимизации фрагментов из тех, которые были найдены при анализе текста на дубликаты.

Для проверки возможного эффекта от оптимизаций целесообразно проанализировать исходный код программ, которые предназначены для отображения на ПЛИС средствами высокоуровневого синтеза. Для синтеза на ПЛИС существуют несколько общепринятых наборов таких программ (бенчмарков), специально собранных для оценки эффектов от новых оптимизаций. Например, наборы CHStone [11] и Rosetta [12] содержат достаточный объем исходного кода программ из разных предметных областей.

Результаты

Для проверки разрабатываемых методов был использован исходный код программ бенчмарка CHStone [13]. Для поиска участков кода, представляющих интерес для оптимизации, использовался простой порог на число вычислительных операций (арифметических, логических) во фрагменте кода.

Программа	всего строк	фраг-мен-тов	% похожих строк	похожих строк	% похожих строк всего	Похожих строк всего
3d-rendering	3836	9	2%	72	86%	3300
BNN	14203	358	27%	3880	46%	6562
digit recognition	433	5	5%	21	8%	34
Face detection	1490	14	7%	102	54%	806
harness	753	12	22%	167	27%	206
Optical flow	2404	46	21%	501	38%	930
Spam filter	2541	9	5%	132	83%	2223

Таблица 1. Результаты анализа текста программ CHStone. Помечены результаты по похожим фрагментам больше порога.

Использованный метод позволяет быстро проанализировать большой объем кода, что даст возможность точнее оценить перспективы для оптимизации за счет анализа большого числа программ, ускоряемых с помощью ПЛИС.

С помощью стандартного профилирования программ бенчмарка CHStone с использованием профилировщика gprof [14] было обнаружено, что в 50% случаев «горячие участки» программы включали в себе найденные повторяющиеся фрагменты, например, в программе AES:

```
x = (statement[(i + 1) % 4 + j * 4] << 1);
if ((x >> 8) == 1)
  x ^= 283;
x = (x << 1);
if ((x >> 8) == 1)
  x ^= 283;
x ^= statement[(i + 1) % 4 + j * 4];
x = (x << 1);
if ((x >> 8) == 1)
```

Был найден следующий фрагмент кода, повторяющийся 4 раза с небольшими модификациями (другими значениями констант).

После того, как такие фрагменты кода были найдены в исходной программе, возможно их преобразование в функцию. Это можно сделать в 2 этапа:

1. Преобразовать все частично продублированные фрагменты в полностью продублированные и добавленный к ним код, присваивающий значения вспомогательным переменным. Для этого можно использовать методы из [4]
2. Преобразовать все копии продублированных фрагментов в вызовы новой функции, что можно сделать автоматически с помощью OPC.

Эти преобразования программы призваны помочь работе инструментов высокоуровневого синтеза для ПЛИС.

Перспективы

Среди современных подходов к автоматическому программированию ПЛИС стоит выделить, кроме средств высокоуровневого синтеза, средства параллельной разработки программного и аппаратного обеспечения (co-design), которые могут занимать промежуточное положение между полностью автоматическим синтезом схемы на ПЛИС по высокоуровневой программе и ручным программирование ПЛИС на низкоуровневом языке. Co-design позволяет, например, использовать готовые архитектуры процессоров в виде схемы на ПЛИС, к которым добавляются новые инструкции, поддерживаемые компилятором, для ускорения работы целевой программы. Среди таких средств можно выделить систему [15] использующую упрощенную VLIW-архитектуру. Возможно применение описанных выше методов для поиска участков кода - кандидатов в дополнительные инструкции для подобных архитектур.

Выводы

Проверены методы быстрого анализа большого объема исходного кода для проверки потенциала оптимизации схем ПЛИС за счет частично или полностью продублированных фрагментов кода. Полученные результаты свидетельствуют об актуальности разработки преобразования code factoring в высокоуровневом внутреннем представлении для использования в составе компилятора с языка C.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект 18-37-00179\18.

Литература

1. “Оптимизирующая распараллеливающая система” 2018. www.ops.rsu.ru. [дата доступа: 28-May-2018].
2. B. Y. Steinberg, A. P. Bugliy, D. V. Dubrov, Y. V. Mikhailuts, O. B. Steinberg, and R. B. Steinberg, “A Project of Compiler for a Processor with Programmable Accelerator,” in *Procedia Computer Science*, 2016, vol. 101, pp. 435–438.
3. P. Zhao and J. N. Amaral, “Function Outlining and Partial Inlining,” *17th Int. Symp. Comput. Archit. High Perform. Comput.*, pp. 101–108, 2005.
4. G. Lóki, Á. Kiss, J. Jász, and Á. Beszédes, “Code Factoring in GCC,” *Proc. 2004 GCC Dev. Summit*, no. June, 2004.
5. А. П. Баглий, Д. В. Дубров, Б. Я. Штейнберг, and Р. Б. Штейнберг, “Повторное использование ресурсов при конвейерных вычислениях,” in *Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции*, 2017, с. 43–46.
6. C. V. Lopes *et al.*, “DéjàVu: a map of code duplicates on GitHub,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 1–28, 2017.
7. J. G. Tong, “Software profiling for an FPGA-based CPU core,”, thesis, University of Windsor 2007.
8. С. В. Полуян, “Профилирование и его применение в диалоговом оптимизирующем распараллеливателе,” *Научный сервис в сети Интернет: суперкомпьютерные центры и задачи: Труды Международной суперкомпьютерной конференции*, 2010, pp. 652–653.
9. D. Grune and M. Huntjens, “Het detecteren van kopieen bij informatica-practica,” *Informatie*, vol. 31, no. 11. pp. 864–867, 1989.
10. F. Ma, “On the Study of Tree Pattern Matching Algorithms and Applications,” University Of British Columbia, 2004.
11. “CHStone benchmark for FPGA high-level synthesis tools,” 2018. http://www.ertl.jp/chstone/. [дата доступа: 29-May-2018].
12. Y. Zhou *et al.*, “Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs,” pp. 269–278, 2018.

13. Y. Hara, H. Tomiyama, S. Honda, and H. Takada, “Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis,” *J. Inf. Process.*, vol. 17, pp. 242–254, 2009.
14. S. L. Graham, P. B. Kessler, and M. K. Mckusick, “gprof: a Call Graph Execution Profiler,” in *ACM SIGPLAN Notices*, 1982, vol. 17, no. 6, pp. 120–126.
15. O. Esko, P. Jääskeläinen, P. Huerta, C. S. De La Lama, J. Takala, and J. I. Martinez, “Customized exposed datapath soft-core design flow with compiler support,” *Proc. - 2010 Int. Conf. F. Program. Log. Appl. FPL 2010*, pp. 217–222, 2010.

References

1. “Optimiziruiushchaia rasparallellivaiushchaia sistema” 2018. www.ops.rsu.ru. [date of access: 28-May-2018].
2. B. Y. Steinberg, A. P. Bugliy, D. V. Dubrov, Y. V. Mikhailuts, O. B. Steinberg, and R. B. Steinberg, “A Project of Compiler for a Processor with Programmable Accelerator,” in Procedia Computer Science, 2016, vol. 101, pp. 435–438.
3. P. Zhao and J. N. Amaral, “Function Outlining and Partial Inlining,” 17th Int. Symp. Comput. Archit. High Perform. Comput., pp. 101–108, 2005.
4. G. Lóki, Á. Kiss, J. Jász, and Á. Beszédes, “Code Factoring in GCC,” Proc. 2004 GCC Dev. Summit, no. June, 2004.
5. A. P. Baglii, D. V. Dubrov, B. Ia. Shteinberg, and R. B. Shteinberg, “Povtornoe ispolzovanie resursov pri konveiernykh vychisleniiakh,” in Nauchnyi servis v seti Internet: trudy XIX Vserossiiskoi nauchnoi konferentsii, 2017, s. 43–46.
6. C. V. Lopes et al., “DéjàVu: a map of code duplicates on GitHub,” Proc. ACM Program. Lang., vol. 1, no. OOPSLA, pp. 1–28, 2017.
7. J. G. Tong, “Software profiling for an FPGA-based CPU core,”, thesis, University of Windsor 2007.
8. S. V. Poluian, “Profilirovanie i ego primenenie v dialogovom optimiziruiushchem rasparallelivatele,” Nauchnyi servis v seti Internet: superkompiuternye tsentry i zadachi: Trudy Mezhdunarodnoi superkompiuternoi konferentsii, 2010, pp. 652–653.
9. D. Grune and M. Huntjens, “Het detecteren van kopieen bij informatica-practica,” Informatie, vol. 31, no. 11. pp. 864–867, 1989.
10. F. Ma, “On the Study of Tree Pattern Matching Algorithms and Applications,” University Of British Columbia, 2004.
11. “CHStone benchmark for FPGA high-level synthesis tools,” 2018. http://www.ertl.jp/chstone/. [data dostupa: 29-May-2018].
12. Y. Zhou et al., “Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs,” pp. 269–278, 2018.
13. Y. Hara, H. Tomiyama, S. Honda, and H. Takada, “Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis,” J. Inf. Process., vol. 17, pp. 242–254, 2009.

14. S. L. Graham, P. B. Kessler, and M. K. Mckusick, “gprof: a Call Graph Execution Profiler,” in ACM SIGPLAN Notices, 1982, vol. 17, no. 6, pp. 120–126.
15. O. Esko, P. Jääskeläinen, P. Huerta, C. S. De La Lama, J. Takala, and J. I. Martinez, “Customized exposed datapath soft-core design flow with compiler support,” Proc. - 2010 Int. Conf. F. Program. Log. Appl. FPL 2010, pp. 217–222, 2010.