Для цитирования: Русяев Р.М., Баранников С.В., Нейман-заде М.И. Особенности реализации механизма обработки исключений С++ для платформы «Эльбрус» // Вопросы радиоэлектроники. 2018. № 2. С. 45–50.
УДК 004.4’422

Р.М. Русяев[1, 2], С.В. Баранников[1], М.И. Нейман-заде[1, 2, 3]
[1] АО «МЦСТ», [2] ПАО «ИНЭУМ им. И.С. Брука», [3] МФТИ (ГУ)

ОСОБЕННОСТИ РЕАЛИЗАЦИИ МЕХАНИЗМА ОБРАБОТКИ ИСКЛЮЧЕНИЙ С++ ДЛЯ ПЛАТФОРМЫ «ЭЛЬБРУС»

Механизм исключений является важной частью крупных приложений, написанных на языках высокого уровня, таких как С++, поэтому имеется острая необходимость в их эффективной реализации. В данной работе рассматривается подход к обработке исключений, называемый zero cost exceptions handling, позволяющий минимизировать накладные расходы на обработку исключений в пользовательском приложении. Это достигается благодаря локализации вспомогательного кода, требуемого для обработки исключений, в специальные участки, называемые landing pad. Дано подробное описание реализации данного механизма, показывающее, за счет чего происходит минимизация накладных расходов. Дается краткий обзор альтернативного подхода к обработке исключений (механизм setjmp/longjmp), используемого в предыдущей версии компилятора и основанного на использовании функций из стандартной библиотеки языка С – setjmp, longjmp. Описаны нововведения и доработки со стороны компилятора и оптимизирующих фаз, необходимые для реализации рассматриваемого механизма. Приведены результаты сравнения производительности, подтверждающие преимущество механизма zero cost exceptions handling по сравнению с механизмом setjmp/longjmp.

Ключевые слова: *архитектура «Эльбрус», исключения С++, обработка исключений, zero cost exceptions handling.*

Введение

Роль исключений в широко используемых языках программирования, таких как С++, весьма существенна, но для их реализации требуются дополнительные накладные расходы, которые влекут за собой увеличение времени выполнения исполняемого приложения. Для их сокращения был разработан механизм zero cost exceptions handling (0eh), который позволяет локализовать весь код обработки исключений так, что он не перемешивается с основным кодом исполняемой программы. Это заметно повышает производительность по сравнению с используемым в предыдущих версиях компилятора механизмом setjmp/longjmp (sjlj), который в процессе исполнения программы поддерживает стек исключений, инструментируя основной код исполняемого приложения.

Краткое описание механизма setjmp/longjmp

Механизм sjlj основан на вызовах функций setjmp и longjmp из стандартной библиотеки языка С. Функция setjmp сохраняет контекст исполняемого приложения, затем вызовом функции longjmp осуществляется нелокальная передача управления в точку, где был произведен вызов соответствующей функции setjmp. При использовании данного подхода try-блок заменяется на вызов функции setjmp, а выражение языка throw – на вызов функции longjmp. Эта техника имеет существенные недостатки с точки зрения производительности приложения [1]:

- В начале каждого try-блока должна быть вызвана функция setjmp; кроме того, должен быть поддержан список jmp_buf – структура, в которую setjmp сохраняет контекст приложения.
- Во время исполнения программы необходимо поддерживать в актуальном состоянии список локальных объектов, предназначенных для уничтожения.

Далее в статье приводится описание процессов, необходимых для поддержки исключений, использующих механизм 0eh, который позволяет устранить эти недостатки.

Механизм zero cost exceptions handling

Принципиальное описание

Впервые этот механизм был реализован для процессоров с архитектурой IA-64; его ABI (Application Binary Interface) приведен в [2]. Необходимость в создании 0eh была обусловлена тем, что на поддержку исключений в коде исполняемого приложения требуются дополнительные накладные расходы, заключающиеся в инструментировании этого кода для возможности обработки исключения. Основная идея заключается в минимизации накладных расходов в том случае, если исключение не было

«брошено» из функции. Реализация этого процесса (throw exception) заключается в создании объекта исключения и инициации раскрутки стека, при которой происходит поиск обработчика, соответствующего данному объекту исключения. С точки зрения языка C++ это выполнение инструкции throw, а обработка исключения – выполнение кода в catch-части, соответствующей конструкции try-catch. В связи с тем, что исключения бросаются значительно реже в сравнении с кодом, построенным для его обработки, был разработан механизм 0eh, который позволяет снизить указанные накладные расходы практически до нуля. В настоящее время он реализован во многих современных компиляторах, таких как gcc и clang [3].

Все действия по обработке исключений локализованы в участке кода, называемом landing pad, управление на который передается только в случае, если исключение было брошено. Соответственно, основной код исполняемой программы не будет перемешан с кодом обработки исключений, что обеспечивает минимальные накладные расходы.

При компиляции приложения инструкция throw раскрывается в два вызова функций из библиотеки поддержки libsupc++ (которая входит в состав стандартной библиотеки C++):

- __cxa_allocate_exception – выделяет память под объект выбрасываемого исключения;
- __cxa_throw – инициирует процесс раскрутки стека, который более подробно будет рассмотрен позже.

Код, выполняющий действия в случае, если исключение было брошено, – landing pad, – должен быть построен в следующих ситуациях:

- Исключение бросается из try-блока. В этом случае будет построен код landing pad, соответствующий catch-блоку языковой конструкции, управление на который передается по типу брошенного исключения. Если тип брошенного исключения не соответствует ни одному catch-блоку, то продолжается раскрутка стека вызовом _Unwind_Resume. Построенный в результате код landing pad называется обработчиком исключения.
- Функция имела локальные объекты, для которых должны быть вызваны деструкторы этих объектов. В этом случае код landing pad, называемый cleanup, будет содержать вызовы соответствующих деструкторов, после чего будет продолжена раскрутка стека.
- Функция имеет спецификацию, определяющую, исключения каких типов могут быть брошены из нее. В этом случае будет построен код landing pad, проверяющий соответствие типа выбрасываемого исключения спецификации функции. Landing pad такого типа также называется обработчиком исключений.

Компилятор подготавливает необходимую информацию, используемую в процессе раскрутки стека, посредством создания cfi (call frame information) – директив в ассемблерном файле. Ассемблер раскрывает директивы согласно бинарному формату DWARF, а также генерирует специальную информацию, называемую LSDA (Language Specific Data Area), для каждого landing pad содержащую данные по исключениям, которые этот landing pad может обрабатывать [4].

Процесс раскрутки стека инициирует библиотечная функция _Unwind_RaiseException, располагающаяся в компиляторной библиотеке поддержки libgcc, которую вызывает __cxa_throw при бросании исключения. В свою очередь, функция _Unwind_RaiseException читает информацию из секции исполняемого файла, в которой находятся необходимые данные, являющиеся результатом раскрытия cfi-директив. Также в этой секции располагаются сведения о специальной функции, называемой personality routine, которая подготавливает необходимую информацию, используемую в landing pad на основе данных из LSDA.

Процесс бросания исключения

Процесс бросания исключения начинается с создания объекта исключения, который участвует в раскрутке стека, пока не будет пойман в одном из обработчиков исключения. В общем случае объект исключения создается последовательными вызовами функции __cxa_allocate_exception, которая выделяет для него память, и конструктора для создания объекта в выделенной области. После того как объект исключения сгенерирован, он передается по косвенности в функцию __cxa_throw вместе с указателем на деструктор этого объекта. Функция __cxa_throw инициирует процесс раскрутки стека вызовов посредством вызова _Unwind_RaiseException.

Прежде чем приступить к описанию процесса раскрутки стека, рассмотрим организацию стека на архитектуре «Эльбрус».

Процесс раскрутки стека

Организация стековой памяти архитектуры «Эльбрус»

В отличие от традиционной модели, используемой в популярных архитектурах (таких как x86, arm, x86–64), где стек организован в виде непрерывного участка памяти, который доступен для записи в непривилегированном режиме и помимо пользовательских данных хранит адреса возврата вызывающих процедур, в эльбрусовской реализации стек организован как три непересекающиеся области памяти:

- Пользовательский стек, в котором располагаются локальные данные пользовательской программы, такие как агрегатные объекты, массивы и прочее.
- Процедурный стек, часть которого расположена в регистровом файле, а часть – в оперативной памяти. Данный стек предназначен для хранения локальных данных процедуры и содержит скалярные переменные, а также агрегатные объекты небольших размеров. Каждая процедура имеет выделенную для нее часть регистрового файла – регистровое окно, в случае нехватки которого при вызове очередной функции происходит автоматическая откачка в соответствующую часть процедурного стека, расположенного в памяти.
- Стек связующей информации, предназначенный для хранения адресов возврата вызывающих процедур, который, как и процедурный стек, частично располагается на регистрах, а частично – в памяти: в случае нехватки регистров при очередном вызове функции происходит аппаратная откачка регистровой части в область, располагающуюся в памяти.

Таким образом, стек, в котором расположены данные приложения вместе с адресами возврата, представлен в виде трех отдельных стеков, причем все данные пользовательского приложения расположены в процедурном и пользовательском стеках, а адреса возврата процедур – в стеке связующей информации, который недоступен для записи в непривилегированном режиме. Такое устройство стека позволяет повысить безопасность приложения в случаях атак переполнения буфера.

Раскрутка стека вызовов

Раскрутка стека – это процесс, который осуществляет вызов деструкторов локальных объектов каждого стекового фрейма, пока не будет найден фрейм с обработчиком исключения, соответствующим объекту брошенного исключения. Процесс инициируется функцией из библиотеки libgcc – _Unwind_RaiseException, выполняемой в два этапа:

1. Для каждого стекового фрейма происходит вызов функции personality routine, которая на основе информации LSDA определяет, существует ли обработчик исключения, соответствующий брошенному исключению, в очередном стековом фрейме, и если так, то функция оповещает об этом функцию _Unwind_RaiseException, возвращая соответствующее значение. При этом если пройдены все стековые фреймы и не был найден ни один обработчик исключения, то _Unwind_RaiseException возвращает управление в функцию __cxa_throw, которая завершает программу вызовом std:: terminate. Следует заметить, что на данном этапе не выполняется никаких действий, за исключением поиска обработчика исключения.
2. Если первый этап выполнен успешно (т.е. для исключения был найден его обработчик), то процесс раскрутки стека повторяется, но теперь для каждого стекового фрейма вызываются деструкторы существующих локальных объектов, которые должны быть уничтожены посредством передачи управления на коды cleanup, построенные компилятором в коде пользовательского приложения. Поэтому, чтобы продолжить раскрутку стека после уничтожения объектов, в коде исполняемой программы строится вызов _Unwind_Resume, который запускает вторую фазу процесса раскрутки стека, продолжающуюся до тех пор, пока не будет встречен обработчик брошенного исключения.

Также следует отметить, что за счет явного разделения стека данных и стека вызовов нет необходимости хранить некоторую дополнительную информацию о стековых кадрах процедур.

Обработка исключения

Действия, выполняемые кодом landing pad, можно классифицировать следующим образом:

- Cleanup – вызов деструкторов для локальных объектов, которые должны быть уничтожены в процессе раскрутки стека.
- Обработчик исключения, который, в свою очередь, делится на несколько типов:
  - catch-обработчик – соответствует catch-части языковой конструкции try-catch;
  - unexpected-обработчик – строится в том случае, если для функции задана спецификация, показывающая, какие типы исключений могут быть брошены из нее. Если тип брошенного исключения не соответствует спецификации, то вызывается std:: unexpected;
  - terminate-обработчик – строится, например, в том случае, если для функции задана спецификация noexcept. Если исключение будет брошено, то программа завершится посредством вызова std:: terminate.

Каждый обработчик исключения начинается с вызова __cxa_begin_catch, который увеличивает счетчик пойманных исключений и помечает текущее исключение как пойманное. После выполнения кода обработчика исключения вызывается функция __cxa_end_catch, уменьшающая счетчик пойманных исключений и уничтожающая объект исключения. После этого процесс обработки исключений считается завершенным.

Адаптация компилятора для поддержки механизма zero cost exceptions handling

Для поддержки описываемого механизма обработки исключений необходимы определенные доработки компилятора, например, адаптация некоторых оптимизаций. В данном разделе дан обзор реализованных дополнений и описаны некоторые аналитические структуры данных, используемые компилятором.

Промежуточное представление компилятора

Процесс компиляции исходной программы в целевой код включает ряд последовательных этапов:

- Работа front-end части компилятора: на этом шаге выполняется лексический, синтаксический и семантический анализы исходной программы, в процессе которых генерируется структура данных, называемая «абстрактным синтаксическим деревом» (АСТ).
- Перевод АСТ во внутреннее представление данных компилятора, называемое промежуточным представлением (intermediate representation, IR).
- Оптимизация программы, представленной в виде промежуточного представления.
- Генерация целевого кода.

IR представляет собой внутренний язык компилятора, состоящий из разного рода операций и их возможных операндов. Существует несколько промежуточных представлений разных уровней абстракции – приближенных к исходному языку (высокоуровневых) или близких к генерируемому для целевой платформы коду (низкоуровневых). Над промежуточным представлением могут строиться различные аналитические структуры данных – например, графы использований-определений (Def-Use Graph), зависимостей и др.

Граф потока управления

Над промежуточным представлением строится структура данных, называемая графом потока управления (Control Flow Graph, CFG), узлами которого являются базовые блоки, называемые также линейными участками. Дуги графа отображают передачу управления между узлами. Базовые блоки содержат группу операций, обладающих следующими свойствами:

- Передача управления может быть произведена только на первую операцию из группы; иными словами, если управление передается на базовый блок, то выполняются все его операции.
- Группа операций, содержащихся в базовом блоке, оканчивается операцией передачи управления на начальную операцию другого базового блока.

Таким образом, все операции промежуточного представления разбиваются на группы, соответствующие узлам графа потока управления, что позволяет прозрачно отображать потоки управления исходной программы.

Операция INVOKE

Вызов функции в промежуточном представлении выполняется операцией CALL, но рассматриваемый подход к обработке исключения предполагает, что в случае, когда функция бросит исключение, управление неявным образом должно быть передано на специальный код по обработке исключения – landing pad. Для отображения потока управления от операции вызова на код обработки исключения вводится операция INVOKE, которая, помимо свойств операции CALL, обладает свойством передачи управления, отсутствующим в семантике операции CALL, и является последней операцией базового блока. На рисунке отображено различие операций CALL и INVOKE на уровне графа потока управления. Таким образом, в случае вызова функции, бросающей исключение, удается явным образом отобразить поток управления на уровне CFG, что, в свою очередь, позволяет безопасно выполнять различного рода оптимизации.

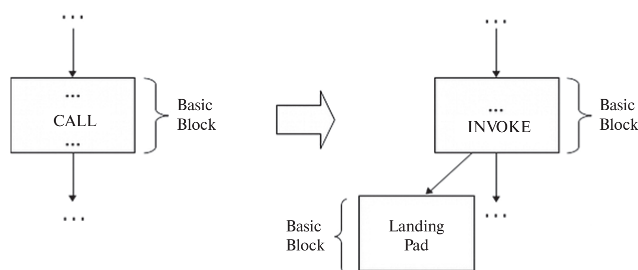

Рисунок. Операции CALL и INVOKE на уровне CFG

Примеры оптимизаций

Преобразование операции INVOKE в CALL

Если при анализе CFG выясняется, что функция никогда не бросает исключение, то операция INVOKE преобразуется в операцию CALL, при этом удаляется соответствующий код обработки исключения. Это значительно упрощает граф потока управления, сокращая количество ветвлений, что положительно влияет на оптимизации.

Inline-подстановка вызова функции через операцию INVOKE

При выполнении оптимизации, выполняющей inline-подстановку функции в точку вызова в том случае, когда функция вызывается через операцию INVOKE, необходим ряд дополнительных преобразований, позволяющих сохранить корректность действий, если будет брошено исключение из функции, вызываемой подставляемой функцией. Для этого

необходимо все вызовы функций без спецификации noexpect или throw(), которые осуществляются через операции CALL, преобразовать в операции INVOKE, перенаправив поток данных от преобразованной операции на landing pad подставляемого вызова. Также нужно удалить все подставленные вызовы _Unwind_Resume, продолжающие процесс раскрутки стека, и перенаправить управления от базовых блоков, в которых они находились, на landing pad подставляемого вызова.

Сравнение показателей производительности механизмов 0eh и sjlj

Как отмечалось, механизм 0eh обеспечивает преимущество в производительности программ по сравнению с режимом sjlj. В таблице приведены результаты замеров производительности по времени исполнения задач C++ из пакетов SPEC CPU2006 и SPEC CPU2017 [5, 6]: отображено отношение времен исполнения задач, скомпилированных с поддержкой режима sjlj, к временам исполнения задач, собранных с поддержкой 0eh. Замеры производились на компьютере «Эльбрус 801» с одним микропроцессором «Эльбрус-Р1», имеющим восемь ядер и тактовую частоту 1,3 ГГц. Задачи были скомпилированы в базовом режиме.

Опции компиляции для задач из пакета SPEC CPU2006:

- для целочисленных задач: -O4 -ffast -static;
- для вещественных задач: -O4 -ffast -ffast-math -static.

Опции компиляции задач из пакета SPEC CPU2006:

- для целочисленных задач: -O3 -ffast -static;
- для вещественных задач: -O4 -ffast -ffast-math -static.

Результаты приведены только для задач, отношение времен исполнения которых изменилось более чем на 1%. В соответствии со значениями среднего геометрического прирост производительности для задач из пакета SPEC CPU2006 составляет 6,4%, а для задач из пакета SPEC CPU2017 – 23,7%, что говорит о существенном преимуществе 0eh в сравнении с режимом sjlj.

Стоит обратить внимание, что значительное ускорение задачи 523.xalancbmk преимущественно обусловлено двумя факторами:

- удалением вызовов setjmp, которые генерировались при использовании модели sjlj и занимали около 30% времени исполняемой программы;
- применением оптимизации inline-подстановки наиболее горячих функций, которые не удавалось подставить при использовании техники sjlj, что в дальнейшем позволило произвести ряд цикловых оптимизаций.

Заключение

В работе был рассмотрен механизм обработки исключений 0eh, позволяющий минимизировать затраты на код обработки исключений в пользовательском приложении. Это осуществляется посредством локализации такого кода в специальный участок – landing pad, что обеспечивает разделение кода обработки исключений с основным кодом исполняемого приложения. Описан принцип функционирования данного механизма, а также приведены результаты замеров 0eh в сравнении с ранее используемой техникой обработки исключений – sjlj, показывающие существенное увеличение производительности приложений, использующих 0eh. Адаптация и развитие оптимизаций для 0eh составляет предмет дальнейших исследований, которые должны привести к более существенному увеличению производительности 0eh в сравнении с sjlj.

Таблица. Отношение времен исполнения задач из пакетов SPEC CPU2006 и SPEC CPU2017, собранных с sjlj и 0eh

Пакет SPEC CPU2006		Пакет SPEC CPU2017	
Название задачи	Отношение времен исполнения	Название задачи	Отношение времен исполнения
–	–	507.cactuBSSN	1,033
447.dealII	1,043	510.parest	1,013
450.soplex	0,983	520.omnetpp	1,066
471.omnetpp	1,040	523.xalancbmk	2,424
483.xalancbmk	1,200	541.leela	1,071
Среднее геометрическое	1,064	Среднее геометрическое	1,237

СПИСОК ЛИТЕРАТУРЫ

1. Christophe de Dinechin. C++ Exception Handling for IA-64. URL: http://static.usenix.org/events/osdi2000/wiess2000/full_papers/dinechin/dinechin_html (accessed 05.11.2017)
2. Itanium C++ ABI: Exception Handling. Available at: http://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html (accessed 05.11.2017)
3. Exception Handling in LLVM. Available at: http://llvm.org/docs/ExceptionHandling.html (accessed 05.11.2017)
4. Exceptions Handling Tables. Available at: https://itanium-cxx-abi.github.io/cxx-abi/exceptions.pdf (accessed 05.11.2017)
5. SPEC CPU2006. Available at: https://www.spec.org/cpu2006 (accessed 05.11.2017)
6. SPEC CPU2017. Available at: https://www.spec.org/cpu2017 (accessed 05.11.2017)

ИНФОРМАЦИЯ ОБ АВТОРАХ

Русяев Роман Михайлович, аспирант, ПАО «ИНЭУМ им. И.С. Брука»; инженер-программист, АО «МЦСТ», 119334, Москва, ул. Вавилова, д. 24, тел.: 8 (499) 135-60-94, e-mail: roman.m.rusiaev@mcst.ru.

Баранников Сергей Владимирович, инженер-программист, АО «МЦСТ», 119334, Москва, ул. Вавилова, д. 24, тел.: 8 (499) 135-60-94, e-mail: sergey.v.barannikov@mcst.ru.

Нейман-заде Мурад Искендер-оглы, к.ф.-м.н., начальник отделения, АО «МЦСТ», ПАО «ИНЭУМ им. И.С. Брука»; доцент, МФТИ (ГУ), 119334, Москва, ул. Вавилова, д. 24, тел.: 8 (499) 135-88-69, e-mail: muradnz@mcst.ru.

For citation: Rusyaev R.M., Barannikov S.V., Neiman-Zade M.I. Features of exceptions handling implementation for Elbrus architecture. Voprosy radioelektroniki, 2018, no. 2, pp. 45–50.

R.M. Rusyaev, S.V. Barannikov, M.I. Neiman-Zade

FEATURES OF EXCEPTIONS HANDLING IMPLEMENTATION FOR ELBRUS ARCHITECTURE

An exception mechanism is an important part of large applications written in high-level languages such as C++, so there is an urgent need for their effective implementation. This paper discusses the approach to exception handling, called zero cost exceptions handling, which helps to minimize the overhead of exceptions handling in user applications. This is achieved by localization of the auxiliary code required to handle exceptions to special areas called landing pad. A detailed description of this mechanism implementation is given, showing the way of its overhead minimization. Also an overview of the setjmp/longjmp-based approach is given. This approach is used in the previous version of the compiler. The innovations and improvements on the part of the compiler and the optimizing passes necessary for the implementation of the mechanism under consideration are described. The results of the performance evaluation confirm the advantage of the zero cost exceptions handling mechanism in comparison with the setjmp/longjmp mechanism.

Keywords: Elbrus architecture, C++ exceptions, exceptions handling, zero cost exceptions handling.

REFERENCES

1. Christophe de Dinechin. [C++ Exception Handling for IA-64]. Available at: http://static.usenix.org/events/osdi2000/wiess2000/full_papers/dinechin/dinechin_html accessed 05.11.2017)
2. [Itanium C++ ABI: Exception Handling]. Available at: http://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html (accessed 05.11.2017)
3. [Exception Handling in LLVM]. Available at: http://llvm.org/docs/ExceptionHandling.html (accessed 05.11.2017)
4. [Exceptions Handling Tables]. Available at: https://itanium-cxx-abi.github.io/cxx-abi/exceptions.pdf (accessed 05.11.2017)
5. [SPEC CPU2006]. Available at: https://www.spec.org/cpu2006 (accessed 05.11.2017)
6. [SPEC CPU2017]. Available at: https://www.spec.org/cpu2017 (accessed 05.11.2017)

AUTHORS

Rusyaev Roman, graduate student, PJSC Brook INEUM; software engineer, JSC MCST, 24, ulitsa Vavilova, Moscow, 119334, Russian Federation, tel.: +7 (499) 135-60-94, e-mail: roman.m.rusiaev@mcst.ru.

Barannikov Sergey, software engineer, JSC MCST, 24, ulitsa Vavilova, Moscow, 119334, Russian Federation, tel.: +7 (499) 135-60-94, e-mail: sergey.v.barannikov@mcst.ru.

Neiman-zade Murad, PhD, head of Department, JSC MCST, PJSC Brook INEUM; assistant professor, MIPT, 24, ulitsa Vavilova, Moscow, 119334, Russian Federation, tel.: +7 (499) 135-88-69, e-mail: muradnz@mcst.ru.