На правах рукописи

Аветисян Арутюн Ишханович

СОВРЕМЕННЫЕ МЕТОДЫ СТАТИЧЕСКОГО И ДИНАМИЧЕСКОГО АНАЛИЗА ПРОГРАММ ДЛЯ АВТОМАТИЗАЦИИ ПРОЦЕССОВ ПОВЫШЕНИЯ КАЧЕСТВА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Автореферат
диссертации на соискание ученой степени
доктора физико-математических наук

Москва – 2012

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования Российской академии наук

Научный консультант:

доктор физико-математических наук, академик РАН
Иванников Виктор Петрович

Официальные оппоненты:

доктор физико-математических наук, чл.-корр. РАН
Воеводин Владимир Валентинович

доктор технических наук, чл.-корр. РАН
Каляев Игорь Анатольевич

доктор физико-математических наук, профессор
Крюков Виктор Алексеевич

Ведущая организация: Федеральное государственное бюджетное учреждение науки Вычислительный центр им. А.А. Дородницына Российской академии наук

Защита диссертации состоится “15” ноября 2012 г. в 15 часов на заседании диссертационного совета Д.002.087.01 при Федеральном государственном бюджетном учреждении науки Институте системного программирования Российской академии наук по адресу: 109004, Москва, ул. А. Солженицына, 25.

С диссертацией можно ознакомиться в библиотеке Федерального государственного бюджетного учреждения науки Институте системного программирования Российской академии наук

Автореферат разослан “___”_____2012 г.

Ученый секретарь
диссертационного совета _____/Прохоров С.П./

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Актуальность. Методы статического и динамического анализа программ были предназначены в первую очередь для разработки оптимизирующих компиляторов: каждая оптимизационная фаза компилятора базируется на одном из таких методов. Первые компиляторы были разработаны в конце 50-х годов прошлого века. Одним из основных критериев качества разрабатываемых программ являлась их эффективность. За пятьдесят лет развития техника оптимизирующей компиляции существенно продвинулась, и для одного вычислительного устройства (ядра) современные оптимизирующие компиляторы обеспечивают такую высокую эффективность объектного кода, что стало возможным почти полностью отказаться от ассемблера при разработке как прикладных, так и системных программ.

В настоящее время методы статического и динамического анализа программ применяются и для решения таких задач, как обратная инженерия, синтез программ по их спецификациям, восстановление программного обеспечения (ПО), анализ и обеспечение различных аспектов безопасности ПО, рефакторинг, анализ и преобразование бинарного кода и др. Связано это с тем, что статический и динамический анализ позволяют накопить сведения о семантике анализируемой программы, необходимые для решения этих проблем.

Современные вызовы связаны с такими долгосрочными тенденциями развития отрасли разработки ПО, как существенное усложнение аппаратуры (параллелизм на всех уровнях, специализированные устройства ускорения вычислений, распределенность), взрывной рост размера программных систем (десятки-сотни миллионов строк кода), массовое внедрение мобильных систем, переход на «облачные» инфраструктуры, доступность практически любых систем через сеть (Интернет, Интранет). В связи с этими вызовами особенно актуальна проблема автоматизации процессов обеспечения качества программных систем на этапах их разработки. Качество современных программных систем определяется тремя взаимосвязанными компонентами: (1) эффективностью; (2) безопасностью; (3) переносимостью. Как правило, в настоящее время обеспечение требуемого качества ПО связано с использованием ручного труда высококвалифицированных специалистов и является искусством. Это существенно увеличивает стоимость и сроки разработки ПО.

Общепризнанно, что одним из перспективных направлений решения проблемы автоматизации процессов обеспечения требуемого качества программного обеспечения является развитие методов анализа программ и разработка соответствующих инструментов.

Рассмотрим проблему автоматизации процессов обеспечения качества ПО подробнее.

Автоматизация процессов обеспечения эффективности. Цель такой автоматизации – это обеспечение требуемого уровня эффективности при минимизации ресурсов, затрачиваемых на разработку и сопровождение. При этом, современные серверы состоят из нескольких процессоров, имеющих многоядерную архитектуру, каждое ядро обладает параллелизмом на уровне

команд. На базе многоядерных процессоров строятся и высокопроизводительные кластеры. В серверах и в узлах кластеров широко используются акселераторы – специализированные процессоры, содержащие тысячи ядер, что позволяет обеспечить высокую степень распараллеливания вычислений (примером акселератора является карта GPU).

Таким образом, аппаратура современных вычислительных систем обеспечивает возможность параллельного выполнения вычислений на всех уровнях: на уровне команд (за счет возможности одновременной выдачи нескольких команд, дублирования функциональных устройств, конвейеризации и векторизации вычислений), на уровне параллельно выполняющихся потоков (каждый поток запускается на отдельном ядре), на уровне параллельно выполняющихся процессов (каждый многопоточный процесс запускается на отдельном узле кластера). Потенциально это может обеспечить очень высокую производительность вычислительных систем.

Однако практическое использование возможностей современной аппаратуры для организации высокоэффективных вычислений требует решения сложных оптимизационных проблем на всех уровнях параллельного выполнения. В результате задача автоматического распараллеливания программ оказывается слишком сложной для современных автоматических методов распараллеливания программ. В настоящее время задача автоматической компиляции параллельных программ успешно решается лишь для узкого класса программ, допускающих параллельное выполнение без синхронизации. В отличие от разработки последовательных программ, где достаточно хорошо развиты технологические средства, в настоящее время не существует устоявшихся технологических процессов, позволяющих разрабатывать эффективные параллельные программы.

Необходимы принципиально новые методы статического и динамического анализа, которые будут использоваться при создании программных и инструментальных средств, поддерживающих автоматизацию процессов разработки эффективных параллельных приложений с учетом особенностей развития аппаратуры.

Автоматизация процессов обеспечения безопасности программно-аппаратных систем.

Развитие сетевых технологий привело к тому, что практически все современные компьютеры (кластеры, серверы, персональные компьютеры, встроенные системы и др.) доступны через сеть (Internet/Intranet). Широкое распространение получила концепция «облачных вычислений», в рамках которой компьютер пользователя является лишь входной точкой, обеспечивающей доступ к мощным вычислительным системам, базам данных и др.

Использование доступа через сеть остро ставит проблемы безопасности, в том числе – проблемы безопасности программного обеспечения. Доступ к компьютерным ресурсам через локальную или глобальную сеть позволяет злоумышленникам, взломав систему защиты, получать конфиденциальную информацию, а также при необходимости парализовать работу жизненно важных информационных инфраструктур. Следует отметить, что основным источником проблем безопасности являются ошибки в системном ПО (как ошибки

кодирования, так и ошибки проектирования). Но поскольку устранить эти ошибки невозможно, приходится использовать разные технологии обеспечения безопасности: антивирусы, защита по периметру, аудит кода и др. Далеко не все проблемы компьютерной безопасности могут быть решены средствами статического и динамического анализа. Но указанные средства позволяют решить ряд проблем, обеспечивающих такие важные аспекты компьютерной безопасности, как, например, обеспечение устойчивости ПО к внедрению программ атакующего.

Устойчивость ПО к внедрению программ атакующего связана с отсутствием в нем «уязвимостей», т.е. таких ошибок разработчика, которые трудно обнаружить на этапе тестирования, но которые влияют на устойчивость работы ПО. Такие ошибки естественно искать в исходном коде программ. Для обнаружения ошибок этого класса широко используются методы статического и динамического анализа программ.

Ранее внедренные программы атакующего (часто их называют «недокументированными возможностями ПО», – далее НДВ) имеют целью воздействовать на атакованную систему по сигналу атакующего или постоянно обеспечивать дополнительные функции системы, не заметные для ее администрации. Как правило, такие программы внедряются в бинарный код, что значительно затрудняет их обнаружение.

Методы статического и динамического анализа дают возможность разработать новые технологии анализа программ (как на уровне исходного, так и на уровне бинарного кода), позволяющие обеспечить обнаружение дефектов (уязвимости, НДВ) и защиту от их использования.

Автоматизация процессов обеспечения переносимости (с сохранением качества приложений)

Современное многообразие процессорных архитектур, как для настольных и серверных, так в особенности и для мобильных систем, влечет актуальность задачи обеспечения переносимости приложений между архитектурами, а часто – и внутри одного семейства архитектур из-за серьезных различий между процессорными реализациями (разница в наборах команд, количестве регистров, особенно векторных, объемах кэш-памяти и т.п.). Эта задача традиционно решается либо пересборкой приложения, написанного на языках общего назначения для новой архитектуры, либо использованием динамической компиляции (например, JavaVM).

Для сохранения качества приложения, то есть в основном производительности, в первом случае возникает необходимость поддерживать десятки бинарных версий приложения, собранных для различных архитектур и вариантов одной и той же архитектуры (например, для разных поколений процессоров x86-64). Эта необходимость связана с разным поведением машинно-зависимых оптимизаций (распределения регистров, планирования и конвейеризации кода, векторизации) даже на вариантах одной архитектуры, причем влияние на производительность программы может достигать десятков процентов.

В случае JavaVM приложение распространяется в промежуточном представлении для некоторой фиксированной архитектуры, а для сохранения производительности дополнительно применяются динамические оптимизации на целевой архитектуре во время выполнения приложения, в том числе с учетом профиля пользователя. Такой подход показал свою жизнеспособность и широко применяется. Однако JavaVM имеет ряд принципиальных ограничений, в частности, JavaVM плохо приспособлена для таких платформ как высокопроизводительные и встроенные системы. Кроме того, для языков общего назначения этот подход неприменим.

Обеспечение переносимости приложений на языках общего назначения, хотя бы в рамках вариантов одного семейства процессорных архитектур, делает необходимым создание новых методов статического и динамического анализа и соответствующей инфраструктуры, позволяющей эффективно применять машинно-независимые оптимизации на стороне разработчика, а машинно-зависимые оптимизации и оптимизации с учетом специфики приложения – на целевой архитектуре.

Цель и задачи работы. Разработка и развитие методов статического и динамического анализа программ, и реализация соответствующих программных и инструментальных средств, обеспечивающих автоматизацию программирования для современных платформ и надежное функционирование программного обеспечения в сетевой среде (Internet/Intranet).

Для достижения поставленной цели в диссертационной работе необходимо решить следующие основные задачи:

- Разработать методы машинно-ориентированной оптимизации программ, позволяющие учитывать особенности функционирования современных процессоров с параллелизмом на уровне команд, с целью максимально использовать преимущества их архитектуры.
- Создать технологический процесс, обеспечивающий высокопродуктивную разработку приложений для систем с распределенной памятью.
- Разработать масштабируемые (анализ большого объема кода в приемлемое время с высокой степенью достоверности) методы статического анализа программ, обеспечивающие нахождение уязвимостей и других дефектов в программах на языках C/C++.
- Разработать методы анализа бинарного кода с целью восстановления алгоритмов и нахождения недокументированных возможностей.
- Разработать технологический процесс, обеспечивающий переносимость программ, написанных на языках общего назначения C/C++, за счет автоматического учета особенностей целевой платформы.

Методы исследования. Для решения поставленных задач использовались методы теории множеств, теории графов, теории отношений, теории решеток, абстрактной интерпретации, теории компиляции, в том числе, динамической компиляции.

Научная новизна. В диссертации получены следующие новые результаты, которые выносятся на защиту:

- Новый метод планирования команд и конвейеризации циклов, учитывающий особенности функционирования современных процессоров (ARM, EPIC), с поддержкой условного и спекулятивного выполнения
- Новые методы разработки JavaMPI-программ, обеспечивающие высокий уровень автоматизации разработки за счет точного прогнозирования на инструментальной машине поведения разрабатываемого приложения с учетом особенностей высокопроизводительной вычислительной системы
- Новые методы статического анализа исходного кода программ, обеспечивающие аудит, нахождение уязвимостей и других дефектов как исходного кода на языках C/C++, так и бит-кода LLVM, в приемлемое время с высокой степенью достоверности
- Новые методы комбинированного статического и динамического анализа бинарного кода, базирующиеся на сборе и анализе трасс, позволяющие восстанавливать алгоритмы и находить недокументированные возможности в защищенном бинарном коде
- Новый метод двухэтапной компиляции, обеспечивающий переносимость программ, написанных на языках общего назначения C/C++, в промежуточном представлении LLVM (бит-код) с использованием динамической компиляции

Практическая значимость работы определяется тем, что на базе разработанных методов реализованы программные средства, обеспечивающие практическое решение рассматриваемых проблем при разработке и анализе программ. Эти средства используются в различных научно-исследовательских, промышленных и других организациях.

- На основе новых методов машинно-ориентированной оптимизации реализован и внедрен в основную ветвь промышленного свободно распространяемого компилятора GCC новый планировщик потока команд, который используется при оптимизации приложений для платформ EPIC (например, Itanium), а также как основа при разработке машинно-ориентированных оптимизаций для других архитектур (например, ARM).
- Инструменты среды ParJava интегрированы в промышленную среду разработки программ Eclipse на основе свободного ПО и применяются для разработки параллельных приложений, в частности, в ИФЗ РАН реализовано приложение моделирования процесса зарождения интенсивных атмосферных вихрей по теории В.Н. Николаевского.
- Инструменты среды Svace интегрированы в промышленную среду разработки программ Eclipse на основе свободного ПО. По контракту с компанией «Самсунг» Svace внедрена в технологический процесс разработки ПО внутри компании (несколько тысяч программистов, десятки миллионов строк кода).

- Интегрированная среда статического и динамического анализа бинарного кода используется для решения практических задач по обеспечению безопасности ПО.
- Метод двухэтапной компиляции на базе LLVM реализован для Linux-платформ на базе архитектур x86 и ARM. По контракту с компанией «Самсунг» на основе двухэтапной компиляции создается «облачное хранилище» приложений нового поколения, обеспечивающее с одной стороны переносимость программ в рамках семейства ARM, а с другой стороны, высокую степень надежности и безопасности приложений, доступных в рамках хранилища, за счет использования среды Svace и среды анализа бинарного кода.

Апробация работы. Основные результаты диссертационной работы докладывались и обсуждались на конференциях и семинарах различного уровня. В том числе, 27 докладов на международных конференциях и 7 на всероссийских. В частности: Всероссийская научная конференция “Высокопроизводительные вычисления и их приложения», г. Черноголовка, 30 октября-2 ноября, 2000; Международный семинар «Супервычисления и математическое моделирование», Саров, 13-15 июня, 2001; Международная конференция “Parallel Computing Technologies (PaCT) Novosibirsk, Sept.3-7, 2001; Computer Science and Information Technologies (CSIT), Yerevan, September 17 – 20, 2001; Международной конференции “Параллельные вычисления и задачи управления” Москва, 2-4 октября, 2001; Parallel Computational Fluid Dynamics 2003, May 13-15, 2003 Moscow; Russian-Indian Workshop on High Performance Computing in Science and Engineering, June 16 – 20, 2003, Moscow; Conf. on Computer Science and Information Technology. September 22-26, 2003, Yerevan; The 10th EuroPVM/MPI conference. LNCS 2840. Sept. 2003, Venice; XII Международная конференция по вычислительной механике и современным прикладным программным системам, июнь – июль 2003, Владимир; Всероссийская научная конференция «Научный сервис в сети Интернет». Новороссийск, 20-25 сентября 2004; Parallel Computational Fluid Dynamics, May 24-27, 2004, Canary Islands, Spain; Fifth Mexican International Conference Avances en la Ciencia de la Computacion ENC’04, 20 – 24 September, Colima, Mexico; Международная научная конференция «Суперкомпьютерные системы и их применение», 26-28 октября 2004, Минск; Всероссийская научная конференция «Научный сервис в сети Интернет: технологии распределенных вычислений». Новороссийск, 19-24 сентября 2005; Computer Science and Information Technologies (CSIT), Yerevan, September 19 – 23, 2005; Всероссийская научная конференция «Научный сервис в сети ИНТЕРНЕТ: технологии параллельного программирования», г. Новороссийск, 18-23 сентября 2006; 13th International Conference On The Methods Of Aerophysical Research. 5 – 10 February, 2007, Novosibirsk; MTPP 2007 Parallel Computing Technologies First Russian-Taiwan Symposium Pereslavl-Zalesskii (Russia), September 2-3, 2007; Parallel Architectures and Compilation Techniques (PACT), Brasov, Romania, September 15-19, 2007; International Conference on the Methods of Aerophysical Research, Novosibirsk, 2007; Sixth International Conference on Computer Science and

Information Technologies (CSIT'2007), 24-28 September, Yerevan, Armenia; Международная научная конференция «Космос, астрономия и программирование» (Лавровские чтения), 20-22 мая 2008 г. Санкт-Петербург; Седьмая международная конференция памяти академика А.П. Ершова Перспективы систем информатики. Рабочий семинар "Наукоемкое программное обеспечение", 15-19 июня 2009 г. Новосибирск; Международная конференция «РусКрипто'2009»; SAMOS 2009 Workshop, 2009; XIX Общероссийская научно-техническая конференция «Методы и технические средства обеспечения безопасности информации». Санкт-Петербург, 05-10 июля 2010 г.; Всероссийская конференция "Свободное программное обеспечение-2010", СПб, 26-27 октября 2010 года; HiPEAC 2010, January 2010.

Гранты и контракты. Работа по теме диссертации проводилась в соответствии с планами исследований по проектам, поддержанным: грантами РФФИ: 06-07-89119-а «Исследование и разработка технологии решения вычислительно-сложных задач на базе распределенных вычислительных ресурсов», 08-01-00561-а "Инструментальная поддержка процесса разработки больших научно-технических приложений", 08-07-00279-а "Исследование и разработка системы автоматического обнаружения дефектов в исходном коде программ", 08-07-00311-а "Исследование и разработка набора инструментальных средств для языков параллельного программирования нового поколения", 08-07-91850-КО_а "Компиляция программ для высокопроизводительных встраиваемых процессоров", 09-07-00382-а "Методология поддержки разработки эффективных параллельных программ в среде ParJava", 10-07-92600-КО_а "Кодогенерация и автоматическая настройка для акселераторов", 11-07-00466-а "Исследование и разработка инструмента динамического анализа программ для автоматического обнаружения дефектов"; контрактами: «Исследование и разработка средств повышения защищённости программного обеспечения от внешних атак» в рамках ФЦП "Исследования и разработки по приоритетным направлениям развития науки и техники на 2002-2006 годы" (ГК № 02.447.11.1004); "Высокоуровневые модели параллельных вычислений и их библиотеки поддержки времени выполнения" (ГК № 07.514.11.4001) и "Проектирование и разработка web-ориентированного производственно-исследовательского центра, ориентированного на решение задач анализа программ" (ГК №.07.514.11.4040) в рамках ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы"; в рамках следующих Программ фундаментальных исследований Президиума РАН: Программа №1 "Проблемы создания научной распределенной информационно-вычислительной среды на основе развития GRID технологий и современных телекоммуникационных сетей", Программа № 4 «Фундаментальные проблемы системного программирования», Программа №14 «Высокопроизводительные вычисления и многопроцессорные системы». Также работы проводились по договорам с зарубежными и отечественными организациями. Кроме того, работа была поддержана грантом Президента Российской Федерации для поддержки молодых российских ученых и ведущих научных школ РФ.

Авторские свидетельства. Свидетельство об официальной регистрации программы для ЭВМ №2005613149. “Программа поиска уязвимостей типа переполнения буфера в исходном коде программ на языке С”. Федеральная служба по интеллектуальной собственности, патентам и товарным знакам. Зарегистрировано в Реестре программ для ЭВМ 05.12.2005 г. Правообладатель: Учреждение Российской академии наук Институт системного программирования РАН. Авторы: Аветисян А.И., Белеванцев А.А., Гайсарян С.С., Журихин Д.М., Маликов О.Р., Мельник Д.М., Несов В.С., Спиридонов С.В.
Свидетельство об официальной регистрации программы для ЭВМ №2006613032. “Среда межпроцедурного анализа программ”. Федеральная служба по интеллектуальной собственности, патентам и товарным знакам. Зарегистрировано в Реестре программ для ЭВМ 31.08.2006 г. Правообладатель: Учреждение Российской академии наук Институт системного программирования РАН. Авторы: Аветисян А.И., Белеванцев А.А., Гайсарян С.С., Журихин Д.М., Маликов О.Р., Мельник Д.М., Несов В.С., Спиридонов С.В.

Личный вклад. Выносимые на защиту результаты получены соискателем лично. В опубликованных совместных работах постановка и исследование задач осуществлялись совместными усилиями соавторов при непосредственном участии соискателя.

Публикации. По материалам диссертации опубликовано 50 работ, в том числе, 1 учебное пособие. 17 статей опубликовано в изданиях, входящих в список изданий, рекомендованных ВАК РФ. Получено 2 свидетельства о регистрации программ для ЭВМ. Наиболее значимые работы перечислены в списке публикаций.

Структура и объем диссертационной работы. Диссертация состоит из введения, шести глав и заключения, изложенных на 271 страницах, списка литературы из 273 наименований, содержит 54 рисунка и 14 таблиц.

СОДЕРЖАНИЕ РАБОТЫ

Во введении обосновывается актуальность исследований, приводятся цель и задачи работы, перечисляются используемые методы исследования, формулируется научная новизна и практическая значимость полученных результатов, приводятся сведения о результатах внедрения и использования.

Первая глава представляет собой аналитический обзор современного состояния методов статического и динамического анализа и их применения для решения проблем продуктивности и безопасности.

Во второй главе рассматриваются методы планирования кода, наилучшим образом учитывающие особенности архитектуры EPIC и поддерживающие конвейеризацию циклов.

В разделе 2.1 описывается новый метод планирования команд и конвейеризации циклов, учитывающий особенности функционирования современных процессоров (ARM, EPIC), с поддержкой условного и спекулятивного выполнения. Особенностями предложенного метода, базирующимися на схеме селективного планирования, являются: четкое разделение шагов по выявлению и использованию параллелизма, позволяющее легко добавлять новые преобразования команд и эвристики по выбору наилучшей для планирования команды; поддержка нескольких точек планирования делает равноправными команды из разных базовых блоков и путей выполнения, что позволяет строить в среднем лучшие расписания для регионов планирования без четко выраженных «горячих» путей; организация конвейеризации циклов на основе существующей инфраструктуры планировщика. В целом предлагаемый метод содержит следующие основные шаги:

1. Построение регионов планирования (ациклических подграфов графа потока управления) по данной процедуре программы.
2. Планирование каждого региона отдельно:
    a. Обход фронтом планирования (набором точек планирования) региона в топологическом порядке.
    b. Для каждой точки планирования, при наличии свободных ресурсов:
        i. Выявление параллелизма (сбор доступных для планирования команд в данной точке).
        ii. Использование параллелизма (выбор лучшей команды с помощью набора эвристик или переборных алгоритмов).
        iii. Планирование выбранной команды. При этом обновляются структуры данных планировщика, необходимые для первых двух шагов по сбору команд и выбору лучшей.

При выявлении параллелизма основную функциональность обеспечивает ядро планировщика, выполняющее сбор готовых для планирования команд (возможно, с учетом одного или нескольких преобразований) и перенос команды в точку планирования с поддержанием корректности программы (шаги 2b-i и 2b-iii). Ядро реализует два базовых преобразования команд – *клонирование* и *унификацию* команд, используя в основном данные анализа зависимостей для выполнения своих действий. Остальные преобразования команд обеспечиваются *модулями* расширения – это *переименование регистров* и связанный с ним *перенос через копии*, *спекулятивное* выполнение, *условное* выполнение. Модули обращаются к ядру за необходимой информацией о свойствах команд, жизни регистров и т.п. для проведения преобразования, при этом модули могут добавлять свою информацию в структуры данных, поддерживаемые ядром. Конвейеризация циклов также может рассматриваться как модуль, но с точки зрения корректности

получаемой программы в основном обеспечивается правильным построением регионов и выбором их порядка обхода.

Использование параллелизма (шаг 2b-ii) заключается в поддержке набора эвристических алгоритмов, сортирующих собранные команды в соответствии с заданным приоритетом, после чего для выдачи выбирается наиболее приоритетная команда. Возможен дополнительный перебор элементов множества готовых команд для упорядочивания выдаваемых команд в пределах одного цикла. Как и в случае модулей, применение эвристик, основанных на информации о критическом пути команды, относительных вероятностях выполнения отдельных команд, жизни регистров и т.п., требует обращения к ядру для поддержания всех необходимых данных в актуальном состоянии.

В разделе 2.2 рассматривается улучшения предложенного в предыдущем разделе метода, основанного на алгоритме селективного планирования, и реализация метода в компиляторе GCC. Схема алгоритма селективного планирования в том виде, как она изложена в литературе, не позволяет применить ее в реальном компиляторе (GCC) языков Си, Си++ и Фортран, а также применить ее к архитектурам с явным параллелизмом команд. Основными причинами этого являются:

- Предполагаемая применимость основных описанных преобразований команд ко всем командам внутреннего представления компилятора;
- Отсутствие механизма выбора лучшей на данном шаге команды для планирования из множества доступных команд;
- Большая сложность, выражающаяся в значительном росте времени компиляции на реальных программах (первоначальная версия алгоритма, реализованная нами для компилятора GCC, замедляла его в два раза);
- Отсутствие преобразований, необходимых для выявления параллелизма на современных архитектурах с явным параллелизмом команд типа Itanium (спекулятивное, условное выполнение) и на встраиваемых архитектурах типа ARM (условное выполнение).

В частности, предлагаются более эффективные структуры данных планировщика, основанные на понятии *виртуальной* команды, содержащей потоково-нечувствительную информацию и служащей контейнером данных, обеспечивающим минимальные затраты памяти за счет ленивого копирования. Виртуальные команды также отражают в своем описании степень применимости основных преобразований команд ко всем командам внутреннего представления.

Описывается улучшенный способ анализа зависимостей для поддержки реальных программ, дающий возможность динамического пересчета зависимостей после изменения планируемого кода. Это улучшение достигается за счет двух идей: кэширования *контекста* анализатора зависимостей, т.е. такого описания состояния модифицированных регистров и ячеек памяти последними выполненными командами, по которому можно построить зависимости между ними и текущей обрабатываемой командой, и реализации *событий* анализатора зависимостей, позволяющих гибко отреагировать на найденную зависимость по

памяти или по регистру в любой части команды. С помощью событий анализатора зависимостей можно вычислять приоритеты команд, а также обеспечивать готовность входных данных команды.

Предложенный в селективном планировании прием сохранения промежуточных множеств готовых команд в начале каждого базового блока оказывается недостаточным для преобразованных команд, так как при поиске необходимо отменять сделанные преобразования ровно в тех точках региона планирования, при перемещении через которые эти преобразования были впервые применены. Для ускорения поиска предлагается механизм кэширования преобразований, который при выполнении преобразования некоторой команды записывает обе формы команды – изначальную и преобразованную – в вектор *истории преобразований*, сохраняемый для каждой команды из региона. При поиске запланированной команды, зная текущую обрабатываемую команду региона, легко найти в кэше старую версию искомой команды.

Рассматриваются модификации стандартных преобразований, таких, как переименование регистров и конвейеризация. При переименовании регистров необходимо учитывать регистры, которые нельзя использовать в качестве целевых из-за ограничений архитектуры, а также регистры, содержащие значения, необходимые на других путях выполнения. При конвейеризации команд необходимо корректно планировать созданный компенсационный код как в заголовке, так и в теле конвейеризуемого цикла, обеспечить конечность процесса конвейеризации, ограничив количество попыток запланировать команду. Наиболее важным улучшением является оценка выгодности перемещения команд при конвейеризации, т.к. в итеративном процессе планирования сложно оценить влияние конвейеризованных команд на расписание цикла в целом. Для уплотнения появляющихся при конвейеризации “пустот” в расписании используется дополнительный короткий проход планировщика над частями конвейеризованного цикла, но без повторной конвейеризации. Для спекулятивного выполнения вводятся эвристики, позволяющие увеличить промежуток между спекулятивными загрузкой и проверкой, а также предсказать задержку на команде проверки и отказаться от проведения спекулятивного преобразования.

Приводятся особенности внедрения разработанного метода планирования в компилятор GCC, связанные с модификациями кодогенератора и эвристик планировщика.

Далее описывается выполненная поддержка спекулятивных преобразований в разработанном планировщике. При переносе промежуточного множества команд через текущую посещаемую команду устраняемые зависимости по данным между проносимой и посещаемой командами удаляются, а переносимая инструкция преобразуется к спекулятивному виду. Зависимости по управлению устраняются аналогичным образом при переносе команд через условный переход. Для отслеживания вероятностей зависимостей необходимы изменения в анализаторе зависимостей, запоминающие статус зависимости, который кодирует как тип зависимости, так и ее вероятность.

При сборе доступных команд формируется множество потенциальных кандидатов на спекулятивное перемещение, а при выборе наилучшей команды модифицируются эвристики выбора с учетом найденных вероятностей зависимостей по данным. Дополнительно отдается предпочтение командам, перенесенным через меньшее количество точек слияния потока управления, так как их перемещение создает меньше компенсационных копий.

При перемещении команды генерируется ее спекулятивная версия и команда спекулятивной проверки. Для этого дополнительно был доработан кодогенератор архитектуры IA-64 в компиляторе GCC.

Полученные экспериментальные результаты на платформе Intel Itanium показывают среднее ускорение для тестов SPEC CPU FP 2000 в 3.5%, на некоторых тестах – до 11% по сравнению со старым планировщиком GCC.

Основной прирост производительности дает конвейеризация циклов, в которой важную роль играет поддержка спекулятивного выполнения. Это связано с тем, что конвейеризация в предложенном планировщике обрабатывает циклы с неизвестным числом итераций, т.е. возможна выдача команд из "лишней" (следующей за последней) итерации цикла, и для подавления исключительных ситуаций в таких командах – например, при выходе за границу обрабатываемого массива – необходима поддержка спекулятивного выполнения на уровне аппаратуры. Кроме того, для условного перехода, организующего циклическое выполнение, легко предсказать направление перехода, поэтому вероятность успеха выданной при конвейеризации спекулятивной загрузки велика.

В разделе 2.3 описывается разработанное преобразование поддержки условного выполнения команд в предлагаемом планировщике. Для добавления поддержки команд с условным выполнением решаются следующие подзадачи:

- Вычисление предиката условно выполняющейся команды, сохранение его в атрибутах команды и выполнение преобразования;
- Добавление условно выполняющихся команд во множество готовых команд (это происходит во время объединения множеств доступных команд от обеих ветвей на команде условного перехода);
- Обеспечение корректности при переносе выбранной в условно выполняющейся форме команды к границе планирования (на этапе поиска изначальных команд при спуске через команду условного перехода необходимо преобразовывать команды из условной формы в обычную, при этом поиск необходимо продолжать только на ветви, соответствующей предикату выбранной команды; при создании компенсационного кода вместо удаления изначальной команды к ней добавляется предикат, противоположный тому, с которым команда запланирована);
- Обеспечение корректности преобразований, применяемых селективным планировщиком к команде в условно выполняющейся форме (поддержка подстановки через условное присваивание, если выполнение модифицируемой команды контролируется тем же предикатом; поддержка переименования регистров в условной команде;

конвейеризация циклов с использованием условного выполнения – перенос команд через обратную дугу цикла не приводит к увеличению количества выполняемых команд, а необходимость в использовании переименования регистров и спекулятивных загрузок возникает реже).

В полученных экспериментальных результатах в целом производительность набора тестов вещественных вычислений увеличилась на 2%, пять тестов показали заметный рост производительности (на 5% и более), несколько тестов замедлились, но не более, чем на 2.5%. Наибольший прирост производительности объясняется улучшенной конвейеризацией циклов, не требующей затрат на спекулятивные команды проверки.

В третьей главе рассматривается проблема повышения продуктивности параллельных вычислений на вычислительных системах с распределенной памятью (высокопроизводительных кластерах). Фактическим стандартом разработки приложений для таких систем является последовательный язык программирования, расширенный стандартной библиотекой передачи сообщений (как правило, это библиотека *MPI*), которая содержит разнообразные возможности организации обмена данными между параллельными ветвями программы. В частности, это связано с тем, что для организации эффективных вычислений требуется различные процедуры обмена для различных классов приложений. Таким образом, эффективность прикладной *MPI*-программы существенно зависит от того, какие процедуры библиотеки *MPI* в ней использованы и как размещены в программе обращения к этим процедурам. В отсутствие оптимизирующих компиляторов выбор процедур библиотеки *MPI* и расстановка обращений к ним осуществляются разработчиком вручную. Таким образом, фрагменты параллельной программы, от которых в наибольшей степени зависят ее масштабируемость и эффективность, разрабатываются практически на ассемблерном уровне. В этих условиях повышение продуктивности может быть достигнуто путем разработки инструментальных средств, позволяющих разработчику сократить время разработки и сопровождения прикладной *MPI*-программы. Один из возможных наборов таких инструментов предлагает интегрированная среда *ParJava*.

Для правильного выбора процедур библиотеки *MPI* и правильного размещения обращений к ним необходимо многократно получать достаточно точную оценку времени выполнения всей разрабатываемой *MPI*-программы и некоторых ее фрагментов. Для *JavaMPI*-программ такие оценки могут быть получены во время их выполнения на кластере с использованием подсистемы генерации временного профиля среды *Java*, но это связано с большими затратами как машинного времени, так и времени прикладного программиста, что существенно понижает продуктивность разработки. Кроме того, подсистема генерации временного профиля ориентирована на программы, выполняемые в одном процессе, и не позволяет точно измерить время обмена данными между процессами *JavaMPI*-программы. Интегрированная среда *ParJava* позволяет автоматически построить модель разрабатываемой *JavaMPI*-программы и

получать необходимые оценки времени выполнения, используя интерпретацию модели на инструментальном компьютере.

В разделе 3.1 рассматривается модель *JavaMPI*-программы *Progr*, выполняемой в *K* независимых параллельных процессах (на кластере), которая по определению представляет собой *K* одинаковых *логических* (*модельных*) *процессов*. С каждым логическим процессом *p* программы *Progr* ($0 \le p \le K$-1) связано *модельное время*, определяемое по показаниям модельных часов этого процесса. Модельные часы логического процесса *p* реализуют отображение $T^p(V)$ фрагментов модели программы на моменты времени. Интерпретация модели программы *Progr* на интерпретаторе среды *ParJava* позволяет определять модельное время выполнения каждого структурного фрагмента программы *Progr* и всей программы в целом.

Поскольку *JavaMPI*-программа разрабатывается в модели *SPMD*, достаточно определить модель одного процесса, которая определяется как совокупность моделей всех классов, входящих в состав *Progr*. Модель каждого класса – как множество моделей всех методов этого класса; кроме того, в модель класса включается модель дополнительного метода, описывающего поля класса, его статические переменные, а также инициализацию полей класса и статических переменных. Модель метода представляется в виде пары ⟨модель потока управления, модель вычислений⟩.

Модель потока управления (МПУ) представляет собой модифицированное дерево управления метода, полученное из графа потока управления путем выделения и сворачивания областей. Внутренние узлы представляют операторы управления моделируемой программы, а листовые - ее базовые блоки. Каждый внутренний узел МПУ представляет собой дескриптор области, в состав которой входят все операторы метода, соответствующие узлам, входящим в состав поддерева с корнем в этом узле. В качестве базовых блоков в МПУ рассматриваются не только линейные последовательности вычислений (вычислительные базовые блоки), но также и вызовы библиотечных, пользовательских и коммуникационных методов. Каждому базовому блоку *B* сопоставлен динамический атрибут – модельное время его выполнения $Time(B) = T^p_{exit}(B) - T^p_{entry}(B)$, соответствующее разности показаний модельных часов в начале и в конце выполнения блока *B*. Соответственно каждому внутреннему узлу *V* сопоставляется модельное время *Time*(*V*) выполнения части метода, описываемой поддеревом с корнем *V*. В частности, модельное время *Time*(*f*) выполнения метода *f* равно *Time*(Bd_f), где Bd_f – тело метода *f*, представленное как один составной оператор.

Модель вычислений содержит байт-код каждого базового блока моделируемого метода, что позволяет интерпретировать ее на *JavaVM*. По окончании интерпретации очередного базового блока модели вычислений вызывается интерпретатор среды *ParJava*, который, с помощью МПУ, определяет очередной базовый блок, который следует интерпретировать.

Следует отметить, что модель *MPI*-программы, в которой внутренние узлы соответствуют ее операторам, адекватно отражает свойства моделируемой программы только для среды программирования *Java*, в которой структура программы, выполняемой на *JavaVM* (байт-кода), практически не отличается от структуры исходной программы. Для других традиционных языков программирования (*C*/*C++*, *Fortran*), модель должна строиться не над исходным кодом программы, а над ее промежуточным представлением с учетом результатов статической оптимизации, так как оптимизаторы современных компиляторов существенно меняют структуру программы.

Раздел 3.2. посвящен проблеме оценки времени, затрачиваемого на передачу данных между процессами *JavaMPI*-программы. Каждый обмен данными описывается с помощью функций библиотеки *MPI*, время выполнения которых определяется не только параметрами узлов кластера, но и параметрами его коммуникационной сети. В диссертации разработаны реализованные в среде *ParJava* модели коммуникационных функций библиотеки *MPI*, позволяющие оценить время, затрачиваемое на передачу данных между *MPI*-процессами. Модель каждой коммуникационной функции описывается с помощью восьми *базовых операций обмена*, выделенных при анализе структуры библиотеки *MPI*: `Init`, `Free`, `Pack`, `Unpack`, `Post`, `Get`, `Copy`, `Wait`. В работе показано, что для оценки времени выполнения этих операций следует использовать технические характеристики сети кластера (например, *Myrinet, InfiniBand* и т.п.).

В разделе 3.3 рассматривается проблема моделирования так называемых «горячих» методов *JavaMPI*-программы. Как известно, в процессе выполнения *Java*-программы на интерпретаторе строятся оценки ее частотного и временного профилей, что позволяет выявить «горячие» методы этой программы («горячими» называются методы, на выполнение которых тратится большая часть времени выполнения). В процессе дальнейшего выполнения программы на *JavaVM* ее «горячие» методы поступают на вход оптимизирующего динамического компилятора времени выполнения (*JIT*-компилятора), и их код заменяется эквивалентным «агрессивно» оптимизированным «родным» кодом узла кластера, на котором выполняется программа. Считается, что «горячие» методы составляют не более 15-20% всех методов программы, так что их оптимизация и компиляция не занимает много времени. Поскольку «агрессивная» оптимизация может существенно изменить граф потока управления моделируемой программы, в моделях «горячих» методов должны быть учтены изменения, вносимые *JIT*-компилятором.

Учет влияния динамического компилятора на выполнение *JavaMPI*-программы производится, исходя из следующих предпосылок:

1. Динамический компилятор оптимизирует только «горячие» методы.
2. Динамический компилятор существенно изменяет как отдельные базовые блоки, так и весь граф потока управления «горячих» методов.
3. Динамический компилятор выполняется только один раз.

В среде *ParJava* предусмотрена перестройка моделей «горячих» методов *JavaMPI*-программы после окончания работы *JIT*-компилятора. Такая

перестройка моделей методов *JavaMPI*-программы является частью интерпретации *JavaMPI*-программы. Для перестройки моделей «горячих» методов используется их бинарное представление, сгенерированное *JIT*-компилятором.

В разделе 3.4 определяется и обсуждается *процесс интерпретации модели JavaMPI-программы*, в среде *ParJava*.

По определению интерпретация каждого логического процесса модели *JavaMPI*-программы выполняется независимо от интерпретации других логических процессов и начинается с интерпретации модели метода `main()` одного из классов, входящих в состав указанной программы (имя этого класса и, в случае необходимости, значения параметров метода `main()` указываются пользователем перед началом интерпретации). В процессе интерпретации каждого внутреннего узла V модели определяется модельное время выполнения этого узла $Time(V)$. При этом модельным временем выполнения метода `f()` будет $Time(V_f)$, а оценкой времени выполнения всей программы - $Time(V_{main})$, т.е. оценка времени выполнения метода `main()`, с которого начался процесс интерпретации.

Начальные показания модельных часов всех логических процессов устанавливаются равными нулю. Показания модельных часов логического процесса обновляются после интерпретации в этом процессе очередного базового блока путем добавления значения атрибута *Time* указанного базового блока к текущему значению модельного времени. Интерпретация работы коммуникационных функций использует показания модельных часов различных процессов для выбора сценария выполнения коммуникации и определения ее продолжительности. Во время интерпретации коммуникационной функции считаем, что интерпретатор параллельной программы определил логические процессы, участвующие в коммуникации, и доступны показания модельных часов каждого логического процесса.

При интерпретации внутренних узлов показания модельных часов не меняются, так как интерпретация внутреннего узла сводится к выбору очередного базового блока, который необходимо интерпретировать. Следовательно, если в логическом процессе p базовый блок B_2 интерпретируется непосредственно после базового блока B_1, то $T^p_{entry}(B_2) = T^p_{exit}(B_1)$. В МПУ каждый базовый блок представлен своим дескриптором, который содержит модельное время выполнения этого базового блока, определенное до начала интерпретации на целевой вычислительной системе, и ссылку на байт-код базового блока, что позволяет интерпретировать модель вычислений программы на *JavaVM*. Интерпретация модели вычислений используется только для определения текущих значений переменных, используемых в МПУ для выявления очередного узла модели, который следует интерпретировать. Если список таких переменных в дескрипторе базового блока пуст, блок не подлежит интерпретации. Во всех случаях в качестве оценки модельного времени выполнения базового блока берется оценка из его дескриптора.

Каждый логический процесс модели *JavaMPI*-программы интерпретируется на инструментальном компьютере в отдельном потоке.

Таким образом, для интерпретации метода (функции) *JavaMPI*-программы МПУ этого метода передается интерпретатору среды *ParJava*, который, интерпретируя в требуемом порядке внутренние узлы МПУ, вычисляет модельное время, необходимое для выполнения этого метода. Каждый процесс (поток) содержит свой экземпляр интерпретатора среды *ParJava*. Чтобы определить модельное время, необходимое для выполнения *JavaMPI*-программы на P процессорах, в каждом логическом процессе p определяется модельное время $Time_p$(`main()`) выполнения метода `main()` и в качестве требуемой оценки времени берется величина $\max_p(Time_p(\texttt{main()})$.

Аналогичным образом определяются оценки времени выполнения фрагментов (например, методов, или областей) *JavaMPI*-программы. В работе показано (**утверждение 3.7**), что относительная погрешность оценки модельного времени выполнения программы не превосходит наибольшей относительной погрешности оценки модельного времени выполнения ее базовых блоков.

В разделе 3.5 рассматриваются проблемы частичной интерпретации модели. Частичная интерпретация позволяет многократно интерпретировать только те методы или их части, которые интересуют разработчика в данный момент, заменив остальные части метода редуцированными блоками.

Редуцированный блок состоит только из дескриптора, в котором учтено значение модельного времени с требуемой точностью (указанное значение получено во время предыдущих интерпретаций программы). Интерпретация редуцированного базового блока тривиальна и состоит в учете вклада этого блока в модельное время.

По определению, результатом применения операции редукции к базовому блоку B является базовый блок B_{red}, которому соответствует пустая последовательность интерпретируемых выражений (все выражения были вычислены во время предыдущих интерпретаций и их значения известны) и значение оценки модельного времени которого – известная (по предыдущим интерпретациям) константа. Результатом применения операции редукции к внутреннему узлу V является базовый блок B_{red}, значение оценки модельного времени которого было вычислено во время предыдущих интерпретаций модели программы. Сформулировано достаточное условие корректности применения операции редукции к внутреннему узлу V (**утверждение 3.1**): редукция произвольного внутреннего узла V корректна, если узел V *замкнут по коммуникациям,* то есть при интерпретации узла V не должно оставаться непринятых сообщений, и все операции приема должны завершаться.

Отметим, что хотя при интерпретации модели параллельной программы выполняется намного меньший объем вычислений, чем при выполнении самой программы, интерпретация может занимать значительное время. Время интерпретации увеличивается, прежде всего, в связи с тем, что интерпретация выполняется на инструментальной машине, производительность которой меньше, чем у целевой системы. Кроме того, при большом количестве процессов в

исходной программе моделирующие их потоки будут выполняться в псевдопараллельном режиме, что еще больше увеличит время интерпретации. Редукция позволяет использовать результаты интерпретации *JavaMPI*-программы для упрощения ее модели, что делает возможным интерпретировать ее «по частям»: определять время работы только интересующих фрагментов программы, один раз интерпретировать многократно использующиеся части программы, а также использовать результаты интерпретации частей программы при интерпретации всей программы, сокращая время одного сеанса интерпретации.

В разделе 3.6 исследуется проблема прогноза времени выполнения *JavaMPI*-программы при использовании заданного числа узлов кластера.

Интерпретация *JavaMPI*-программы может быть существенно ускорена, если во время ее предварительного прогона оценивать время выполнения не только вычислительных базовых блоков, но и более крупных фрагментов: отдельных итераций циклов, циклов в целом, а в некоторых случаях – и гнезд циклов. В качестве крупных фрагментов программы удобно рассматривать *области*, т.е. подграфы графа потока управления метода, имеющие единственный входной базовый блок, доминирующий над всеми остальными блоками области, и не более одной обратной дуги, конец которой указывает на входной базовый блок. Легко видеть, что циклы и гнезда циклов языка *Java* являются областями: в качестве входного базового блока в этом случае можно рассматривать заголовок цикла.

Вводится понятие *простой области*: область называется простой, если время ее выполнения не зависит от параметров метода, в состав которого она входит (например, область не содержит ветвлений, или, при наличии ветвлений, время выполнения каждой трассы примерно одинаково). Каждая простая область может быть редуцирована. Если тело цикла является простой областью, то и весь цикл является простой областью и может быть редуцирован. Если тело метода является простой областью, то такой метод может быть редуцирован.

Если в состав области входят базовые блоки, соответствующие обращениям к функциям библиотеки *MPI*, то такая область не является простой, так как время выполнения функции библиотеки *MPI* не может быть определено точно. Это следует из того, что *MPI*-процессы, между которыми осуществляется обмен данными, асинхронны, и невозможно точно определить не только время выполнения функций обмена данными, но и момент начала их выполнения. Рассматриваются методы измерения времени выполнения циклов в различных частных случаях. Среда *ParJava* реализована и использовалась при решении ряда прикладных задач.

В разделе 3.7 рассмотрена модельная задача решения линейного уравнения теплопроводности и реальная задача моделирования процесса зарождения интенсивных атмосферных вихрей, выполненная в Институте физики Земли РАН. Показано, как среда *ParJava* обеспечила возможность получения достаточно точных оценок границы (погрешность не более 10%) области масштабируемости в каждом из рассмотренных случаев.

В четвертой главе рассматривается проблема поиска уязвимостей и других дефектов (далее – дефектов) в исходных кодах программ с помощью статического анализа и описывается среда анализа Svace, обеспечивающая, как на этапе аудита кода, так и на этапе его разработки, нахождение соответствующих дефектов в ПО, содержащем десятки миллионов строк кода, с высокой степенью достоверности.

Важной особенностью статического анализа является то, что анализируется программа целиком, в том числе, редко достигаемые участки кода, что позволяет найти ошибки, которые обычно остаются незамеченными в ходе тестирования. Результатом статического анализа является список предупреждений с указанием типа дефекта, места возможной ошибки в исходном коде и данных о том, как именно может произойти ситуация, описанная в предупреждении.

В разделе 4.2 рассматриваются теоретические аспекты поиска дефектов с помощью статического анализа. Предложены следующие требования к методам и алгоритмам анализа:

- анализ должен производиться автоматически, не требуется специально подготавливать исходный код, либо вмешиваться в ходе анализа;
- анализ должен наиболее полно учитывать контекст исследуемой процедуры;
- должна обеспечиваться масштабируемость – возможность анализа программ размером в миллионы строк кода и сотни тысяч функций за приемлемое время (для большинства программ время анализа не должно превышать 12 часов);
- необходимо обеспечить возможность анализа при отсутствии исходного кода всей программы (например, используемых библиотек);
- большая часть предупреждений должна быть истинной, т.е. корректно отражать найденные в коде ситуации, являющиеся проявлениями искомого дефекта.

Для достижения этих требований в работе предложен новый класс эвристик, реализованный с помощью новых методов статического анализа исходного кода программ. Предложенные эвристики, базирующиеся на понятии абстрактного объекта памяти и его контекста, позволяют для каждой процедуры выполнять итерационный статический анализ, вычисляющий совокупность значений необходимых анализу атрибутов объектов памяти (*контекст* процедуры). Анализ является межпроцедурным и управляется графом вызовов. Для обеспечения высокой скорости анализа и его масштабируемости используется оригинальные эвристики обхода графа вызовов, поддерживаемые механизмом аннотаций.

Аннотация является структурой данных, связанной с процедурой анализируемой программы, которая абстрактно описывает эффект от выполнения данной процедуры для ее произвольного вызова – побочные эффекты, возвращаемые значения, способы использования и изменения параметров и других доступных процедуре данных. Аннотации позволяют исключить часть вершин графа вызовов из анализа, что решает фундаментальную проблему обеспечения масштабируемости и сокращения времени анализа без существенных потерь в доле истинных предупреждений.

Для обеспечения возможности анализа при отсутствии исходного кода всей программы введен механизм спецификаций, обеспечивающий возможность описания свойств функций (например, стандартных библиотечных функций языка Си), необходимых во время анализа. Эти спецификации пишутся на языке Си в виде заглушек, в которых указаны только полезные для анализа операции и позволяют получать необходимые аннотации.

Предложенные эвристики позволяют выявлять такие классы дефектов, как использование неинициализированных данных, переполнение буфера, уязвимости форматной строки, разыменование нулевого указателя, утечка памяти и многое другое.

В разделе 4.3 описывается архитектура среды Svace и особенности реализации ее текущей версии, которая позволяет анализировать программы, написанные на языках Си и Си++. Инструмент анализа оперирует файлами с внутренним представлением, сгенерированными компилятором из файлов исходного кода программы и содержащими описания всех используемых типов, глобальных и локальных переменных модуля компиляции, а также текст функций кода для абстрактной регистровой машины. Используется компилятор LLVM-GCC на основе открытого компилятора GCC, генерирующий внутреннее представление для LLVM – популярной системы для построения компиляторов. Представление LLVM (биткод) компактно, его можно сохранять на диске как в бинарном, так и в текстовом виде, добавлять к коду программы метаданные, компоновать и т.п. Структура биткода специально разрабатывалась для быстрой загрузки и поиска необходимых фрагментов.

Статический анализатор состоит из ядра и набора детекторов – компонент, осуществляющих поиск конкретных типов предупреждений и реализующих эвристики, необходимые для обнаружения ситуаций соответствующего вида. Ядро предоставляет интерфейсы прикладного уровня, обеспечивающие реализацию детекторов. Помимо биткод-файлов, при анализе могут использоваться пользовательские спецификации библиотечных функций, которые также преобразуются в биткод-файлы.

Представление программы в виде биткод-файлов и использование инфраструктуры промышленного компилятора LLVM обеспечивает автоматический анализ, без специальной подготовки исходного кода и без вмешательства в ход анализа.

В разделе 4.4 описывается ход статического анализа. Анализатор запускается над множеством сгенерированных биткод-файлов и множеством спецификаций библиотечных функций, необходимых для анализа и также хранящихся в виде биткод-файлов. Управление ходом анализа ведет ядро, вызывающее детекторы по мере надобности. После окончания анализа сгенерированные предупреждения сохраняются в текстовом виде и внутреннем формате. Компонент Eclipse позволяет просматривать результаты с привязкой к исходному коду программы.

Аннотация создается автоматически как результат анализа процедуры. Детекторы могут сохранять в аннотации необходимые им данные для последующего анализа и выдачи предупреждений. Необходимые анализатору спецификации стандартных библиотечных функций (например, языка Си)

пишутся также на Си в виде заглушек. Исходный код спецификаций транслируется компилятором LLVM-GCC, и результат поставляется в виде биткод-файлов. Эти файлы анализируются до начала анализа остальных файлов, в результате чего создаются аннотации соответствующих библиотечных функций.

Статический анализ программы происходит в четыре этапа. На первом этапе все биткод-файлы считываются по очереди в произвольном порядке, и строится общий граф вызовов программы. На втором этапе граф вызовов обходится в обратном топологическом порядке, так что каждая процедура посещается после того, как были посещены все вызываемые из нее процедуры. Для каждой посещенной в этом порядке обхода процедуры программы выполняется внутрипроцедурный анализ. На третьем этапе выполняется специфический для Си++ анализ кода, исследующий взаимодействие методов классов. На четвертом этапе принимаются решения о выдаче либо исключении некоторых предупреждений на основании собранных в ходе основного анализа статистических данных и фрагментов исходного кода, соответствующих местам потенциальной выдачи предупреждений.

В ходе анализа конкретной процедуры инструменту доступно внутреннее представление лишь этой процедуры и данные о вызываемых ей процедурах, присутствующие в виде аннотаций. В результате анализа процедуры создается ее аннотация, которая может использоваться в дальнейшем. Размер аннотаций ограничен сверху, поэтому в ходе анализа каждой процедуры обрабатывается ее внутреннее представление и ограниченный дополнительный объем информации, не зависящий от полного размера программы. Таким образом, анализ всей программы масштабируется линейно.

При анализе процедуры строится ее граф потока управления, после чего проводится потоково-чувствительный анализ, аналогичный анализу потока данных, при этом после нескольких проходов сверху вниз анализ завершается проходом снизу вверх. С каждой дугой графа потока управления ассоциируется *контекст* – информация о потоке данных, установленная для путей выполнения, проходящих через данную дугу. Анализ потока данных происходит путем «продвижения» контекста по дуге, входящей в инструкцию, через эту инструкцию и построение контекста на выходе из инструкции, основываясь на том, как инструкция изменяет состояние памяти. Для инструкции вызова продвижение контекста заключается в получении аннотации вызываемой процедуры и использовании этой аннотации для построения выходного контекста.

Каждый детектор может вводить новые атрибуты для отслеживания необходимых свойств данных программы и выдачи предупреждений. Чтобы определить способы продвижения и слияния новых атрибутов, детекторы описывают обработчики этих событий, которые вызываются инфраструктурой анализа, и специальные процедуры манипуляции атрибутами, которые используются при написании спецификаций. Обработчики событий имеют доступ к инструкции внутреннего представления и всем элементам контекста.

В разделе 4.5 описываются экспериментальные результаты. После окончания анализа его результаты могут быть просмотрены пользователем в среде Eclipse. Для каждого предупреждения показывается сообщение и релевантные места в исходном коде программы (например, может быть указана полная межпроцедурная трасса по процедурам программы, приводящая к ошибке).

Тестовый пакет из 23 проектов (1,5 миллиона строк кода, пакеты busybox, openssh, postgresql-8.4, gtk+2.0, openssl, sqlite3 и др.) анализируется менее, чем за 2 часа. Время на анализ многих небольших проектов занимает менее минуты.

Полные результаты оценки разработанного инструмента статического анализа не могут быть раскрыты как полученные в ходе работы по договорам с коммерческими компаниями. В целом из 15 групп оцениваемых предупреждений средняя доля истинных предупреждений составила 67% (от 25% до 100%). Более высокая доля истинных предупреждений (50-85%, в зависимости от типа) наблюдается у групп предупреждений о возможном разыменовании нулевого указателя и небезопасном использовании контролируемых внешним вводом данных. Более низкая доля (40-60%, в зависимости от типа) у предупреждений об утечках памяти, использовании неинициализированных данных или освобожденной памяти, и переполнении буфера.

При сравнении результатов анализа с коммерческими инструментами, присутствующими на рынке, можно заметить, что доля истинных предупреждений в среднем схожа с разработанным инструментом, а также пересечение найденных ошибок для некоторых классов ошибок достаточно велико (50-80%), из чего можно сделать вывод, что количество пропусков истинных ошибок незначительно.

В пятой главе рассматривается подход к анализу бинарного кода, состоящий в комбинации статического и динамического анализа, а также методы и программные средства, позволяющие его автоматизировать. Основной задачей, в которой появляется необходимость в анализе бинарного кода, является задача понимания программ. Необходимость решения этой задачи возникает в целом ряде практических ситуаций: при анализе вредоносного кода, когда исходный код отсутствует (исходная программа написана на языке ассемблера), либо недоступен; при поиске НДВ в коммерческом ПО, которое поставляется без исходных кодов; при поддержке унаследованных приложений, в процессе которой бинарный код приложения в результате многочисленных исправлений перестаёт соответствовать исходному коду и, например, в случае обнаружения ошибки требуется анализ бинарного кода.

При работе с бинарным кодом применимы методы статического анализа, описанные в предыдущей главе. Однако для работы требуется, чтобы программа была представлена в виде набора графов: графа вызовов, определяющего связи по вызовам между отдельными функциями программы, графов потока управления, каждый из которых описывает поток управления функции, входящей в состав программы. Задача получения такого представления по машинному коду в ряде случаев может быть достаточно сложна.

Бинарный код программы, как правило, представляется в виде исполняемого файла некоторого известного формата (например, PE32 для ОС Windows или ELF для Linux). Предполагается, что исполняемый файл удовлетворяет «стандартной модели компиляции»: состоит из области кода и области статических данных, таблицы функций, импортируемых программой из сторонних динамически загружаемых библиотек, а также списка функций, экспортируемых программой, в случае, если программа сама является библиотекой. Область кода содержит последовательность функций программы. Библиотечные функции, которые при сборке программы были статически слинкованы с ней, содержатся в её коде и, вообще говоря, неотличимы от функций программы.

Модель программы может быть представлена в виде двух областей – кода и данных. При этом в области кода должны быть выделены участки, соответствующие отдельным статическим переменным. Область кода должна быть дизассемблирована, то есть представлена в виде последовательности ассемблерных команд (ассемблерного листинга) и размечена на функции. В простейшем случае, каждая функция представляет собой непрерывный участок кода и функции следуют одна за другой непрерывно. Для каждой функции должны быть восстановлены её параметры, т.е. выделены участки стека и регистры, через которые они передаются. Также требуется выделить участок стека, содержащий локальные (автоматические) переменные функции и разбить его на области, соответствующие отдельным переменным. Также в случае, если функция возвращает некоторое значение, должны быть определены области памяти (или регистры), в которых будут содержаться возвращаемое значение функции и адрес возврата. Кроме того, требуется представить программу в виде графа вызовов и для каждой функции построить её граф потока управления.

При статическом анализе выделение областей кода и данных, а также таблиц импорта и экспорта, выполняется в процессе разбора исполняемого файла – данные о смещениях и размерах этих областей содержатся в заголовке. Для того чтобы выделить в области данных границы отдельных переменных требуется проанализировать команды чтения и записи к этим данным из области кода. Аналогично, для того, чтобы выделить границы функций, требуется определить их начало, т.е. адрес, на которой управление передаётся с помощью специальной инструкции вывода, и конец – адрес (точнее максимальный адрес, в случае если их несколько) специальной команды возврата.

Современные дизассемблеры используют алгоритм рекурсивного спуска, который позволяет осуществить для каждой функции анализируемой программы анализ потока управления и построить граф потока управления, вычисляя адреса переходов. Алгоритм рекурсивного спуска позволяет автоматически определять области кода, соответствующие отдельным функциям. В современных дизассемблерах автоматически распознаются библиотечные функции, помещенные в область кода, и определяются параметры их вызовов.

Особенности машинных инструкций многих архитектур позволяют выполнять переходы и вызовы по адресам, значения которых определяются во время выполнения: в дизассемблере вычисляется не сам адрес перехода, а адрес памяти, по которому находится адрес перехода. Это препятствует работе

алгоритма рекурсивного спуска и приводит к неполноте графа вызовов и графа потока управления. В современных дизассемблерах для преодоления этого ограничения используется интерактивный отладчик, позволяющий вручную уточнить граф потока управления, выяснив вычисляемые адреса переходов.

Однако статический анализ трудно применять к программам, снабженным средствами защиты от анализа. К таким программам относится, в частности, вредоносный код, а также дорогостоящее лицензионное ПО. В первом случае целью защиты является создание препятствий для разработки средств противодействия вредоносному коду, во втором – препятствий незаконному копированию. В таких программах целенаправленно эксплуатируются конструкции, затрудняющие анализ. В частности, известны и активно применяются на практике программные системы, снабжающие исполняемый код программ «навесной» защитой. «Навесная» защита не позволяет восстанавливать ассемблерный листинг программы, нарушает работу программного отладчика. Методы запутывания («обфускации») кода тоже значительно усложняют анализ кода: даже при наличии интерактивного отладчика анализ запутанного кода занимает неприемлемо много времени из-за большого количества ручных операций. Еще одним из распространённых типов защиты кода от анализа является динамический дешифратор – код программы в исполняемом файле находится в зашифрованном виде и расшифровывается в процессе выполнения. В этом случае пытаются снять дамп в тот момент, когда весь код будет уже расшифрован, однако, в случае «ленивой» расшифровки по требованию (то есть выполняемой, только в момент передачи управления на соответствующий код) такого момента может в принципе не существовать, если на текущих входных данных исполняется не весь код программы. Наконец, во вредоносном коде часто применяется механизм защиты, состоящий в использовании самомодифицирующегося кода. В этом случае, по одним и тем же адресам в разные моменты выполнения может находиться разный код, что приводит к невозможности построения однозначного листинга. Одним из вариантов использования подобного механизма является процесс внедрения вредоносного кода с помощью так называемых «зацепок». В этом случае часть кода одной из системных функций перезаписывается переходом на адрес вредоносного кода. Перезапись происходит в процессе выполнения, а адрес, по которому осуществляется перезапись, часто определяется динамически или по таблице, поэтому с помощью статического анализа проанализировать процесс внедрения вредоносного кода практически невозможно.

Рассмотренные случаи указывают на то, что чистый статический подход зачастую неприменим при анализе бинарного кода и необходимо использование методов динамического анализа. Однако динамический анализ без статического также не позволяет полностью решить задачу, так как не обеспечивает полноты покрытия кода программы при её выполнении на конкретном наборе входных данных. Таким образом, наиболее эффективным видится подход на основе комбинации динамического и статического анализа, при котором статическое представление дополняется данными, полученными в ходе динамического анализа.

Поскольку при анализе бинарного кода необходимо комбинировать методы статического и динамического анализа, причем моменты перехода от одного анализа к другому, как правило, не могут быть определены автоматически, необходимо обеспечить возможность интерактивного взаимодействия системы анализа с пользователем (аналитиком), управляющим системой на основании априорных знаний об анализируемой программе. Изучение всей программы, даже с учётом её разбиения на функции может быть весьма трудоёмкой задачей, поэтому требуется, чтобы аналитик тем или иным образом выделял наиболее важные из этих функций, с которых следует начать анализ.

В разделе 5.1 рассматривается оригинальный подход к динамическому анализу бинарного кода, реализованный в интегрированной среде статического и динамического анализа. Генерация трассы осуществляется с помощью одного из средств трассировки: существуют как аппаратные, так и программные решения. Трасса записывается как последовательность машинно-независимых инструкций в промежуточном представлении (раздел 5.2). Во время трассировки выполняется сценарий работы программы, выбранный аналитиком так, чтобы обеспечить выполнение исследуемого им алгоритма. В трассе фиксируется последовательность выполнявшихся на процессоре инструкций, аппаратные прерывания, пользовательский ввод и приходящие извне сетевые пакеты. Затем производится первичное структурирование трассы: выделяется код, относящийся к различным процессам и потокам.

После того, как трасса получена, требуется поднять уровень её представления с последовательности инструкций до последовательности вызовов функций и базовых блоков. Это, в частности, позволит получить информацию, необходимую для дополнения статического представления программы: адреса инструкций, содержащих код, что позволит увеличить долю дизассемблированной части программы; вычисляемые адреса инструкций передачи управления, что позволит уточнить графы потока управления отдельных функций; вычисляемые адреса вызовов функций, что позволит уточнить граф вызовов программы.

В процессе анализа программы возникают следующие подзадачи: поиск реализации интересующего аналитика алгоритма для анализа; выделение подтрассы найденного алгоритма; восстановление входных и выходных данных алгоритма; восстановление интерфейсов между отдельными частями алгоритма; восстановление потока данных рассматриваемой функции.

В интегрированной среде реализована подсистема (**раздел 5.5**), обеспечивающая получение символьной информации о трассе. После того, как вызовы интересующих функций найдены, требуется выделить из них те, которые относятся к работе интересующего алгоритма. Часто выделение выполняется на основе значений параметров функций. Так как функция библиотечная, то параметры и способ их передачи известны и могут быть описаны аналитиком с использованием реализованной подсистемы описания функций. После того, как функция описана, выполняется проход по всем её вызовам в трассе и восстанавливаются значения параметров с помощью реализованного алгоритма восстановления буфера. Результаты выводятся аналитику в виде списка

комментариев к вызовам функций, содержащих прототип описанной функции и значения параметров.

После выделения функции, реализующей некоторую функциональность, требуется понять алгоритм её работы. Наличия её блок-схемы на основе графа потока управления для этого недостаточно – требуется получить входные и выходные параметры функции и восстановить их тип и размер. В исходном коде на языке высокого уровня описания параметров и их типов содержатся в объявлении функции в явном виде. В бинарном коде объявления отсутствуют, а вместо имён переменных в коде используются ячейки памяти и регистры.

В условиях стандартной модели компиляции параметры могут быть восстановлены, как в статическом, так и в динамическом подходе. В случае если высокоуровневый параметр имеет базовый тип (например, символьный или целочисленный), то в бинарном коде ему будет соответствовать ячейка памяти или регистр. Однако в случае составных типов, таких как массив и структура, передаваемых обычно через указатель в статическом подходе, также как и при работе с исходным кодом возникает проблема алиасинга, так как для доступа к одной и той же области памяти могут использоваться различные ячейки, содержащие указатель. При использовании динамического подхода известны фактические адреса всех обращений к памяти, что позволяет обойти эту проблему. Таким образом, для определения границ области, параметров передаваемых по указателю и структуры этой области, эффективнее применять структурный анализ построенного по трассе графа потока. Описываемая система содержит реализацию алгоритма восстановления формата данных, который рассматривается **в разделе 5.4**.

Для понимания работы анализируемого алгоритма требуется не только знание параметров и их структуры, но и связи (зависимости) по данным между отдельными входными и выходными параметрами, а также последовательности инструкций, реализующие эти связи, то есть, фактически, процесс преобразования входных параметров в выходные. Для получения этих зависимостей и инструкций используется прямой и обратный слайсинг, который может применяться как в статическом (статический слайсинг), так и в динамическом (динамический слайсинг) подходах. Статический слайс будет содержать большее количество инструкций, так как должен соответствовать работе алгоритма на всех входных данных. Он может быть избыточен, в частности, в связи с неточностью анализа работы с указателями (алиасинг). Динамический слайс соответствует работе анализируемого алгоритма только на конкретных входных данных (заданных при снятии трассы) и может быть неполон, в частности из-за неполноты покрытия кода алгоритма при снятии трассы. Комбинация статического и динамического слайсинга позволяет достичь баланса между точностью и объёмом, что необходимо для правильного понимания анализируемого алгоритма.

Проверка корректности выявленного алгоритма производится на контрольном примере – ассемблерной программе, построенной по выделенной с помощью слайсинга части программы. В случае динамического слайса контрольный пример должен повторить работу исследуемого алгоритма на

наборе входных данных, используемом при трассировке, выдав тот же результат. Оформленный в виде контрольного примера алгоритм затем может быть исследован на предмет наличия ошибок, НДВ и т.п.

Методика, реализованная в интегрированной среде, была применена для решения ряда практических задач и показала свою высокую эффективность.

В разделе 5.6 рассматриваются примеры практического применения разработанной методики.

В первом примере (раздел 5.6.1) происходит автоматическое выделение кода модельного алгоритма, причем рассматриваются две реализации алгоритма, в виде кода для виртуальной машины и в виде управляемого машинного кода. Каждая полученная трасса состояла примерно из 500 000 шагов. Оба варианта реализации были успешно выделены, для виртуальной машины результат составил 356 инструкций, для управляемого кода – 143.

Второй пример (раздел 5.6.2) заключается в восстановлении алгоритма, работающего с секретными данными: рассматривался алгоритм генерации ключа системы SPLM. Исполняемый файл получает на вход текстовый файл, содержащий идентифицирующие компьютер данные, на выходе – файл с лицензионным ключом. Последовательно были проведены следующие действия: из IDA Pro были импортированы символы стандартных функций, в трассе была выявлена запись лицензионного ключа в файл функцией WriteFile WinApi, среди параметров этой функции был выявлен буфер памяти, содержащий лицензию, с использованием заранее построенного интегрированной средой графа зависимостей для данной трассы был автоматически выделен код, ответственный за построение лицензии. Для облегчения навигации использовалось дерево вызовов функций. Помимо того, в коде был обнаружен некритичный дефект: программа оставляла файловые дескрипторы незакрытыми.

В третьем примере (раздел 5.6.3) требовалось восстановить формат файла, содержащего исполняемый код. Рассматривалась программа-распаковщик, принимающая на вход файл testfile.bin размером 2100 KB и выводящая распакованный код на стандартный вывод. Первоначальный поиск функции ReadFile не дал результат. Дальнейший анализ показал, что было применено отображение файла в память процесса использованием функции ZwMapViewOfSection. Восстановление её параметров позволило выделить область памяти, в которую был отображён файл. Восстановление формата осуществлялось от точки возврата данной функции до конца трассы. По результатам восстановления буфера, содержащего указанный файл, был сделан вывод о том, что фактически читалась незначительная часть файла, причём файл читался непоследовательно, т.е. с разрывами. В результате автоматического анализа формат считанной части файла был восстановлен.

В рамках четвертого примера (раздел 5.6.4) требовалось исследовать поведение вредоносного кода Trojan.Win32.Tdss.ajfl и выделить в виде ассемблерного листинга его код, в момент окончательного разворачивания в памяти зараженной машины. Виртуальная машина AMD SimNow была заражена указанным зловредным кодом путём запуска его исполняемого файла Trojan.Win32.Tdss.ajfl.exe. Данный вредоносный код перехватывает любое

обращение к диску на уровне файловой системы. При снятии трассы была вызвана команда оболочки cmd.exe type для печати на консоль содержимого некоторого файла; выполнение этой команды гарантированно передает управление вредоносному коду. В первую очередь в полученной трассе была найдена функция CreateFileW, для которой было восстановлено значение первого параметра, содержащего имя открываемого файла. После чего был найден вход в ядро ОС, реализованный инструкцией SYSENTER. Затем был найден переход в модуль файловой системы ntfs.sys. Этот переход соответствовал вызову функции IofCallDriver. Первая инструкция этой функции была перезаписана инструкцией JMP, передающей управление вредоносному коду, который затем осуществляет вызов одной из своих функций, используя таблицу переходов. В результате был получен один из диапазонов адресов, по которым был загружен вредоносный код. По данному диапазону была проведена фильтрация трассы и построен ассемблерный листинг, который был сохранён в файл в текстовом виде.

В шестой главе рассматривается разработанный в диссертации метод двухэтапной компиляции, обеспечивающий переносимость программ, написанных на языках общего назначения Си/Си++, с использованием динамической компиляции. Метод двухэтапной компиляции на базе LLVM реализован для Linux-платформ на базе архитектур x86 и ARM. Также в главе рассматривается разработанное в диссертации «облачное хранилище» приложений нового поколения, обеспечивающее с одной стороны переносимость программ в рамках семейства ARM, а с другой стороны, высокую степень надежности и безопасности приложений, доступных в рамках хранилища, за счет использования среды *Svace* и интегрированной среды статического и динамического анализа бинарного кода.

В разделе 6.1 обсуждается метод двухэтапной компиляции, обеспечивающий переносимость программ, написанных на языках общего назначения Си/Си++, между вариантами процессорных архитектур одного семейства (для переносимости программ между различными архитектурами необходимо накладывать дополнительные ограничения, например, на использование ассемблерных вставок). Двухэтапная компиляция позволяет существенно сократить накладные расходы, связанные с учетом особенностей конкретных платформ, тем самым решая фундаментальную проблему развертывания и поддержки приложений.

На первом этапе приложение компилируется на машинах разработчиков специальным набором компиляторных инструментов на базе LLVM, при этом выполняются лишь машинно-независимые оптимизации. Результат компиляции сохраняется в биткод-файлах LLVM, дополнительно автоматически генерируется информация об устройстве программного пакета и о схеме его инсталляции. На втором этапе программа оптимизируется на машине пользователя с учетом его поведения и особенностей его вычислительной системы. Поддерживается несколько режимов работы: а) автоматическая кодогенерация бинарной программы, оптимизированной под конкретную архитектуру, и инсталляция программы с помощью сохраненной на первом этапе информации; б)

динамическая оптимизация программы во время её работы с учетом собранного профиля пользователя с помощью реализованной инфраструктуры сбора профиля на основе пакета Oprofile и JIT-оптимизации на основе JIT-компилятора LLVM; в) оптимизация программы с учетом профиля пользователя во время простоя системы для экономии ресурсов.

При экспериментальной проверке корректности работы созданной системы двухпроходной компиляции на системах x86 и ARM дополнительно была проведена оптимизация компонент LLVM для их более быстрой работы и потребления меньшего объема памяти. При кодогенерации на платформе ARM было достигнуто сокращение использования памяти на 1.6-10.9% и времени компиляции на 10-20%.

В разделе 6.2 обсуждается применяемая на втором этапе двухпроходной компиляции система динамической компиляции с учетом данных динамического профиля. Система сборки профиля реализована на основе профилировщика Oprofile и использует т.н. *выборочное* профилирование: системный компонент (модуль ядра Linux) собирает статистику о ходе выполнения программы с помощью аппаратных счетчиков на машине пользователя и сохраняет ее в буфер, а демон-профилировщик считывает данные из буфера, распознает и записывает их в FIFO-каналы. Окончательное формирование базы данных профиля осуществляется библиотекой, которая по сигналу от демона получает обработанные данные профиля из FIFO-каналов. Эта данные могут быть сразу переданы динамическому компилятору или сохранены для дальнейшей оптимизации во время простоя системы.

Для корректного использования собранных данных во время компиляции необходима подстройка полученного динамического профиля до точного профиля по базовым блокам и ребрам графа потока управления программы. Профиль неизбежно содержит погрешности из-за накладных расходов самого профилировщика и из-за того что необходимо сохранять лишь некоторую выборку значений аппаратных счетчиков для ускорения профилирования. Следовательно, такие профили не могут быть непосредственно использованы оптимизационными фазами компилятора (LLVM) так же, как и данные статического профиля – например, построенный по ребрам графа потока управления приблизительный профиль не будет удовлетворять уравнениям потока. Для решения этой задачи используется модифицированный алгоритм сглаживания динамического профиля для удовлетворения уравнениям потока, чтобы его можно было непосредственно использовать теми же оптимизациями, что и ранее поддерживали статический профиль программы.

Разработанная система динамической компиляции с учетом данных профиля пользователя была протестирована как на корректность работы, так и на производительность. К сожалению, существующие оптимизации в LLVM мало используют данные статического профиля и поэтому должны быть дополнительно доработаны. В работе использовалась существующая в LLVM оптимизация перемещения базовых блоков по данным динамического профиля, применение которой для программы SQLite привело к ускорению работы программы на тестовом наборе данных на ~3%. Разработанный алгоритм

построения суперблоков и настройки оптимизаций LLVM с учетом сформированных суперблоков протестирован на данных статического профиля и будет использоваться с динамическим профилем в ходе дальнейших работ.

В разделе 6.3 предлагается метод организации «облачного хранилища» приложений нового поколения, обеспечивающего как переносимость программ в рамках одного семейства процессорных архитектур ARM, так и высокую степень надежности и безопасности хранимых приложений. Для построения такого облачного хранилища необходимо использование предложенной схемы двухэтапной компиляции для получения представления программы в виде бит-код файлов LLVM, а также метаинформации об устройстве приложения и о схеме его установки.

После помещения приложения в хранилище для обеспечения безопасности приложение может быть автоматически проверено инструментами среды *Svace* на наличие критических ошибок и уязвимостей и в случае необходимости может быть проведен более глубокий анализ методами, использованными в среде анализа бинарного кода. Таким образом, обеспечивается необходимая степень продуктивности и безопасности приложений.

Данный подход может быть использован для приложений как с открытым, так и с закрытым исходным кодом без каких-либо опасений со стороны разработчиков, так как восстановление исходного кода приложения по биткод-файлам LLVM является затруднительным. Отметим, что такой подход к развертыванию программ широко применяется в таких средах, как Java.

В заключении формулируются основные результаты и выводы диссертационной работы.

ОСНОВНЫЕ РЕЗУЛЬТАТЫ РАБОТЫ

В результате проведения теоретических и практических исследований по тематике диссертации получены следующие результаты:

1. Исследован и разработан новый метод планирования команд и конвейеризации циклов, учитывающий особенности функционирования современных процессоров (ARM, EPIC), с поддержкой условного и спекулятивного выполнения.
2. Исследованы и разработаны методы и поддерживающие их программные инструменты разработки параллельных программ на языке *Java* с явными обращениями к коммуникационной библиотеке, позволяющие проводить итеративную разработку с минимальным использованием целевой аппаратуры, для повышения продуктивности разработки программ для систем с распределенной памятью.
3. Исследованы и разработаны масштабируемые методы статического анализа исходного кода программ, обеспечивающие нахождение уязвимостей и других дефектов, для эффективного аудита, как исходного кода на языках C/C++, так и бит-кода LLVM, содержащего десятки миллионов строк, в приемлемое время с высокой степенью достоверности.

4. Исследованы и разработаны комбинированные методы статического и динамического анализа бинарного кода, базирующиеся на сборе и анализе трасс его выполнения, с целью восстановления алгоритмов и нахождения недокументированных возможностей в защищенном бинарном коде.
5. Исследован и разработан метод двухэтапной компиляции, обеспечивающий переносимость программ, написанных на языках общего назначения C/C++, с использованием динамической компиляции.

Кроме того, разработанные программные средства используются в учебном процессе факультетов ВМК МГУ и ФУПМ МФТИ. Содержание диссертационной работы легло в основу учебных курсов по параллельному программированию, машинно-ориентированной компиляции, анализу программ на факультетах ВМК МГУ, ФУПМ МФТИ и МИФИ.

ПУБЛИКАЦИИ ПО ТЕМЕ ДИССЕРТАЦИИ

Статьи в журналах рекомендованных ВАК РФ

1. А.И. Аветисян, С.С. Гайсарян, О.И. Самоваров. “Возможности оптимального выполнения параллельных программ, содержащих простые и итерированные циклы на неоднородных параллельных вычислительных системах с распределенной памятью”. Программирование, 2002, № 1, с. 38-54.
2. В.П. Иванников, С.С. Гайсарян, А.И. Аветисян, В.А. Падарян. «Оценка динамических характеристик параллельной программы на модели.» // Программирование, №4, 2006.
3. В.П. Иванников, А.И. Аветисян, С.С. Гайсарян, В.А. Падарян. «Прогнозирование производительности MPI-программ на основе моделей.» // Журнал «Автоматика и телемеханика», 2007, N5, с. 8-17.
4. А.И. Аветисян, В.В. Бабкова и А.Ю. Губарь. «Возникновение торнадо: трехмерная численная модель в мезомасштабной теории турбулентности по Николаевскому»// «ДАН/Геофизика», том 419, №4, с. 547-552.
5. Аветисян А.И., Бабкова В., Гайсарян С.С., Губарь А.Ю. «Рождение торнадо в теории мезомасштабной турбулентности по Николаевскому. Трехмерная численная модель в ParJava.» // Журнал «Математическое моделирование». 2008, т. 20, № 8, с. 28-40.
6. В.П. Иванников, Аветисян А.И., Гайсарян С.С., Акопян М.С. “Особенности реализации интерпретатора модели параллельных программ в среде ParJava.” Журнал «Программирование». 2009 т. 35, №1 с. 10-25
7. А.И. Аветисян, С.С. Гайсарян, В.В. Бабкова. Итеративная разработка параллельных программ в среде ParJava //Программирование, №4, 2009, с. 56-72.
8. А.Ю.Тихонов, А.И. Аветисян. Развитие taint-анализа для решения задачи поиска некоторых типов закладок . Труды ИСП РАН, т. 20, 2011 г. с. 9-24.
9. Арутюн Аветисян, Андрей Белеванцев, Алексей Бородин, Владимир Несов. Использование статического анализа для поиска уязвимостей и

критических ошибок в исходном коде программ. Труды ИСП РАН том 21, 2011, стр. 23-38.
10. Арутюн Аветисян, Алексей Бородин. “Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок”. Труды ИСП РАН том 21, 2011, стр. 39-54.
11. А.И. Аветисян, К.Ю. Долгорукова, Ш.Ф. Курмангалеев. Динамическое профилирование программы для системы LLVM. Труды ИСП РАН том 21, 2011, стр. 71-82.
12. А.И. Аветисян, М.С. Акопян, С.С. Гайсарян. Методы точного измерения времени выполнения гнезд циклов при анализе JavaMPI-программ в среде ParJava. Труды ИСП РАН том 21, 2011, стр. 83-102.
13. Дмитрий Мельник, Александр Монаков, Арутюн Аветисян. “Поддержка команд с условным выполнением в селективном планировщике команд”. Труды ИСП РАН том 21, 2011, стр. 103-118.
14. А.И. Аветисян. “Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения”. Труды ИСП РАН том 22, 2012, стр. 1-11.
15. А.И. Аветисян. “Планирование команд и конвейеризация циклов на современных архитектурах”. Труды ИСП РАН том 22, 2012, стр. 12-19.
16. А.И. Аветисян, А.И. Гетьман. “Восстановление структуры бинарных данных по трассам программ”. Труды ИСП РАН том 22, 2012, 78-95.
17. А.Ю.Тихонов, А.И. Аветисян. “Комбинированный (статический и динамический) анализ бинарного кода”. Труды ИСП РАН том 22, 2012, 120-131.

Статьи, учебные пособия и материалы конференций

18. А.И. Аветисян, И.В. Арапов, С.С. Гайсарян, В.А. Падарян. “Среда разработки параллельных Java-программ для однородных и неоднородных сетей JavaVM” Труды Всероссийской научной конференции “Высокопроизводительные вычисления и их приложения”. Изд-во Московского университета, М. 2000.
19. А.И. Аветисян, И.В. Арапов, С.С. Гайсарян, В.А. Падарян. “Параллельное программирование с распределением по данным в среде ParJava.” Вычислительные методы и программирование. Том 2, М. 2001 стр. 70-87.
20. А.И. Аветисян, В.А. Падарян. “Библиотека интерфейсов и классов, расширяющих язык Java средствами разработки параллельных программ в модели SPMD.” Труды Института Системного Программирования РАН. Том 2, М. 2001, стр. 49-64.
21. А.И. Аветисян, И.В. Арапов, С.С. Гайсарян, В.А. Падарян. “Среда ParJava для разработки SPMD-программ для однородных и неоднородных сетей JavaVM.” Труды ИСП РАН. Том 2, М. 2001, стр. 27 – 48.
22. A. Avetisyan, S. Gaissaryan, O. Samovarov. “Extension of Java Environment by Facilities Supporting Development of SPMD Java-programs”. V. Malyshkin

(Ed.): PaCT 2001, LNCS 2127, Springer-Verlag Berlin Heidelberg 2001, p. 175 – 180.
23. V. Ivannikov, S. Gaissaryan, A. Avetisyan, O. Samovarov. “ParJava: IDE Supporting SPMD Java-Programming” Computer Science and Information Technologies (CSIT), Yerevan, Sept. 17 – 20, 2001. Proceedings, p. 92 – 96.
24. В.П. Иванников, С.С. Гайсарян, А.И. Аветисян. “Среда ParJava: разработка масштабируемых параллельных SPMD-программ в окружении Java.” Труды Международной конференции “Параллельные вычисления и задачи управления” Москва, 2-4 октября, 2001.
25. Victor Ivannikov, Serguei Gaissaryan, Arutyun Avetisyan, Vartan Padaryan. Development of Scalable Parallel Programs in ParJava Environment. // Parallel CFD 2003, pp. 291 – 293
26. V. Ivannikov, S. Gaissaryan, A. Avetisyan, V. Padaryan. “Analyzing dynamic properties of parallel program in ParJava Environment” Computer Science and Information Technologies (CSIT), Yerevan, September 22 – 26, 2003. Proceedings, p. 19 – 23.
27. Victor Ivannikov, Serguei Gaissaryan, Arutyun Avetisyan, Vartan Padaryan. Improving properties of a parallel program in ParJava Environment // The 10th EuroPVM/MPI conference, Venice, Sept. 2003, LNCS v. 2840, 491-494.
28. Сергей Гайсарян, Арутюн Аветисян, Вартан Падарян, Катерина Долгова. Среда разработки параллельных Java-программ, использующих библиотеку MPI. // Труды всероссийской научной конференции «Научный сервис в сети Интернет». Новороссийск, 20-25 сентября 2004, с. 177-179.
29. Victor Ivannikov, Serguei Gaissaryan, Arutyun Avetisyan, Vartan Padaryan. “Development of Scalable Parallel Programs in ParJava Environment”. B. Chetverushkin, A. Ecer, et al (eds.) // Parallel Computational Fluid Dynamics, 2004 Elsevier B.V., pp. 417 – 423.
30. Victor P. Ivannikov, Serguei S. Gaissaryan, Arutyun I. Avetisyan, Vartan A. Padaryan. Checkpointing improvement in ParJava environment. // Parallel CFD 2004, May, 24-27, 2004, Gran-Canaria.
31. Victor Ivannikov, Serguei Gaissaryan, Arutyun Avetisyan, Vartan Padaryan and Hennadiy Leontyev. “Dynamic Analysis and Trace Simulation for Data Parallel Programs in the ParJava Environment”. M. Estrada, A. Gelbukh (Eds.) // Avances en la Ciencia de la Computacion, ENC’04, Colima, Mexico, pp. 481-488.
32. Сергей Гайсарян, Арутюн Аветисян, Вартан Падарян, Геннадий Леонтьев. Применение среды ParJava для разработки параллельных программ. // Международная научная конференция «Суперкомпьютерные системы и их применение». 26-28 октября 2004, Минск, с. 99-103.
33. В.П. Иванников, С.С. Гайсарян, А.И. Аветисян, В.В. Бабкова, В.А. Падарян "Разработка параллельных Java программ для высокопроизводительных вычислительных систем с распределенной памятью", Труды ИСП РАН, т. 5, 2004, стр. 41-62.

34. С.С. Гайсарян, А.И. Аветисян, К.Н. Долгова, В.А. Падарян. “Средства восстановления программы после сбоев среды ParJava.”// Тр. Всероссийской научной конф. «Научный сервис в сети Интернет: технологии распределенных вычислений». Новороссийск, 19-24 сентября 2005, стр.
35. V. Ivannikov, S. Gaissaryan, A. Avetisyan, B. Babkova. “Using instrumental computer for SPMD-program analysis and tuning” Computer Science and Information Technologies (CSIT), Yerevan, September 19 – 23, 2005. Proceedings, p. 385 – 388.
36. А.И. Аветисян, С.С. Гайсарян, В.А. Падарян. Принципы реализации модели параллельной программы в среде ParJava. // Труды Всероссийской научной конф. «Научный сервис в сети ИНТЕРНЕТ: технологии параллельного программирования», г. Новороссийск, 18-23 сентября 2006. стр. 106-107.
37. А.Ю. Губарь, А.И. Аветисян, В.В. Бабкова. Численное моделирование возникновения 3D торнадо в теории мезомасштабных вихрей по Николаевскому. // 13th International Conference On The Methods Of Aerophysical Research. 5 – 10 February, 2007, Ed. V.M. Fomin. – Novosibirsk: Publ. House “Parallel” – 250 p; P135-140
38. Аветисян А. И., Бабкова В. В., Губарь А. Ю. Моделирование интенсивных атмосферных вихрей в среде ParJava. // Труды Всероссийской научной конф. «Научный сервис в сети ИНТЕРНЕТ: технологии параллельного программирования», г. Новороссийск, 18-23 сентября 2006. стр. 109-112.
39. Victor Ivannikov, Serguei Gayssaryan, Arutyun Avetisyan, Vartan Padaryan “Model Based Performance Evaluation for MPI Programs” //MTPP 2007 Parallel Computing Technologies First Russian-Taiwan Symposium Pereslavl-Zalesskii (Russia), September 2-3, 2007.
40. A.I. Avetisyan, V.V. Babkova, S.S. Gaissaryan, A.Yu. Gubar “Intensive Atmospheric Vortices Modeling Using High Performance Cluster Systems”// PaCT-2007, LNCS, v. 4671/2007, p. 487-495.
41. A.Yu. Gubar, A.I. Avetisyan, and V.V. Babkova, Numerical modelling of 3D tornado arising in the mesovortice turbulence theory of Nikolaevskiy /International Conf. on the Methods of Aerophysical Research: Proc. Pt III /Ed. V.M. Fomin. – Novosibirsk: Publ. House “Parallel”, 2007. – 250 p; P135-140.
42. Victor P. Ivannikov, Arutyun I. Avetisyan, Varvara V. Babkova, Alexander Yu. Gubar “Tornado arising modeling using high performance cluster systems” // Sixth International Conference on Computer Science and Information Technologies (CSIT’2007), 24-28 September, Yerevan, Armenia.
43. Andrey Belevantsev, Dmitry Melnik, and Arutyun Avetisyan. Improving a selective scheduling approach for GCC. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007. http://sysrun.haifa.il.ibm.com/hrl/greps2007/
44. Arutyun Avetisyan, Andrey Belevantsev, and Dmitry Melnik. GCC instruction scheduler and software pipelining on the Itanium platform. 7th Workshop on

Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-7). Boston, MA, USA, April 2008. http://rogue.colorado.edu/EPIC7/avetisyan.pdf
45. Аветисян А.И., Бабкова В., Гайсарян С.С. High productive programming for cluster systems using Java with explicit call of MPI //Седьмая международная конференция памяти академика А.П. Ершова Перспективы систем информатики. Рабочий семинар "Наукоемкое программное обеспечение", Новосибирск 15-19 июня 2009 г. ООО "Сибирское Научное Издательство", Нс. 2009, с. 1-6.
46. А.И. Аветисян, В.В. Бабкова, А.В. Монаков. Обеспечение высокопродуктивного программирования для современных параллельных платформ Труды ИСП РАН, том 16, 2009 г., стр 9-30.
47. А.И. Аветисян, В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Автоматизация динамического анализа бинарного кода. Материалы международной конференции «РусКрипто’2009».
48. Alexander Monakov and Arutyun Avetisyan. “Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs”, SAMOS 2009 Workshop, LNCS Proceedings, 2009, №5657, pp 288-296
49. О.И. Самоваров, А.И. Аветисян, А.В. Николаев, А.О. Гетьман. «Технолгия использования вычислительной кластерной системы МФТИ-60 для численного модеирования». Москва, МФТИ, 2009. – 73 с.
50. А.И. Аветисян, В.А. Падарян, А.И. Гетьман, М.А. Соловьев. О некоторых методах повышения уровня представления при анализе защищенного бинарного кода. // Материалы XIX Общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». Санкт-Петербург, 05-10 июля 2010 г. Стр. 97-98.
51. Avetisyan A.I., Campbell R., Gupta I., Heath M.T., Ko S.Y., Ganger G.R., Kozuch M.A., O'Hallaron, D., Kunze M.; Kwan T.T., Lai K., Lyons M., Milojicic D.S., Hing Yan Lee, Yeng Chai Soh, Ng Kwang Ming, Luke, J-Y., Han Namgoong. Open Cirrus: A Global Cloud Computing Testbed. / «Computer». Volume 43, Issue:4. – April 2010. pp. 35 – 43.