Обзор распараллеливающих компиляторов

А. И. Серебрянский
andrew.serebryansky@jetbrains.com

Г. А. Сыч
gennadiy.sych@gmail.com

В статье рассматриваются две наиболее функциональные и завершенные открытые платформы для разработки и исследования компиляторов, PROMIS и SUIF. Статья описывает возможности распараллеливания кода, предоставляемые каждой из платформ, а также проводится анализ их достоинств и недостатков.

Введение

В современном программном обеспечении все большую популярность завоевывают многопроцессорные и многоядерные системы. Такие системы могут давать прирост производительности при выполнении не только нескольких ресурсоемких приложений одновременно, но и одной последовательной программы, если она должным образом использует возможности, предоставляемые оборудованием. Как показывает практика, разработка эффективных параллельных программ требует очень высокой квалификации программистов из-за чрезмерной сложности и зачастую связана с возникновением труднообнаруживаемых ошибок. Таким образом, разработка средств, которые бы упрощали процесс распараллеливания, становится важной задачей системного программирования. Распараллеливающие компиляторы становятся сильным инструментом, который освобождает программистов от сложной задачи явного управления параллелизмом. Область параллельных вычислений достаточно широка, она включает в себя исследования различных типов параллелизма (таких как параллелизм на уровне

управления битовыми шкалами, на уровне инструкций, на уровне данных и на уровне подзадач), а также многочисленные параллельные архитектуры, которые различаются по использованию памяти (разделяемая память, распределенная память), топологии соединения вычислительных модулей (звезда, гиперкуб, кольцо, дерево) и по степени связанности процессоров. Каждый тип параллелизма и архитектуры предъявляет особые требования к компиляторам и их подсистемам.

В статье производится обзор современных открытых платформ (frameworks) для разработки и исследования компиляторов для архитектур с разделяемой памятью и реализующих параллелизм на уровне данных и инструкций. Такие платформы были созданы в Университете Иллинойса (Polaris, Parafrase2, PROMIS) и в Университете Стэнфорда (SUIF, National Compiler Infrastructure). Большинство подобных средств предоставляют две основные подсистемы: анализ зависимостей и реструктуризация циклов. Среди всех платформ мы выделили PROMIS и SUIF как наиболее завершенные, функциональные и документированные и постарались раскрыть возможности, предоставляемые этими системами, а также провели работу по исследованию их недостатков и недоработок.

1. Базовые средства распараллеливания

Прежде чем переходить к рассмотрению платформ, следует описать базовые средства, используемые в большинстве современных распараллеливающих компиляторов. Базовая информация, необходимая для успешного осуществления распараллеливающих преобразований, собирается в процессе анализа потока управления и анализа потока данных. Они участвуют в построении глобального «понимания» того, как работает программа и как она использует доступные ей ресурсы.

1.1. Символьный анализ (Symbolic Analysis)

Символьный анализ иногда также называют статическим анализом кода, так как он производится без реального выполнения программы (в отличие от динамического анализа). Сложность данного анализа варьируется от рассмотрения конкретных операций и определений до полного рассмотрения всего исходного кода программы. Результаты анализа используются от простых слу-

чаев (например, для определения потенциальных ошибок в коде) до сложных (например, для формального доказательства соответствия программы заданной спецификации). Существуют формальные методы статического анализа, такие как денотационная семантика, операционная семантика и абстрактная интерпретация.

1.2. Преобразования циклов

После того как вся необходимая информация собрана, производятся распараллеливающие преобразования. Основным инструментом систем, рассматриваемых в данной статье (а также большинства других подобных систем), являются преобразования циклов и их комбинации. В данном разделе будут рассмотрены основные преобразования циклов, которые применяются при распараллеливании. К таким преобразованиям можно отнести следующие.

1. **Обмен циклов (Loop Interchange).** Данное преобразование меняет местами два вложенных цикла с фиксированными итерациями (то есть оно меняет порядок итерационных переменных). Одной из главных целей преобразования является улучшение производительности кэшей при работе с элементами массивов. Оно позволяет предотвратить большое количество кэш-промахов, когда последовательно обрабатываемые элементы массива находятся в различных кэш-блоках.
2. **Разделение цикла (Loop Distribution, Fission).** Разделяет цикл на несколько циклов меньшего размера, каждый из которых содержит подмножество операторов из преобразуемого цикла. Данная трансформация может улучшить параллелизм, локальность данных (data locality) и векторизацию (vectorization).
3. **Слияние циклов (Loop Fusion).** Соединяет два цикла в один. Цикл, являющийся результатом такого преобразования, будет содержать все операторы из исходных циклов. Лишние операторы, такие как изменение итерационной переменной и проверки условий, будут убраны. Данное преобразование можно использовать для улучшения параллелизма на уровне инструкций, локальности данных и векторизации.
4. **Раскрутка циклов (Loop Unrolling).** Данная трансформация дублирует тело цикла и помещает получившиеся инструкции после тела исходного цикла. Одна итерация нового

цикла выполняет эквивалент нескольких итераций исходного цикла. Соотношение количества итераций исходного и конечного циклов зависит от параметров трансформации. Раскрутка циклов позволяет избавиться от накладных расходов на некоторые инструкции в цикле, улучшить попадание в кэш и уменьшить ветвления. Для достижения такого эффекта инструкции, выполняемые в нескольких итерациях, объединяются в одну итерацию. Следует упомянуть, что раскрутка циклов также обладает недостатками. Например, возрастает нагрузка на регистры, которые используются для временных переменных, а также растет объем кода.

5. **Расслаивание циклов (Loop Peeling).** В процессе этой трансформации производится попытка упростить цикл или избавиться от зависимостей, разбивая его на несколько циклов с тем же телом, но разными пределами итераций.
6. **(Loop Tiling/Blocking).** Это преобразование разделяет пространство итераций циклов на блоки, для того чтобы данные, используемые на каждой итерации, могли оставаться в кэше для их последующего повторного использования. Разделение пространства итераций ведет к разделению больших массивов на маленькие блоки, которые помещаются в кэш полностью.
7. **Нормализация циклов (Loop Normalization).** Данная трансформация меняет индекс цикла с фиксированной итерацией так, что он начинается с нуля и увеличивается на единицу.

1.3. Приватизация (Privatization)

При распараллеливании одними преобразованиями циклов обойтись нельзя, так как для того, чтобы ими воспользоваться, необходимо устранить имеющиеся зависимости по данным. Приватизация — одна из традиционных техник, позволяющих избавиться от зависимостей по хранению (storage-related dependencies), связанных со скалярными переменными (scalar privatization) и массивами (array privatization). В процессе приватизации для каждого процессора, участвующего в параллельном выполнениии цикла, создается приватная копия переменной. Таким образом, запись и последующие чтения этой переменной различными процессорами не пересекаются друг с другом.

1.4. Редукция (Reduction)

Большое внимание уделяется операции редукции (или обновления), которая зачастую используется в различных научных приложениях. К таким операциям относятся аккумулирующие операции, такие как сложение, умножение, минимум/максимум. Эффективное распараллеливание таких операций может привести к значительному увеличению производительности всего приложения. Для трансформирования редукционных переменных необходимо провести анализ зависимости по данным, для того чтобы доказать, что перед нами на самом деле операция редукции, а также заменить последовательное вычисление редукции на параллельный алгоритм.

2. PROMIS

Среда PROMIS — это оптимизирующий и распараллеливающий компилятор, поддерживающий несколько языков, разрабатываемый в Исследовательском центре суперкомпьютерных вычислений университета Иллинойса [1, 3]. Последняя доступная версия выпущена 19 ноября 2002 года. Одной из основных задач, которые ставились при разработке PROMIS, является предоставление пользователям расширяемой платформы для разработки и исследования компиляторов. Компилятор PROMIS имеет Java, C, C++ и FORTRAN77 фронтэнды.

Внутренее представление кода в компиляторе PROMIS делится на три уровня абстракции.

1. HUIR (High level Universal Intermediate Representation). Это верхний уровень внутреннего представления. Он содержит все поддерживаемые компилятором выражения (неподдерживаемые выражения преобразуются в поддерживаемые) произвольной сложности. На данном уровне сохраняется максимум возможной семантики исходного кода.
2. LUIR (Low level Universal Intermediate Representation). Каждое выражение представляется как композиция более простых выражений. Все операции имеют форму, похожую на трехадресный код, поэтому некоторые сложные выражения преобразуются в наборы простых, а некоторые параметры функций заменяются на временные переменные.
3. IUIR (Instruction level Universal Intermediate Representation). Это нижний уровень внутреннего представления кода в ком-

пиляторе PROMIS. Для перехода на этот уровень каждая операция уровня LUIR ассоциируется с кодом машинной операции (machine opcode). Машинно-зависимые оптимизации производятся на этом уровне.

2.1. Проходы компилятора

Процесс компиляции в среде PROMIS состоит из нескольких проходов. На первом этапе исходный код синтаксически разбирается, а затем преобразовывается во внутреннее представление самого высокого уровня. Затем следуют проходы анализа (потока управления, потока данных), оптимизаций и генерации конечного кода. На первых фазах компиляции помимо внутреннего представления кода строятся следующие графы.

1. Иерархический граф задач (Hierarchical Task Graph). Операторы и поток управления программы представляются именно этим графом. Каждый узел иерархического графа задач представляет собой либо оператор, либо составной, начальный (start node) или конечный узел (stop node). Каждый уровень иерархии начинается с начального узла и заканчивается конечным.
2. Граф потока управления (Control Flow Graph). Представление в виде графа всех возможных путей исполнения в программе. Каждый узел в таком графе соответствует базовому блоку (basic block), а направленные дуги соответствуют потоку управления. Реализация этого графа в PROMIS отличается от обычного графа потока управления тем, что узлами являются не базовые блоки, а узлы HTG.
3. Граф зависимости по данным (Data Dependence Graph).
4. Граф зависимости по потоку управления (Control Dependence Graph). В данном графе узлами являются исполняемые операторы, а дуги соответствуют прямой зависимости по управлению. Если оператор X определяет, будет ли выполнен оператор Y, то говорят, что оператор Y зависит по управлению от оператора X.
5. Граф вызовов (Call Graph). Статический граф вызовов скомпилированной программы.

2.1.1. Анализ зависимостей по данным (Data Dependence Analysis)

Анализ зависимости по данным разделен на две составляющие: основанный на именах (Name-based Dependence Analysis) и основанный на индексах (Array Subscript-based Dependence Analysis). Зависимость по индексам определяется следующим образом: пусть операторы S_1 и S_2 обращаются к массиву X с помощью линейных функций $h(i_1, .., i_k)$ и $g(i_1, .., i_k)$ соответственно, где $i_1, ..., i_k$ — целочисленные индексы соответствующих вложенных циклов. Таким образом, между S_1 и S_2 существует зависимость, если существуют такие целочисленные $i_1, ..., i_k$ и $j_1, ..., j_k$, что i_p, j_p принадлежат $[L_p, U_p]$ (L_p, U_p — верхние и нижние границы цикла) и $h(i_1, ..., i_k) = g(j_1, ..., j_k)$. В компиляторе PROMIS используются три теста, которые могут определить отсутствие зависимости.

1. GCD Test. Этот метод состоит в попытке опровергнуть выполнимость линейных уравнений и основан на следующем наблюдении: если уравнение $a_1 * i_1 + a_2 * i_2 + ... + a_k * i_k = b$, где a_k целочисленные, имеет решение, тогда b кратно $gcd(a_1, a_2, ..., a_k)$, где gcd — наибольший общий делитель (Greatest Common Divisor).
2. Banerjee Test. Этот метод состоит в попытке опровергнуть выполнимость уравнений, основанных на границах цикла. Предположим, что коэффициенты a_i неотрицательны (для простоты). Из неравенств $L_j i_j U_j$, $j = 1...k$, следует, что если для $a_1 i_1 + a_2 i_2 + ... + a_k i_k = b$ есть решение, то b должно быть ограничено $a_1 L_1 + ... + a_k L_k$ и $a_1 U_1 + ... + a_k U_k$. Если это не так, то система независима. Иначе существует вещественное решение (что не подразумевает целочисленного решения) и система предполагается зависимой.
3. Omega Test. Целочисленный алгоритм, позволяющий определить существование и условия зависимости между двумя ссылками на массив. Он определяет существование целочисленного решения для произвольного набора линейных уравнений и неравенств. Для данного теста PROMIS использует библиотеку Omega версии 1.1. Эта библиотека реализована как часть проекта Omega [2] — набора каркасов (frameworks) и алгоритмов для анализа и трансформации научных программ, и предоставляет набор процедур для манипулирования множествами линейных ограничений над целочисленными переменными.

2.1.2. Символьный анализ (Symbolic Analysis)

Символьный анализ использует абстрактную интерпретацию для того, чтобы получить информацию о возможных значениях переменных [4]. Он разделяется на внутрипроцедурный (intraprocedural) и межпроцедурный (interprocedural). Внутрипроцедурный анализ применяется ко всем процедурам в программе одновременно для достижения большей производительности. Каждая процедура конвертируется в форму Static Single Assignment (SSA). SSA — это такое представление кода, в котором значение каждой переменной присваивается только один раз. Существующие переменные разделяются на версии так, что каждое определение получает свою версию. Таким образом, цепочки использование — определение становятся явными, и каждая такая цепочка будет состоять лишь из одного элемента.

2.2. Распараллеливание

В среде PROMIS большое внимание уделяется трансформациям и оптимизациям, которые помогают распараллелить выполнение программ и улучшить локальность данных. К ним относятся трансформации циклов, обработка приватизационных и редукционных переменных, а также распараллеливающие преобразования. Рассмотрим те, что реализованы в среде PROMIS.

2.2.1. Трансформации циклов

Обмен циклов (Loop Interchange). Обмен вложенных циклов может дать прирост производительности в случае распараллеливания итераций. В среде PROMIS данное преобразование осуществляется в два шага. На первом шаге происходит анализ эффективности, то есть анализируются все пары вложенных циклов и вычисляются преимущества обмена для каждой пары, который происходит на втором шаге. Данная реализация также обладает рядом недостатков, которые стоит упомянуть здесь. Во-первых, PROMIS игнорирует все вложенные двойные (two-loops) циклы, а во-вторых, не учитывает идеально вложенные циклы в связи с техническими сложностями во внутреннем представлении исходного кода.

Разделение цикла (Loop Distribution, Fission). Данная трансформация может улучшить параллелизм, локальность данных (locality) и векторизацию (vectorization). Разделение цикла в

PROMIS состоит из двух шагов.

1. PROMIS путем анализа внутреннего представления определяет, каким образом цикл может быть разделен. Основная идея состоит в том, чтобы все циклы DDG оставались в одном цикле программы, для чего необходимо определить сильно-связные компоненты (Strongly Connected Components) в этом графе.
2. Каждая сильно-связная компонента выделяется в отдельный цикл.

Реализация данного преобразования обладает следующими недостатками: из-за несовершенной поддержки потоков управления в PROMIS данная трансформация копирует в цикл все зависимые операторы. Таким образом циклы могут приобрести значительные размеры. Вторым недостатком данного алгоритма является то, что он перманентно изменяет структуру цикла, что приводит к большому количеству лишнего кода, если циклы сливаются с помощью loop fusion.

Слияние циклов (Loop Fusion). Данное преобразование можно использовать для улучшения параллелизма на уровне инструкций, локальности данных и векторизации. В компиляторе PROMIS реализована лишь сама трансформация, отсутствуют механизмы определения ее эффективности для различных циклов. Слияние — отдельный проход, который можно вызвать в любой момент компиляции. Его также можно использовать, для того чтобы убрать результаты разделения циклов. Слияние циклов проводится в два шага.

1. Определить возможность слияния. Анализируются зависимости по данным между сливаемыми циклами. Эти зависимости не должны обладать следующими свойствами:
   - если один из циклов итерируется по переменной i, а второй — по переменной j, и при этом i больше j;
   - циклы имеют разное количество итераций;
   - скалярные присваивания и использования, в частности upward-exposed, могут предотвратить данную трансформацию.
2. Собственно, само слияние.

Раскрутка циклов (Loop Unrolling). Процесс раскрутки состоит из двух шагов.

1. Проверка цикла. На данном шаге проверяется, что обратное ветвление — это последняя инструкция в цикле.
2. Дублирование и вставка инструкций в исходный цикл.

Основным недостатком реализации является то, что обратные дуги (backedge branch) не будут убраны. Они будут использованы для того, чтобы выйти из цикла в случае достижения граничного условия. В стандартной реализации раскрутки эти дуги убираются, и раскрученное тело цикла безусловно выполняет все n итераций, где n — количество повторений цикла.

Расслаивание циклов (Loop Peeling). Во время расслаивания циклов отдельные итерации исходного цикла помещаются в отдельный цикл. Реализация в компиляторе PROMIS очень похожа на реализацию раскрутки циклов, за исключением того, что на каждом шаге выделяется только одна итерация. Кроме того, для удачного расслаивания исходный цикл должен обладать фиксированным числом операций. Преобразование происходит в два шага.

1. Предварительная проверка. PROMIS проверяет, что исходный цикл действительно является циклом с фиксированной итерацией, а также находит итерационные переменные. Если итерационных переменных найти не удается, преобразование не производится.
2. Расслаивание. Компилятор дублирует тело исходного цикла и уменьшает количество итераций на единицу.

Нормализация циклов (Loop Normalization). Данная трансформация меняет индекс цикла с фиксированной итерацией так, что он начинается с нуля и увеличивается на единицу. PROMIS может нормализовывать любые циклы с фиксированной итерацией. Текущие границы цикла и инкремент используются для вычисления новой верхней границы, так как в нормализованном цикле нижняя граница — 0, а инкремент — 1.

2.2.2. Трансформации переменных

Приватизация (Privatization). Переменная считается *приватизируемой* (privatizable), если каждый процессор может создать свою приватную версию этой переменной и производить всю работу

над этой приватной версией. Алгоритм определения приватизации разделяет такие переменные на два класса: полностью приватизуемые переменные и выживающие (live out) приватизируемые переменные. Полностью приватизируемые переменные соответствуют данному выше определению и объявляются перед использованием в каждой итерации цикла. Выживающие приватизируемые переменные также определяются перед использованием в каждой итерации цикла, но их значение используется вне цикла. Для того чтобы поддержать семантику исходной программы, значение выживающей приватизируемые переменной должно быть присвоено исходной переменной в конце последней итерации цикла.

Реализованный в PROMIS алгоритм определения приватизации использует информацию, собранную на проходе символьного анализа, который преобразует внутреннее представление кода в форму Single Static Assignment (SSA). Трансформация приватизации производится на выживающих переменных, так как для полностью приватизационных переменных она не требуется. Первый шаг трасформации создает новую локальную переменную и заменяет на нее все вхождения исходной переменной. На втором шаге в цикл вставляется тест, проверяющий достижение последней итерации и сохраняющий значение локальной копии в исходную переменную, если это последняя итерация.

Редукция (Reduction). Переменной редукции называется такая переменная, которая аккумулирует значение. На данном проходе обрабатываются следующие типы редукционных переменных: суммарные редукции (формы $op = op + / - constant$), редукции произведения (формы $op = op * constant$), редукции максимума/минимума (формы $op = min(op1, op2)$ или $op = max(op1, op2)$). Алгоритм определения редукционных переменных проходит по всем операторам цикла и сопоставляет его с каждым из трех образцов редукцонных переменных. Для каждой из найденных переменных также определяется, используется ли она в других местах цикла. Трансформация редукционных переменных производится в два шага. На первом шаге создается локальная копия редукционной переменной, и на нее заменяются все вхождения исходной. Инициализация редукционной переменной помещается в преамбулу, а конечная редукция и глобальная переменная создаются в эпилоге.

Основным недостатком реализации является ее ограниченность только скалярными значениями и циклами с фиксированной итерацией.

2.2.3. Распараллеливающие преобразования

DO-to-DOALL анализ. Данный вид анализа определяет циклы, не имеющие межитерационных зависимостей. Анализ лишь находит такие циклы, но не производит преобразований над ними. Для того чтобы такие циклы сделать действительно параллельными, необходимо применить IML-преобразование.

IML-преобразование. Illinois-Intel Multithreading Library (IML) — многопоточная/многопроцессорная библиотека, разрабатываемая в Исследовательском центре суперкомпьютерных вычислений университета Иллинойса. IML предоставляет набор удобных функций для воплощения параллелизма. Данный проход преобразует DOALL-циклы в набор вызовов функций IML, которые исполняют тело каждого цикла параллельно. К недостаткам данной реализации можно отнести неправильную работу с приватизационными, live out и редукционными переменными.

Alpha-Coral преобразование. Так же как и в IML-преобразовании, данный алгоритм преобразует параллельные циклы в наборы функций. Каждая из функций содержит одну итерацию цикла. Остальные инструкции, такие как проверка границ и инициализация, убираются. Данная реализация поддерживает только readonly разделяемые переменные (определенные вне потока, но используемые внутри потока).

3. SUIF

SUIF — группа Стэнфордского университета, занимающаяся разработкой одноименной системы — SUIF Compiler System. Это крупный проект, который покрывает достаточно большой круг задач в области компиляторов. Одним из основных направлений, которыми занимается данная исследовательская группа, — автоматическое распараллеливание программ.

В данной статье рассматривается компилятор SUIF версии 1.x [5].

При распараллеливании компилятор должен использовать крупномодульный параллелизм программы (*coarse-grain parallelism*) — находить крупные вычисления, которые могут быть выполнены параллельно. Такой подход необходим для того, чтобы выигрыш в производительности превышал затраты на синхронизацию и коммуникацию между процессорами. Многопроцессорные систе-

мы имеют более сложную иерархию памяти, чем векторные. Например, такие системы содержат многоуровневые кэши. Поэтому для повышения производительности необходимо также обеспечить эффективное использование памяти.

3.1. Крупномодульный параллелизм

Цель распараллеливания заключается в загрузке процессоров как можно более крупными блоками независимых вычислений. Для повышения эффективности все анализы проводятся на едином межпроцедурном фреймворке.

Есть три основные части анализа, которые отвечают за обнаружение крупномодульного параллелизма.

Анализ скалярных переменных. В ходе данного анализа выявляются операторы, содержащие скалярные переменные, и те, которые могут быть выполнены параллельно. Для этого используются анализ зависимостей, анализ приватизации переменных, распознавание редукции. Кроме того, анализ скалярных переменных предоставляет некоторую символьную информацию, включающую распространение констант, обнаружение переменных индукции и инвариантов циклов и др.

Анализ массивов. Этот анализ применяет тест на зависимость по данным, основанный на решении систем линенйных уравнений и неравенств (решение задач целочисленного программирования), для определения того, какие вырезки обращаются к одному и тому же месту в памяти. Такая информация используется при приватизации массивов, распознавании операций, которые могут быть заменены на операции редукции.

Поддержка межпроцедурного анализа. Все анализы используют единый межпроцедурный фреймворк, который реализует *истинный межпроцедурный анализ* [6]. Тело каждой процедуры анализируется на предмет побочных эффектов, которые представляются в виде функции. Затем построенные функции применяются к вызовам процедур, при этом возможно выборочное копирование тела какой-либо процедуры для более оптимальной обработки вызова в разных контекстах (например, в случае, когда формальный параметр принимает несколько различных константных значений).

Авторы статьи [9] утверждают, что описанный подход к межпро-

цедурному анализу более эффективен, чем подстановка процедур, так как при выполнении анализа код «не разбухает» из-за подстановок вызовов.

3.2. Оптимизации использования памяти

В многопроцессорных системах присутствуют четыре аспекта использования памяти, снижающие производительность.

- Межпроцессорное взаимодействие. При использовании различными процессорами одной и той же области памяти возникает необходимость в синхронизации. Такие потери называют *истинными потерями совместного использования*[1].
- Ограниченный размер кэша. Некоторые приложения, например реализующие вычислительные задачи, обрабатывают большие массивы данных, значительно превышающие размер кэша. Часто происходит так, что приложение переиспользует данные только после того, как обработает достаточно большое их количество, что приводит к бедной *временной локальности данных*[2] и потере производительности (*capacity misses*).
- Ограниченная ассоциативность. Обычно каждая область памяти может быть записана лишь в строго определённое место в кэше. Потери из-за конфликта по памяти (*conflict misses*) происходят, когда разные ячейки памяти претендуют на запись в одну и ту же область в кэше.
- Большой размер кэш-строк. Как известно, данные в кэш пересылаются единицами фиксированной длины, которые называются кэш-строками (*cache lines*). Говорят, что вычисление, организованное таким образом, что в каждой кэш-строке используется мало данных перед тем, как она вытеснится из кэша, имеет бедную *пространственную локальность данных* (*data spatial locality*). Такие вычисления приводят к потере производительности. Также в мультипроцессорных системах могут возникать ситуации, когда разные процессоры используют данные из одной и той же кэш-строки, что также приводит к потерям производительности — *ложным потерям совместного использования*(*false sharing misses*).

[1] От англ. *true sharing misses.*

[2] Говорят, что некоторое вычисление имеет *временную локальность данных* (от англ. *data temporal locality*), если оно переиспользует достаточно много данных, к которым получает доступ.

В ходе своей работы компилятор пытается убрать как можно больше кэш-потерь (*cache misses*), а затем минимизировать эффект от оставшихся. Решение описанных выше проблем сводится к выполнению следующих условий:

- процессоры переиспользуют как можно больше данных;
- данные из памяти общего доступа, которые использует какой-либо процессор, расположены непрерывно везде, где это возможно.

Наконец, для того чтобы компенсировать время на оставшиеся кэш-потери, компилятор использует инструкции предвыборки (*compiler-inserted prefetching*) для загрузки данных в кэш перед их использованием.

3.2.1. Увеличение объёма переиспользуемых данных

Для увеличения объёма переиспользуемых процессорами данных применяется *аффинное разбиение*. Построение разбиения заключается в нахождении аффинных отображений вычислений и данных на процессоры. Аффинное отображение (линейное преобразование + смещение) выводится на основе анализа использования данных в программе (вырезок). Из всего множества таких отображений выбираются те, которые максимизируют переиспользование данных. Для более детального описания алгоритма построения см. [10].

3.2.2. Непрерывность данных

Непрерывность данных, используемых каким-либо процессором, обеспечивается тем, что компилятор SUIF управляет расположением данных в памяти. После того как определено, какие данные будут использоваться процессором (из результатов аффинного разбиения), компилятор трансформирует данные (массивы), применяя перестановки (например, транспонируя двумерные массивы) и изменения размерностей, тем самым организуя их непрерывное расположение. Кроме того, с целью обеспечения непрерывности расположения данных нескольких массивов компилятор использует «управляемую раскраску страниц памяти»(*compiler-directed page coloring*) — то есть информацию об использовании данных для управления политикой выделения страниц памяти операционной

системы с целью обеспечения непрерывность данных, используемых каждым процессором, в физическом адресном пространстве.

3.3. Экспериментальные результаты

В целях аппробации SUIF был протестирован на 10 программах из пакета SPECfp95 [8] и на 8 — из пакета NAS [7]. Рис. показывает увеличение быстродействия SPECfp95 и NAS-программ. Замеры производились на 8-процессорной системе (по 300МГц каждый) AlphaServer.

Для того чтобы замерить эффект каждой из техник компилятора, было проведено разбиение производительности на три группы, как показано на рис. , (a),(b). Baseline — ускорение, полученное парраллелизацией, основанной на межпроцедурном анализе зависимостей, приватизации скалярных переменных и преобразовании редукции. Coarse-grain включает в себя baseline, приватизацию и преобразования редукции массивов и полный межпроцедурный анализ скалярных переменных и массивов. Memory — все перечисленное выше плюс оптимизации использования памяти. Из рис. 2 видно, что крупномодульная параллелизация и оптимизации по использованию памяти значительно влияют на прирост производительности, по крайней мере, для половины тестов. В случае тестов «swim» и «tomcatv» компилятор устранил практически все кэш-потери. Для большинства программ, которые не получили выигрыша в производительности, компилятор нашёл значительную часть вычислений, которые могли бы быть распараллелены, но принял решение не проводить распараллеливание за отсутствием выигрыша. Тесты «fpppp» и «buk» не содержали циклов, доступных для статического анализа SUIF.

Результаты тестирования и SPEC оценок, полученных при прогонке на 8-процессорной (440 Мгц каждый) системе Digital AlphaServer 8400, содержит табл. 1.

4. Достоинства и недостатки

4.1. PROMIS

К преимуществам среды PROMIS можно отнести следующее.

1. Фронтенды для различных языков программирования (таких как C, C++, Fortran, Java) и богатые возможности по генерации кода — как ассемблерного, так и на языке C.

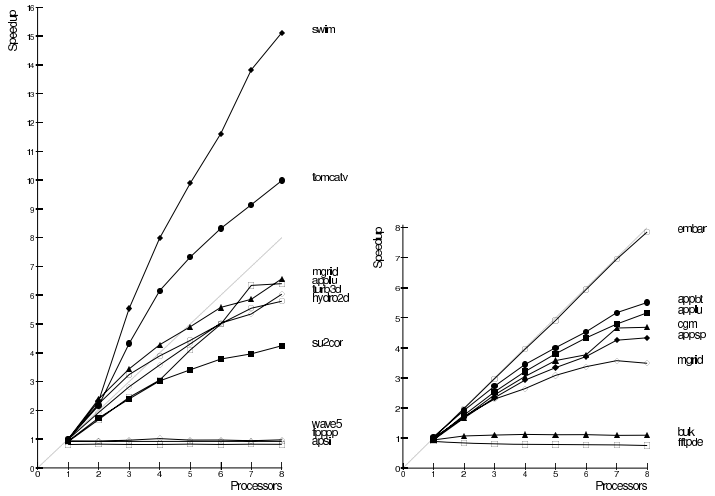

Рис. 1. Увеличение быстродействия: (a) — SPECfp95 и (b) — NAS.

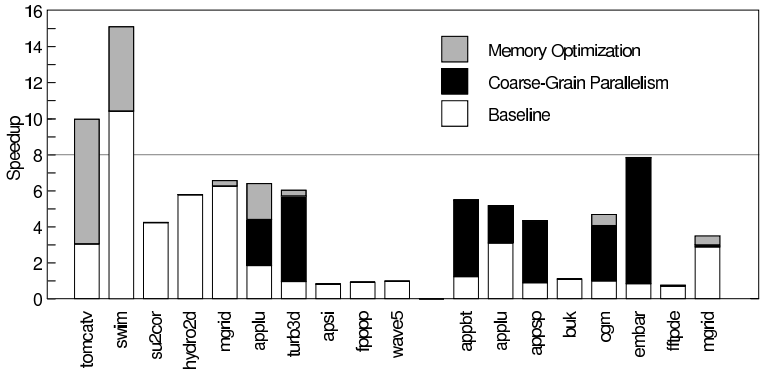

Рис. 2. Влияние оптимизаций SUIF на быстродействие.

2. Мощные средства символьного анализа. Мощное средство — для получения представления о работе программы, которое может быть использовано для различных оптимизаций.
3. Богатый набор преобразований, таких как преобразования

Табл. 1. **Абсолютные результаты тестирования для SPECfp95 на 440 МГц Digital AlphaServer**

	Execution Time (sec)			SPEC ratio		
Benchmark	1P	4P	8P	1P	4P	8P
tomcatv	219.1	30.3	18.5	16.9	122.1	200.0
swim	297.9	33.5	17.2	28.9	256.7	500.0
su2cor	155.0	44.9	31.0	9.0	31.2	45.2
hydro2d	249.4	61.1	40.7	9.6	39.3	59.0
mgrid	185.3	42.0	27.0	13.5	59.5	92.6
applu	296.1	85.5	39.5	7.4	25.7	55.7
turb3d	267.7	73.6	43.5	15.3	55.7	94.3
apsi	137.5	141.2	143.2	15.3	14.9	14.7
fpppp	331.6	331.6	331.6	29.0	29.0	29.0
wave5	151.8	141.9	147.4	19.8	21.1	20.4
SPEC ratio				**15.0**	**44.4**	**63.9**

циклов, преобразования приватных и редукционных переменных, а также IML и Alpha-Coral преобразования.

4. Открытая архитектура — очень просто создавать свои модули для PROMIS и управлять процессом компиляции, добавляя собственные анализы и оптимизации, а также используя существующие.

С другой стороны, PROMIS обладает рядом следующих недостатков.

1. Реализованы только алгоритмы самих преобразований. Отсутствуют механизмы, определяющие то, какие преобразования необходимо применить к данному циклу для того, чтобы получить параллелизм, и тем более параллелизм без синхронизации. В результате все преобразования требуют либо взаимодействия с пользователем, который «вручную» указывает действия для каждого цикла, либо отдельного модуля, который бы управлял преобразованиями автоматически.
2. PROMIS поддерживает незначительное количество преобразований, дающих в результате параллельный код. Реализована только DO-to-DOALL трансформация, которая работает на циклах без меж-итерационных зависимостей.

4.2. SUIF

К достоинствам среды SUIF можно отнести следующее.

1. Анализ зависимостей. Платформа SUIF реализует мощный алгоритм анализа зависимостей такой как афинное разбиение. Он позволяет определять наличие зависимостей в большем количестве случаев чем другие алгоритмы анализа зависимостей и тесты.
2. Автоматизм. В связи с тем что среда SUIF реализует мощный анализ зависимостей, при компиляции и распараллеливании не требуется взаимодействие с программистом. Разработчику не требуется указывать какие циклы стоит преобразовывать для достижения максимальной эффективности параллельной программы.

К недостаткам SUIF можно отнести следующее.

1. Разделение цикла. В SUIF не реализовано преобразование «разделение цикла», что приводит в определённых ситуациях к невозможности распараллелить некоторые циклы, например цикл, содержащий операции ввода-вывода (цикл взят из Perfect club benchmark APS.f):

```
    DO 30 K=1,NZ
        UM(K)=REAL(WM(K))
        VM(K)=AIMAG(WM(K))
        WRITE(6,40) K,ZET(K),UG(K),VG(K),TM(K),DKM(K),UM(K),VM(K)
        WRITE(8,40) K,ZET(K),UG(K),VG(K),TM(K),DKM(K),UM(K),VM(K)
30  CONTINUE
```

   может быть успешно разделён на два, один из которых можно распараллелить:

```
    DO 30 K=1,NZ
        UM(K)=REAL(WM(K))
        VM(K)=AIMAG(WM(K))
2   CONTINUE

    DO 30 K=1,NZ
        WRITE(6,40) K,ZET(K),UG(K),VG(K),TM(K),DKM(K),UM(K),VM(K)
        WRITE(8,40) K,ZET(K),UG(K),VG(K),TM(K),DKM(K),UM(K),VM(K)
3   CONTINUE
```

2. Другие ограничения. Следующие ограничения сужают область применимости распараллеливания в SUIF:

- SUIF не поддерживает анализ, предоставляющий информацию о том, какие переменные могут потенциально указывать на одну и ту же область памяти (видимо, отсутствует анализ указателей);
- SUIF не рассматривает не нормализованные циклы;
- текущая реализация анализа зависимостей рассматривает линейные аффинные функции, поэтому если границы цикла или вырезка из массива содержит нелинейные выражения (в том числе и символьные), то они исключаются из рассмотрения;
- в SUIF отсутствует символьный анализ, нужный, например, в таком случае:

```
14  KBOT = KK - 1
    KTOP = KK
    ... code containing IF statements ...
    DO 18 K=KK,KTOP
18  Q(K) = Q(KBOT)
```

  В результате символьного анализа `KBOT` заменилось бы на `KK-1`, и в результате можно было бы определить, что две вырезки в операторе 18 независимы.
- SUIF поддерживает не все возможные формы редукции. Например, SUIF не распознаёт редукцию одних элементов массивов в другие:

```
    DO 30 I=2,N2P
30  WORK(1) = MAX(WORK(1), WORK(I))
```

Заключение

В статье были рассмотрены открытые системы для создания распараллеливающих компиляторов. Каждая из них обладает как достоинствами, так и недостатками, а также ограниченными областями применения.

По сравнению с системой SUIF среда PROMIS обладает следующими достоинствами: удобный механизм расширения функциональности, реализованы различные алгоритмы преобразования циклов, а также простые тесты зависимостей. Но в PROMIS отсутствуют способы определения необходимости каждого конкретного преобразования, поэтому среда требует постоянного взаимо-

действия с разработчиком для получения эффективной параллельной программы. В то время как SUIF реализует мощные алгоритмы анализа зависимостей и определения эффективности преобразований и может работать полностью автоматически. Таким образом PROMIS можно использовать как каркас для построения отдельного распараллеливающего компилятора для того, чтобы облегчить рутинные задачи, такие как преобразования циклов и символьный анализ. SUIF можно использовать как автоматический распараллеливающий компилятор без дополнений, а также можно расширять функциональность по мере необходимости. Кроме того, последняя версия среды PROMIS была выпущена в 2002 году и затем работа над проектом не была продолжена, а проект SUIF продолжает развиваться.

Список литературы

[1] *Brokish J., Carroll S., Ko W. et al.* The PROMIS Compilation System. May 21, 2001. http://promis.csrd.uiuc.edu.

[2] http://www.cs.umd.edu/projects/omega/.

[3] http://www.csrd.uiuc.edu/promis/.

[4] *Stavrakos N., Carroll S., Saito H.* et al. Symbolic Analysis in the PROMIS Compiler. NCS 1863, 2000. P. 468–471.

[5] http://suif.stanford.edu/suif/suif1/index.html.

[6] *Itigoin F., Jouvelot P., Triolet R.* Semantical interprocedural parallelization: An overview of the PIPS project. In Proceedings of the 1991 ACM International Conference on Supercomputing, Cologne, Germany, June 1991. P. 244–251.

[7] http://www.nas.nasa.gov/Resources/Software/npb.html.

[8] http://www.spec.org/cpu95/.

[9] *Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe et al.* Maximizing Multiprocessor Performance with the SUIF Compiler. Digital Technical Journal 10(1). 1998. P. 71–80.

[10] http://suif.stanford.edu/papers/anderson95/node7.html.