Технология автоматизированной разработки компиляторов языков высокого уровня

для вычислительных систем с распределенными ресурсами

Широкое внедрение вычислительных систем с распределенными ресурсами существенно сдерживается отсутствием средств автоматизированной разработки компиляторов языков высокого уровня (ЯВУ) [1]. В работе [2] были описаны созданные в СНИЛ «Новые информационные технологии» (Гомельский государственный университет, http://NewIT.gsu.unibel.by) средства автоматизированной разработки распределенных вычислительных систем, которые в значительной степени опираются на описываемый в данной работе программный комплекс автоматизированной разработки компиляторов ЯВУ для вычислительных систем с распределенными ресурсами.

Михаил Долинский,
Алексей Толкачев

dolinsky@gsu.unibel.by

1. Общая схема использования программного комплекса

Технология реализации компилятора ЯВУ для вычислительной системы с распределенными ресурсами состоит в последовательном выполнении следующих этапов:

1. Создание компилятора в промежуточное представление. Данный этап частично автоматизирован за счет применения универсального синтаксического анализатора Unisan. На данном этапе разработчик реализует:
   - Описание синтаксиса входного ЯВУ-компилятора.
   - Функции генерации промежуточного представления по построенному в ходе разбора синтаксическому дереву.
   - Необходимый набор интерфейсных функций.

   Результатом работы компилятора в промежуточное представление является промежуточное представление программы, не содержащее информации о параллельном выполнении.
2. Реализация генератора исполняемого кода. На данном этапе разработчик реализует:
   - Функции оптимизации, учитывающие возможности целевой архитектуры.
   - Функции генерации исполняемого кода. Исходной информацией служит промежуточное представление, содержащее информацию о параллелизме.
   - Генерация отладочной информации.
   - Необходимый набор интерфейсных функций.

   Результатом работы генератора исполняемого кода является сохраненный на диске файл в формате ESDI, содержащий отладочную информацию и исполняемый код для целевой платформы.
3. Интеграция компонентов компилятора. Разработчик включает реализованные на предыдущих этапах модули в один проект с универсальными модулями (используется среда разработки Microsoft Visual C++ .NET). Осуществляется компиляция, отладка и верификация компилятора. При необходимости разрабатывается набор тестов для тестирования полученного компилятора. Результатом является исполняемый файл компилятора входного ЯВУ для целевой архитектуры с распределенными ресурсами.
4. Интеграция компилятора со средой разработки. При использовании сред разработки программного и аппаратного обеспечения встроенных систем IEESD или WInter необходимо реализовать:
   - Описание платформы, состоящее из описания командных строк для запуска компилятора, некоторых дополнительных настроек, описания файлов с документацией и др. Описание платформы осуществляется с использованием специального языка систем IEESD и WInter.
   - Разработка документации, интегрируемой в среду разработки. Документация должна быть сохранена в формате HTML.
   - Настройка системы пакетного тестирования. Встроенная в IEESD и WInter система тестирования позволяет осуществлять компиляцию множества тестовых программ и моделирование их исполнения с проверкой результатов в пакетном режиме. В результате генерируется отчет, содержащий информацию о прохождении каждого теста. Пакетное тестирование может применяться как разработчиком на этапе отладки компилятора, так и конечными пользователями для автоматической проверки разрабатываемого программного обеспечения для систем с распределенными ресурсами.

Процесс реализации компилятора ЯВУ для вычислительной системы с распределенными ресурсами схематично изображен на рисунке. Далее описана технология реализации каждого из перечисленных этапов.

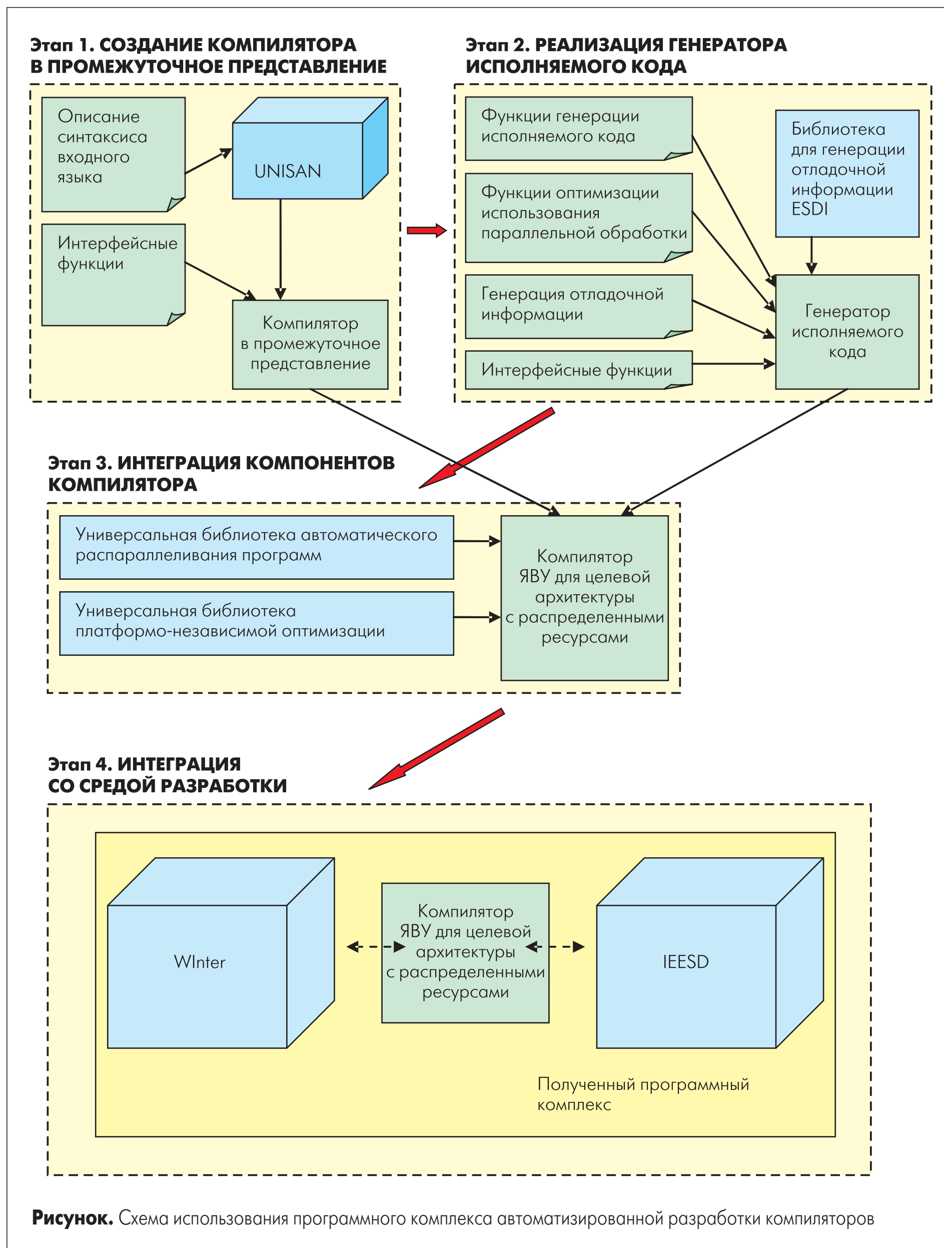

Рисунок. Схема использования программного комплекса автоматизированной разработки компиляторов

2. Создание компилятора в промежуточное представление

2.1. Реализация интерфейсных функций компилятора в промежуточное представление

Для взаимодействия компилятора в промежуточное представление с универсальными блоками используются:

- класс CCFrontEnd, имеющий функцию Compile и поле CCProgram *Program;
- структуры промежуточного представления.

Для создания компилятора необходимо описать класс CCFrontEnd, имеющий функцию Compile и поле CCProgram *Program. Функция Compile должна выполнять трансляцию исходного файла с указанным именем в промежуточное представление, указатель на который помещается в поле Program.

В общем случае разработчик может реализовывать front-end-компилятор с помощью любых средств. В случае если входной информацией для распараллеливающего компилятора является представление программы в некотором бинарном формате, разработчику необходимо самостоятельно реализовать его транслятор в промежуточное представление.

В случае если определенные конструкции входного языка не могут быть представлены имеющимися структурами промежуточного представления, формат промежуточного представления может быть расширен новыми структурами или полями. При этом назначение существующих полей и структур не должно изменяться, поскольку они используются универсальными блоками компилятора. Граф передачи управления и зависимостей по данным всегда должен представляться с помощью стандартных структур. Например, структура CCVariable может быть расширена информацией о типе памяти, в которой размещается переменная; структура CCOperator не может быть расширена новым полем, указывающим на условие выполнения оператора.

Для автоматизации проектирования компиляторов ЯВУ рекомендуется использовать ПТК Unisan.

2.2. Создание синтаксического анализатора входного языка с помощью универсального синтаксического анализатора Unisan

Интеграция с универсальным синтаксическим анализатором Unisan осуществляется следующим образом.

Во-первых, необходимо создать новый класс (далее будем называть его _CMS), унаследованный от CUnisanExternalImplementation (описан в исходном файле parser_implementation.h):

```
class _CMS : public UnisanExternalImplementation;
```

Класс CUnisanExternalImplementation содержит «пустые» реализации всех функций интерфейса IUnisanExternal, которые библиотека parser.dll вызывает в ходе разбора исходного текста. В классе _CMS необходимо реализовать функции, используемые в ходе разбора исходного текста.

Для построения синтаксического дерева используются классы:

- _ASTNode — узел дерева (исходный файл ASTNode.h);
- _ASTFactory — содержит функции создания, удаления и соединения узлов дерева (исходный файл ASTFactory.h).

Необходимо реализовать функции, выполняющие создание и соединение узлов дерева в ходе разбора:

```
IUnisanTreeNode* __stdcall
_CMS::CreateNode(int node_type);

IUnisanTreeNode* __stdcall
_CMS::SetChildren(IUnisanTreeNode **nodes,
                  int nodes_count);
```

Эти функции вызываются библиотекой parser.dll в соответствии с описанными в исходном bnf-файле конструкциями построения синтаксического дерева.

Кроме функций построения дерева необходимо реализовать другие функции IUnisanExternal, выполняющие определенные действия, например функции AddCommand, AddEOF, Error.

После реализации интерфейсных функций для библиотеки parser.dll необходимо создать bnf-файл с описанием синтаксиса входного языка. В описание синтаксиса добавляются конструкции построения синтаксического дерева. Исходный bnf-файл обрабатывается компилятором описания грамматики bnfc.exe.

Следующим шагом является реализация генератора промежуточного представления, который обходит построенное синтаксическое дерево и сохраняет информацию об исходной программе в виде структур промежуточного представления. Для реализации рекомендуется использовать нисходящий рекурсивный алгоритм обхода.

В простейшем случае, когда все конструкции входного языка эквивалентны конструкциям языка C, компилятор в промежуточное представление может быть создан на основе существующего редактированием bnf-файла с описанием грамматики.

3. Создание генератора исполняемого кода

3.1. Анализ возможностей и ограничений целевой платформы

Первым шагом при проектировании компилятора для вычислительной системы с распре-

KiT_45(1)_web.qxd 1/26/2005 10:43 AM Page 140

деленными ресурсами является анализ возможностей и ограничений целевой архитектуры.

Существенное значение для разработки алгоритма генерации исполняемого кода и алгоритмов оптимизации имеют следующие характеристики целевой платформы:

- Максимальное количество параллельно исполняемых инструкций.
- Количество арифметико-логических устройств каждого типа.
- Возможность параллельного обращения к разным типам памяти.
- Возможность конфигурирования платформы, то есть управления перечисленными выше характеристиками в некотором диапазоне, а также создания новых инструкций и арифметико-логических устройств.

В случае использования конфигурируемой архитектуры необходимо разработать алгоритмы проверки возможности аппаратной реализации конкретной конфигурации.

3.2. Использование информации о параллелизме

Следующим шагом после анализа характеристик вычислительной системы является разработка алгоритмов оптимизации параллельного промежуточного представления.

Наиболее сложной задачей при создании back-end-компилятора является разработка алгоритмов оптимизации, позволяющих получать исполняемый код, эффективно использующий возможности данной архитектуры. Получаемое максимально распараллеленное промежуточное представление кодогенератор должен преобразовать таким образом, чтобы минимизировать потери производительности и при этом учесть перечисленные характеристики целевой архитектуры.

Для оптимизации генерируемого кода могут быть использованы эвристические оценки относительного времени выполнения инструкций, полученные в ходе работы распараллеливающего компилятора, а также информация о явно описанном параллелизме, сохраненная с помощью поля ParallelLabel в операторах промежуточного представления.

В некоторых случаях может быть целесообразным реализовать несколько алгоритмов оптимизации и дать конечному пользователю компилятора возможность выбора одного из них.

3.3. Реализация интерфейсных функций генератора исполняемого кода

Для взаимодействия с генератором исполняемого кода используются:

- класс CCBackEnd, имеющий функцию Generate и конструктор с определенными параметрами;
- структуры промежуточного представления.

Для каждого оператора промежуточного представления должная быть реализована функция CCOperator::Generate, выполняющая генерацию исполняемого кода для данного оператора.

Функция CCBackEnd::Generate должна осуществить генерацию исполняемого кода для переданного указателя на промежуточное представление программы и сохранить отладочную информацию и исполняемый код в формате ESDI с помощью динамически подключаемой библиотеки esdi.dll.

3.4. Реализация функций генерации кода

Промежуточное представление для каждой функции имеет древовидную структуру. Алгоритм генерации кода заключается в вызове функции Generate для каждого оператора главного блока функции.

Функции Generate для блочных операторов осуществляют генерацию кода для всех блоков оператора и, кроме того, генерируют дополнительные инструкции проверки условий, условных переходов и т. д.

Предлагается следующая структура генератора исполняемого кода:

- В классе CCBackEnd реализуются общие функции для сохранения кодов исполняемых инструкций, их операндов и отладочной информации.
- Реализация функций Generate операторов промежуточного представления заключается в вызове функций класса CCBackEnd с передачей нужных параметров — бинарных кодов инструкций, мнемоник для генерации ассемблерного текста и т. д.
- Функции Generate операторов промежуточного представления должны проверять тип операндов и генерировать соответствующий исполняемый код для операндов каждого типа.

Рассмотрим функцию генерации кода для оператора сложения CCAddOperator, реализованную в компиляторе языка CMPDL

```
void CCPlusOperator::Generate()
{
l1:     PrintDebug(«Plus», Left, Right, Result);

l2:     if (Result->GetType() == tfFloat) {
l3:         FloatAdd(Left, Right, Result);
        }
        else {
l4:         BE->PutCode(C_ADD, 3,
                Left->IsReference,
                Right->IsReference,
                Result->IsReference);
l5:         BE->PutOperand(Left);
l6:         BE->PutOperand(Right);
l7:         BE->PutOperand(Result);
        }
}
```

В строке l1 функция PrintDebug сохраняет информацию, используемую для отладки компилятора. Строки l2 и l3 осуществляют вызов функции генерации кода в случае, если операндами являются переменные с плавающей точкой. Строки l4–l7 осуществляют генерацию кода для целочисленных операндов. Функции PutCode и PutOperand реализованы в классе CCBackEnd и используются всеми операторами для сохранения исполняемого кода.

3.5. Генерация отладочной информации

Генерация отладочной информации осуществляется в несколько этапов:

- Расположение данных в памяти определяется в ходе выполнения алгоритмов оптимизации. Информация о типе памяти и адрес переменной сохраняется в структуре CCVariable.
- На этапе генерации исполняемого кода функциями Generate в поле Offset операторов промежуточного представления сохраняются смещения инструкций, соответствующих операторам.
- Формируется полное описание отладочной информации и исполняемого кода с помощью структур, описанных в исходном файле CDebugInfo.h.
- Исполняемый код и отладочная информация сохраняется в файл в формате ESDI с помощью библиотеки esdi.dll.

Для отладки генератора отладочной информации и проверки правильности заполнения всех структур используются встроенные в библиотеку esdi.dll средства сохранения отладочного дампа, а также специальная утилита для просмотра содержимого файлов в формате ESDI в текстовом виде.

4. Интеграция компилятора со средой отладки

После реализации компилятора в промежуточное представление и генератора исполняемого кода все компоненты компилятора объединяются на уровне исходного кода в один проект в среде разработки Microsoft Visual C++ .NET. Осуществляется компиляция и отладка полученного компилятора.

При создании множества тестов компилятора за основу может быть взят стандартный набор тестов для компилятора GCC. Компилятор GCC является самым распространенным настраиваемым компилятором с открытыми кодами. Это позволяет говорить о том, что тестирование разрабатываемого компилятора на данном множестве тестов является серьезной проверкой. Из множества тестов GCC следует выбрать тесты, которые не используют специфические возможности платформ, на которые имеется настройка этого компилятора. Кроме того, необходимо учитывать поддерживаемые возможности входного языка. Тесты GCC созданы для тестирования компиляторов языка C, но могут быть достаточно легко переведены и на другие языки.

Для полного тестирования разрабатываемого компилятора кроме имеющегося набора платформенно-независимых тестов компилятора GCC следует разработать тесты, учитывающие специфику целевой архитектуры, в частности, возможности распараллеливания.

Необходимо выделить следующие группы тестовых программ:

- программы, которые должны успешно компилироваться;
- программы, при компиляции которых должна обнаруживаться определенная ошибка;
- программы, которые должны компилироваться, исполняться и возвращать код успешного завершения.

Исполняемые тесты возвращают код завершения вызовом специальной функции. Это позволяет просто реализовать автоматическое тестирование в пакетном режиме в средах WInter и IEESD. Разработана система, позволяющая автоматически производить тестирование на заданном наборе тестов с генерацией детализированного отчета с общей статистикой — количеством успешно и неуспешно выполненных тестов, информацией о специфических ошибках тестирования, а также с информацией по каждому тесту.

KiT_45(1)_web.qxd 1/26/2005 10:43 AM Page 141

Заключение

Основные достоинства предложенной технологии автоматизированной разработки компиляторов ЯВУ для вычислительных систем с распределенными ресурсами заключаются в следующем:

- отсутствие необходимости расширения синтаксиса входного языка для введения специальных конструкций для описания параллелизма;
- анализ программы и автоматическое распараллеливание выполняется универсальным модулем, следовательно, разработчик компилятора освобождается от реализации этих блоков;
- основной задачей разработчика является учет особенностей и ограничений целевой архитектуры;
- поддержка средств отладки реализуется с минимальными трудовыми затратами, поскольку имеется интегрированная с компилятором библиотека для сохранения отладочной информации;
- применимость для создания компиляторов практически любых языков программирования;
- технология может применяться для проектирования оптимизирующих компиляторов для широкого класса вычислительных систем с распределенными ресурсами, в частности, для VLIW-процессоров, для многопроцессорных систем, для генерации аппаратных схем и др.
- значительно снижены необходимые объем работы и трудоемкость реализации оптимизирующих компиляторов для вычислительных систем с распределенными ресурсами по сравнению с другими средствами проектирования компиляторов;
- технология может применяться для обучения специалистов в области проектирования компиляторов для параллельных вычислительных систем.

Литература

1. Долинский М., Толкачев А. Обзор аппаратных и программных средств реализации параллельной обработки // Компоненты и технологии. 2004. № 6.
2. Долинский М., Толкачев А., Коршунов И. Программный комплекс для разработки параллельных вычислительных систем // Компоненты и технологии. 2004. № 5.