Динамическое профилирование программы для системы LLVM

А.И. Аветисян, К. Ю. Долгорукова; Ш. Ф. Курмангалеев,
arut@ispras.ru, unerkannt@ispras.ru; kursh@ispras.ru

Аннотация. При построении системы компиляции для языков общего назначения, учитывающей специфические особенности целевой аппаратуры и наиболее вероятный сценарий использования, необходимо применять методы динамической и адаптивной оптимизации. Исследование таких методов удобно проводить в компиляторной инфраструктуре LLVM. Тем не менее, в настоящий момент LLVM не поддерживает динамический сбор профиля и перекомпиляцию, а также содержит лишь одно преобразование, использующее данные профиля. В рамках данной работы, для LLVM была предложена и реализована система сбора профиля аппаратных прерываний и алгоритм, корректирующий переоценку профиля, а также несколько оптимизирующих преобразований с учетом профиля. Выполнена интеграция сбора профиля и динамического компилятора LLVM, что позволило сохранять качество программ при их переносе на другую архитектуру.

Ключевые слова: профилирование; динамическая компиляция;

1. Введение

В современных компиляторах для языков общего назначения (Си/Си++) основными видами выполняемой оптимизации является статическая оптимизация и оптимизация с использованием профиля программы. При статической оптимизации генерируется одна версия объектного кода программы, которая оптимизирована для некоторой “средней” машины данной процессорной архитектуры, а для учета специфических особенностей конкретной аппаратуры необходимо создавать и поддерживать несколько бинарных версий. Например, использование данных профиля позволит повысить качество планирования.

При оптимизации с учетом профиля программы производится сбор профилей на заданном множестве наборов входных данных и учет полученной статистики, при этом ответственность за подбор входных данных ложится на программиста. Отметим, что статистика на пользовательских наборах данных может значительно отличаться, что в некоторых случаях приводит к замедлению программы. Кроме того, этот подход связан со значительными накладными расходами на сбор профилей и подбор параметров компилятора.

Предлагаемым решением является применение методов динамической (JIT-компиляции) и адаптивной оптимизации в компиляторах для Си/Си++. Динамическая оптимизация во время работы программы имеет то преимущество, что программа оптимизируется на конкретном наборе входных данных для данного конкретного запуска. Собранная статистика используется только для оптимизации данного запуска, а разные запуски программы могут приводить к различным оптимизациям.

Для реализации динамических оптимизаций в компиляторе необходимо распространение программы в объектных файлах, содержащих внутреннее представление, сохраняющее информацию высокого уровня и позволяющее проводить динамический мониторинг, профилирование и оптимизацию программы. Это позволяет сократить затраты на распространение и поддержку программы (достаточно поддерживать одну версию программы, при сборке которой применялись лишь машинно-независимые оптимизации). Окончательная специализация будет происходить автоматически на машине пользователя. При этом значительно упрощается процесс разработки без потерь в производительности получаемой программы, так как не требуется выполнять подбор «хорошего» набора входных данных для сбора статистики.

В качестве основы для построения системы динамической оптимизации программ на языках общего назначения удобно выбрать LLVM [1] – компиляторную инфраструктуру с открытыми исходными кодами на языке Си++. В рамках этого проекта представлены: статический компилятор, компоновщик, виртуальная машина, JIT-компилятор. Функционирование системы обеспечивается единым внутренним представлением, которое может быть представлено в текстовом виде, в виде структур данных в оперативной памяти, а также в двоичном виде как бит-код. Этот бит-код может быть сохранен в промежуточных объектных файлах для дальнейшей оптимизации, в том числе динамической. При этом возможно использовать все предоставляемые LLVM возможности по обработке внутреннего представления (включая различные анализы, трансформации и т.п.). Поэтому инфраструктура LLVM предоставляет удобную базу для исследований по динамической оптимизации программ. Данная работа является развитием идей, предложенных в [2].

К сожалению, в LLVM возможности работы с профилем реализованы в минимальном объеме: из существующих оптимизаций профиль используется только при переупорядочивании базовых блоков, а работа с данными профиля ведется лишь при статической компиляции в виде обычной двухпроходной схемы компиляции. Поэтому необходимой задачей является разработка такой схемы профилирования, которая, во-первых, может применяться динамически, а во-вторых, достаточно эффективна. Мы предложили схему профилирования на основе частичного профилирования, или сэмплинга (sampling). Идея этого метода состоит в записи лишь части данных (проб), выбрав соотношение между записываемыми и пропускаемыми данными

таким образом, чтобы, с одной стороны, профилирование выполнялось с малыми затратами (замедление программы не превосходило бы нескольких процентов), а с другой стороны, полученные данные достаточно полно отражали бы реальную картину поведения программы. Предложенная схема профилирования реализована на основе утилиты Oprofile, кроме того, реализован алгоритм «подгонки» полученного динамического профиля таким образом, чтобы в результате данные профилирования могли быть использованы напрямую оптимизациями в компиляторе LLVM. Мы применили полученные данные в оптимизации перемещения базовых блоков и смогли получить выигрыш по сравнению с программой, собранной без учета профиля.

2. Динамическое профилирование программы для системы LLVM

2.1. Изменения, внесенные в систему сборки LLVM

Так как в LLVM не предусмотрены средства прозрачного, автоматического получения бит-кода с учетом зависимостей между модулями. А также отсутствует поддержка динамического связывания модулей с бит-кодом. Для программ система сборки которых основана на использовании утилит configure и make, была предложена следующая схема двухэтапной компиляции (см.рис 1).

На первом этапе происходит генерация установочного пакета, содержащего в себе модули бит-кода и скрипты автоматического развертывания. Вместо использования оригинальной утилиты configure предлагается использовать специальную обертку configure-proxy, осуществляющей необходимые подстановки и вызывающую оригинальную утилиту, результатом работы является скрипт make.sh, принцип работы которого аналогичен. Для компиляторов реализованы обертки, основанные на том же подходе. Помимо этого необходимые изменения внесены в компиляторы переднего плана, и линкер, эти изменения позволяют отследить зависимости между отдельными модулями программы. После окончания компиляции программы с помощью скриптов пост-обработки происходит, создание инсталляционного пакета на основе сгенерированных ранее зависимостей. Инсталляционный пакет содержит файлы с бит-кодом. Файлы, помеченные как зависимости на этапе постобработки, и скрипты компиляции и установки. На втором этапе во время установки программы существует две альтернативы. Первый вариант установки, это статическая компиляция, второй использование динамической компиляции.

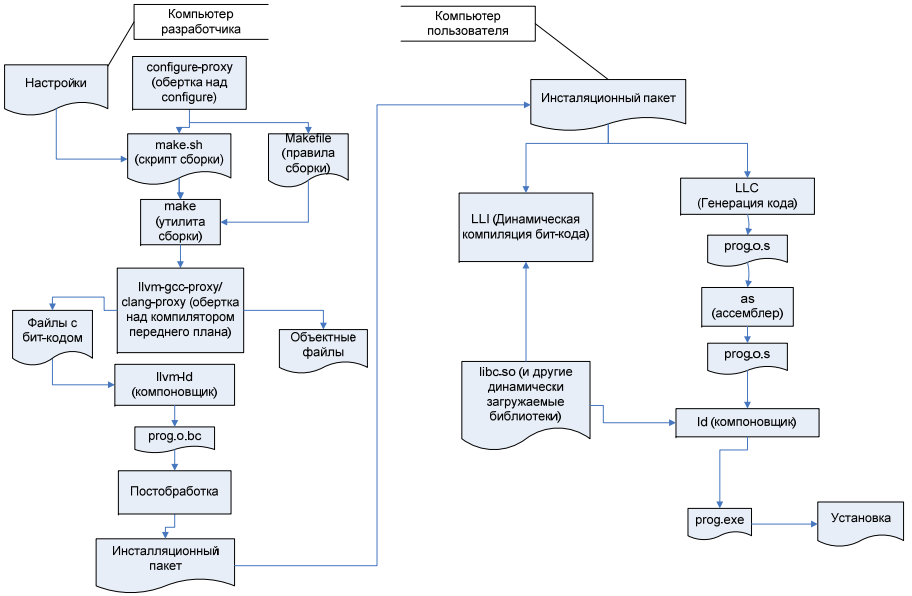

Рис. 1. Схема двухэтапной компиляции.

В обоих случаях все происходит прозрачно для пользователя. Описанный подход позволяет при компиляции учитывать специфику конкретной архитектуры и генерировать более оптимальный код. А так же позволяет использовать динамическую компиляцию и полностью автоматизировать процесс получения бит-кода и определения как статических, так и динамических зависимостей между модулями программы, как содержащими бит-код, так и бинарный код, для программ состоящих более чем из одного модуля, и написанных на языках общего назначения c/c++.

2.2. Существующие подходы к сбору профиля

Существует два способа получить статистику во время исполнения программы: инструментировать ее, - то есть вставлять счетчики в код при генерации или еще в промежуточное представление программы, - либо собирать выборочным профилировщиком аппаратных прерываний. В первом случае код увеличивается в размерах и сильно замедляется работа программы. Как показали результаты тестирования на наборе тестов aburto, при инструментировании ребер между базовыми блоками, в худшем случае происходит замедление в 2-4 раза. Эвристики инструментирования помогают опустить эту разницу лишь до 15-30 процентов [3]. Во втором же случае программа замедляется всего на 2-8 процентов [4], поэтому в этой работе будет рассмотрен именно этот метод - профилирование сэмплингом.

При профилировании компилируемых на лету программ возникает следующая проблема: профилировщик видит только запускаемые программы и таблицу символов этих программ, а символы генерируемого кода ему не видны, и информация о генерированном коде помещается в анонимное пространство имен. Для того, чтобы знать, какая функция исполняется JIT-компилятором, нужно задать способ отображения символов на виртуальные адреса. В виртуальных машинах Java к сэмплингу применяется следующий подход: компилятор записывает в файл виртуальные адреса начала и конца динамически загружаемых методов или участков кода (как правило, это циклы) при их загрузке или выгрузке. Для записи информации о символах используется библиотечный интерфейс JIT-агентов, как правило, включающий в себя функции оповещении о загрузке и выгрузке методов [5].

После получения сэмплов профилировщик сопоставляет полученные в виде пар (вирт. адрес, значение счетчика) выборки с файлами, записанными агентами, и распределяет статистику по символам, записывая ее в файл, который потом передается на считывание JIT-компилятору. Но данный подход не слишком удобен для языков общего назначения, где логика динамически загружаемых классов неприемлема. Но, тем не менее, LLVM предусматривает возможность раздельной компиляции и исполнения функций, поэтому механизм работы с агентом может быть использован. Но в этой работе был использован другой подход. За основу профилировщика был взят Oprofile 0.9.6, в котором механизм записи сэмплов в файл был заменен общением с JIT-компилятором через FIFO файл.

2.3. Описание предлагаемого подхода к сбору профиля

Поскольку предлагаемая система использует в качестве профилировщика утилиту OProfile, опишем принцип ее работы. Oprofile состоит из двух частей – сборщика информации, реализованного как модуль к ядру Linux, и клиентской части, обрабатывающей полученную информацию. Сборщик информации считает количество раз срабатывания некоторого счетчика (например, таймера или события промаха кэша), при переполнении этого счетчика происходит сброс информации в буфер сборщика и далее на жесткий диск. Клиентская часть обрабатывает записанную информацию, строит необходимые графы вызовов и т.п.

Схема работы OProfile была применена для реализации динамического профилирования в LLVM см. рис.2. Система состоит из следующих частей: JIT-компилятор считывает из файла бит-код, компилирует, запускает его и инициирует передачу статистики; модуль ядра Linux собирает статистику о счетчиках на машине и сбрасывает ее в буфер; демон-профилировщик считывает данные из буфера, распознает и записывает их в FIFO-каналы.

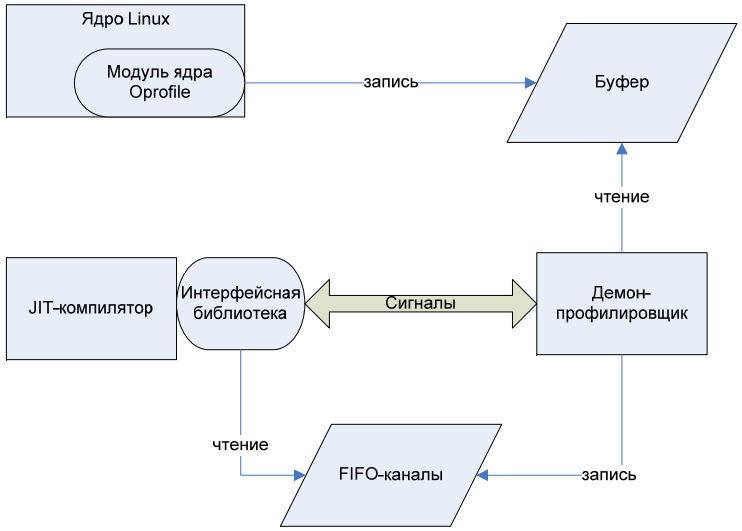

Рис. 2. Взаимодействие компонентов системы.

Демон является отдельной программой, считывающей данные из буфера, созданного модулем OProfile. Основанный на «родном» демоне Oprofile, он обрабатывает только данные, необходимые для Just-In-Time компилятора. Прежде всего, во входных параметрах, помимо тех, что есть у демона OProfile, он принимает имя образа запущенного JIT-компилятора и идентификатор или несколько идентификаторов процессов, в которых будет работать профилируемый код. Так как код генерируется и исполняется самим компилятором, он не виден системе и не может быть идентифицирован, поэтому помещается в анонимное пространство имен. Демон выбирает помеченные как анонимные данные для конкретных процессов; когда к нему приходит сигнал с запросом сбросить собранную информацию, он записывает ее в канал и продолжает работу до получения сигнала о завершении работы. Компилятор, производящий оптимизации во время исполнения (или Just-In-time compiler), представляет собой оптимизирующий компилятор бит-кода LLVM, который был сгенерирован компилятором переднего плана LLVM-GCC или Clang с языка Си или Си++. JIT-компилятор может, как интерпретировать код, так и компилировать с различными уровнями оптимизации. Нами была выполнена реализация библиотеки, осуществляющей интерфейс между демоном-профилировщиком и компилятором. В отличие от виртуальных машин Java, библиотека не пишет информацию о сгенерированных символах в файлы и не перекладывает ответственность за отображение символов на адреса утилитам Oprofile, а общается с демоном посредством сигналов и получает данные от него из

FIFO-каналов, формируя базу данных профиля. Отображение на виртуальные адреса производится в процессе работы JIT-компилятора.

3. Переоценка профиля

У полученной выборки погрешность будет довольно велика ввиду задержки срабатывания сигнала и записи в счетчик. Чтобы получить наиболее корректный профиль программы, мы должны убедиться, что, представив граф потока управления сетью, получим ее такой, что на ней будет выполняться теорема о сохранении потока. Для этого мы применили к профилю программы алгоритм переоценки профиля [6] [7]:

1. По имеющемуся графу потока управления и профилю строится расчетная сеть.
2. Далее на этой сети ищется максимальный поток минимальной стоимости.
3. В соответствии с функцией максимального потока распределяются новые веса в профиле.

Алгоритм принимает на вход структурированный по базовым блокам профиль, а также информацию статического анализа циклов: она будет необходима для предварительной оценки весов ребер графа потока управления функции. Так как профиль собирался для инструкций, суммарное количество сэмплов на базовом блоке не есть частота исполнения блока, поэтому оцениваем последнюю как среднее по инструкциям:

$$BB_{count} = \frac{\sum_{i=1}^{N_{statements}} IR.count_i}{N_{statements}}$$

где $N_{statements}$ - число инструкций в базовом блоке, $IR.count_i$ число сэмплов для i-й инструкции. Для облегчения работы алгоритму также предварительно удаляются все недостижимые пути графа.

3.1. Построение расчетной сети

Необходимость построения дополнительной расчетной сети обусловлена особенностями применяемого к ней впоследствии алгоритма поиска максимального потока минимальной стоимости:

1. в сети не должно быть циклов длины, меньшей 3,
2. должны присутствовать исток и сток.

3.2. Построение функции максимального потока минимальной стоимости

Для построения функции максимального потока был использован алгоритм Эдмондса-Карпа - одна из реализаций алгоритма Форда-Фалкерсона, уточняющая, что при построении увеличивающего пути используется

алгоритм поиска в ширину. Поэтому он является конечным для всех действительных значений пропускной способности ребер, в отличие от самого алгоритма Форда-Фалкерсона, который может работать бесконечно при произвольном выборе пути на каждом шаге [8]. После построения максимального потока строится остаточная сеть, в которой по алгоритму Клейна удаляются циклы отрицательной стоимости [9].

3.3. Пересчет профиля

По сети максимального потока минимальной стоимости однозначно восстанавливается исходный граф потока управления: по ребрам, построенным в результате преобразования вершин, мы восстанавливаем профиль вершин. Ребра, соответствующие исходному графу, не удаляются алгоритмом, т.к. им была назначена бесконечная максимальная пропускная способность, значит, тоже могут быть восстановлены. Теперь профиль удовлетворяет условию сохранения потока, т.к. лишний поток был «слит» в источник или в сток с помощью балансировки в процессе работы алгоритма Эдмондса-Карпа. Таким образом, новыми значениями счетчиков сэмплов станут значения потоков на ребрах сети максимального потока.

3.4. Сохранение собранного профиля в формате LLVM

После пересчета, у нас есть нормализованный профиль, его можно сохранить на диск, но нельзя применить непосредственно к имеющемуся бит-коду. Несоответствие обусловлено тем, что перед запуском кодогенерации LLVM запускает несколько проходов изменяющих машинно-независимое внутреннее представление. Таким образом, для обеспечения возможности использования профиля собранного сэмплингом, следует выполнить все оптимизации, запускаемые над машинно-независимым представлением, запустить требуемые оптимизации использующие профиль и сохранить оптимизированный бит-код на диск. Описанный подход был реализован нами в текущей работе. Но в данный момент имеет некоторые ограничения, связанные тем, что если применяется динамическая компоновка нескольких модулей с бит-кодом, то профиль сохраняется для объединенного представления модулей в памяти, в будущем мы будем сохранять профиль отдельно для каждого модуля.

4. Оптимизации, использующие профиль

Имея профиль уже после работы одной или нескольких функций, мы можем точно определить не только степень «горячести» самой функции - то есть, частоту исполнения по сравнению с другими, - но и также относительную «горячесть» базовых блоков, узкие места внутри функции, которые можно оптимизировать прямо во время работы программы. К сожалению в настоящее время LLVM 2.9 имеет поддержку только одной оптимизации, использующей информацию о собранном профиле (Basic Block Placement). Далее

описываются некоторые оптимизации применение которых с учетом собранного профиля может привести увеличению быстродействия компилируемых программ.

4.1. Перемещение базовых блоков

Основная идея этой оптимизации заключается в том, чтобы расположить код часто исполняемых базовых блоков близко друг к другу, сокращая тем самым время вычисления адресов переходов между блоками, а в большинстве случаев позволяя даже исполнять «горячий код», состоящий из цепочки блоков, последовательно.

4.2. Разбиение блоков

Под разбиением понимается удвоение часто исполняемых базовых блоков графа потока управления, имеющих более одного исходящего ребра и более одного входящего. Суть алгоритма показана на рис. 3.

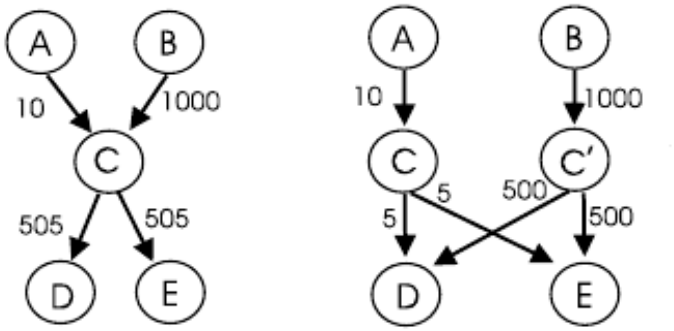

Рис. 3 Разбиение соединенных базовых блоков: слева - до разбиения, справа - после.

Данное преобразование не является оптимизирующим само по себе, но позволяет оптимизациям, основанным на анализе потока данных, - таким, как удаление загрузок, удаление границ массивов, замещение на стеке и пр., - работать более эффективно.

4.3. Встраивание функций

Очень важным преобразованием является встраивание тел функций вместо их вызовов. Встраивание, основанное только на статических эвристиках, на практике не всегда дает оптимальный код. Так, если функция будет встроена в редко исполняемые участки кода, но определенные статическим анализатором как возможно горячие, это не принесет практически никакой пользы. Если же основывать решение о встраивании функции в конкретный блок на

фактической частоте его исполнения, возможно увеличение производительности даже в сравнении со статическими компиляторами.

4.4. Развертка циклов

Имея динамический анализ, можно также разворачивать циклы, основываясь на частоте их исполнения. Например, часто исполняемые циклы разворачивать большее число раз для увеличения производительности, редко исполняемые - меньшее или же вообще не разворачивать, чтобы они занимали меньше памяти.

5. Текущие результаты

На данном этапе реализованы: инфраструктура, обеспечивающая прозрачную компиляцию в промежуточное представление LLVM, динамическое связывание модулей с промежуточным представлением во время выполнения программы, демон-профилировщик, библиотека работы с профилем, сохранение профиля в формате LLVM. Так как в LLVM JIT-компиляторе замещение на стеке пока находится в зачаточном состоянии, тестирование проходило в 2 прохода: сначала функции компилировались с уровнем оптимизации -O2 и сборкой профиля, а потом с использованием профиля компилировались заново. Результаты сравнивались с GCC 4.4.3 с опциями -O2 и стандартным интерпретатором lli с опциями -O2, тестирование проводилось на машине Intel Core2 Duo, E8500 3.00GHz, на ОС Linux Ubuntu 10.04 LTS. Результаты тестирования приведены в табл.1.

Тест	Без сбора профиля	сбор профиля	с учетом профиля	gcc	Единица измерения
Whetstone	3511,6	3219	3308	3594,8	MWIPS
Tfftdp	122,311	120,221	126,719	130,239	VAX_FFTs
Heapsort	4458,57	3585,35	4458,57	2907,72	MIPS
Nsieve	2968	2667,2	3022,9	2996	MIPS
Flops	880,362	854,833	1080,8889	1180,5825	MFLOPS
Fibonacci	0,72	0,97	0,72	0,86	secs

Тест		Без сбора профиля	сбор профиля	с учетом профиля	gcc	Единица измерения
Matrix Multi-plication	normal algorithm	0,48	0,52	0,49	0,48	secs
	temporary variable in loop	0,49	0,5	0,47	0,49	secs
	unrolled inner loop	0,49	0,51	0,49	0,47	secs
	pointers used to access matrices	0,48	0,49	0,48	0,48	secs
	transposed b matrix	0,14	0,14	0,12	0,14	secs
	interchanged inner loops	0,14	0,18	0,13	0,13	secs
Scimark		1042	920,21	1046,22	989,71	Composite Score

Табл. 1. Результаты тестирования.

Помимо этого, были проведены тесты по сбору профиля с выгрузкой его на диск, последующим применением к файлу бит-кода на примере программы sqlite и оптимизацией “Basic Block Placement”, что привело к ускорению работы программы на тестовом наборе данных на ~3%. Для проверки корректности проводилась, рекомпиляция бит-кода, без оптимизации использующей профиль, в этом случае мы наблюдали замедление порядка 0,5%-1%.

Список литературы

[1] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 61 pages.

[2] А. Белеванцев, Д. Журихин, Д. Мельник. Компиляция программ для современных архитектур. Труды Института системного программирования РАН, том 17, 2009 г. стр. 31-50.

[3] Omri Traub, Stuart Schechter, Michael D. Smith. Ephemeral Instrumentation for Lightweight Program Profiling.— 2000. www.eecs.harvard.edu/hube/publications/muck.pdf. Date retrieved: October 7, 2011.

[4] OProfile official website. http://oprofile.sourceforge.net.

[5] Java Virtual Machine Profling Interface documentation. http://download.oracle.com/javase/1.4.2/docs/guide/jvmpi/jvmpi.html. Date retrieved: October 7, 2011.

[6] Vinodha Ramasamy, Robert Hundt, Dehao Chen, Wenguang Chen. Feedback-Directed Optimizations in GCC with Estimated Edge Profiles from Hardware Event Proceedings of GCC Summit 2008. http://research.google.com/pubs/pub36576.html. Date retrieved: August 18, 2011.

[7] R. Levin, I. Newman, G. Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. Proceedings of the 3rd international conference on High performance embedded architectures and compilers.— HiPEAC’08.— Berlin, Heidelberg: Springer-Verlag, 2008.— Pp. 291–304.

[8] Zwick, U. The smallest networks on which the ford-fulkerson maximum flow procedure may fail to terminate. Theoretical Computer Science.— 1995.— Vol. 148.

[9] Levin, R. — Complementing Incomplete Edge Profile by applying Minimum Cost Circulation Algorithms.— Master’s thesis, University of Haifa, 2007.—Aug. http://www.cs.technion.ac.il/~royl/MscThesis_Final_Version_Submission.pdf. 38 pages