Performance optimizations in an LLVM-based cloud application store[1]

Victor Ivannikov
ISP RAS
Moscow, Russia
e-mail: ivan@ispras.ru

Shamil Kurmangaleev
ISP RAS
Moscow, Russia
e-mail: kursh@ispras.ru

Andrey Belevantsev
ISP RAS
Moscow, Russia
e-mail: abel@ispras.ru

Arutyun Avetisyan
ISP RAS
Moscow, Russia
e-mail: arut@ispras.ru

ABSTRACT
This paper describes the two-stage compilation system based on LLVM compiler infrastructure and the performance optimizations made possible by this deployment technique.

Keywords
LLVM, compilation, optimizations, devirtualization, profile information.

1. INTRODUCTION
Deploying applications written in general-purpose languages (C/C++-like) using native binary formats yields several difficulties. First, dozens of versions are required so that the target architecture specifics can be fully taken into account, which can result in significant performance increase. Second, profile-guided optimizations using the profile specific to the given application user are impossible.

The paper presents the two-stage compilation technique that allows distributing such programs in LLVM [1] bitcode. LLVM bitcode can be made portable within the same architecture family or with some limitations also within the architectures that have the same basic types' size.

The two-stage compilation workflow [2] is organized as follows. On the first stage, we generate the setup package with the compiled bit-code and the auto-generated scripts that allow its code generation to the native code on the target device. The setup package is built through utilizing a configure script called configure-proxy, which automatically generates the build script. The original compilers are also invoked using similarly organized wrappers. Also, we have made the changes to the compiler frontend and the linker allowing to record dependencies between separate program modules.

The setup package construction is finalized after the program build and installation. All recorded dependencies (built libraries and executables in the LLVM bitcode format, files that were created during program installation) are included in the package together with the generated scripts for the target device.

We support invoking link-time optimizations to improve the generated code performance. More generally, it is important to optimize code on the first stage as much as possible to lower the burden on the second stage optimizations, which might happen on the mobile target device. On the second stage, we support two installation choices – either static code generation into the native target code or dynamic compilation with profile information generated by the user. Both static and dynamic compilation paths can fully optimize target-specific knowledge while optimizing (e.g. code generation, auto vectorization, software pipelining and scheduling, prefetching, etc.). The implemented tools allow the programmers to support static and dynamic optimization between program bitcode and other modules fully automatically (which may be linked either statically or dynamically and may also be distributed either as native code or bitcode).

2. LLVM BITCODE APPLICATION SERVER
When using two-stage compilation workflow for mobile devices, one must be careful to select the proper optimizations to be executed on the target device, as it is usually limited both in available performance and memory. It is desirable to remove the optimization burden from the target by utilizing a separate cloud application server (known as App Store), which can perform target-specific optimization and code generation for each user mobile device separately and transparently to the user, given that the applications are stored in the setup package format with LLVM bitcode described above.

To perform profile-specific optimizations in the cloud, it is required that the target device should occasionally send the profile information to the cloud, so that the application server can utilize this information and distribute the updated version of the application back to the user. Different strategies of handling profile-specific information are possible. The simplest way will be to unify all user profiles, which lowers the resources required for profile-based optimizations but delivers an average-performing application to every user. Other strategy could be to support saving several sets of profile information corresponding to different classes of application use cases, so that on receiving the new profile information the application server could classify it as belonging to the known class and then deliver the correspondingly optimizing application, or otherwise to create a new profile information class if the new data is substantially different from existing ones.

Storing applications in LLVM bitcode format in the cloud application server calls for other use cases, such as automatic security vulnerability checks using static analysis tools similar to Svace[3], or implementing obfuscation techniques [4] to create unique application versions for every user, which will make creating universal security exploits substantially harder.

[1] We acknowledge support from the Russian Foundation for Basic Research grant #11-01-00954-a

3. TWO-STAGE TOOLCHAIN PERFORMANCE OPTIMIZATIONS

In this section we sketch the most important optimizations that were developed for the two-stage compilation toolchain. The optimizations were tested on ARM devices on a number of benchmarks and packages, including SPEC CPU 2000, Coremark, SQLite, CLucene, Cray and Expedite.

3.1. Speculative Devirtualization

Devirtualization [5] is an analysis technique that allows to determine the possible set of targets for a function pointer call. It is important for object-oriented languages like C++ where any virtual method call results in a function pointer call. Devirtualization allows either to replace an indirect call with a direct one, if there is only one possible target, or to speculatively choose one of the possible variants based on the generated runtime check.

We have implemented a speculative devirtualization approach based on comparing function signatures, type hierarchy graph analysis (also known as inheritance graph), and static analysis of possible types for the given function pointer. When we determine several possible candidates, we generate a single runtime check for the hottest version for its speculative devirtualization and do not devirtualize the remaining cold variants.

The LLVM bitcode does not contain the high-level type information required for devirtualization, so that we have implemented the type hierarchy analysis in the Clang frontend with saving its results in the LLVM bitcode metadata. On our benchmarks we have received ~3% performance increase with just 1% code size increase. When manually assuming closed world application for CLucene, we have received 10% performance improvement. The implementation also passes devirtualization tests of the GCC [6] compiler. We are working on properly recognizing all variants of external functions to avoid their too aggressive devirtualization and on improving static type analysis precision.

3.2. Array Prefetching in Loops

The array prefetching optimization inserts target-dependent prefetching commands for optimizing cache utilization when processing array data in loops. The prefetching commands should be executed well in advance so that the required data would be loaded in cache in time. This requires the knowledge of the number of commands executed between the prefetching command and the data usage. As we have implemented this optimization over the LLVM bitcode, which is not machine-specific, we use heuristic estimation of the number of executed machine commands for every bitcode instruction (e.g. for calls the number is dependent on the number of arguments, for some instructions there is no machine commands generated, etc.).

We also need to estimate the number of loop iterations that we need to prefetch data for. This number can be estimated as a division of prefetching delay commands number to the number of machine commands in a single loop iteration. For example, for ARM Cortex-A9 about 200 commands are executed before the data being prefetched gets loaded to the cache [7]. The number of loop iterations are estimated either through static analysis or from the program profile when it is available.

We unroll loops to avoid prefetching data too often (the data which is already in the cache does not need prefetching). The unroll factor is determined so that the data required for the single iteration of the unrolled loop will occupy one cache line. For example, the cache line size on ARM Cortex-A9 is 32 bytes, and if one loop iteration loads just 4 bytes, we need to unroll the loop 8 times.

The performance increase because of prefetching on SPEC CPU 2000 is ~0.9%, the increase for SQLite, Expedite, Cray and Coremark is ~0.5%-5%, averaging ~2.5%.

3.3. Function Inlining

We have improved the existing LLVM function inlining based on the GCC inlining implementation. We were mostly interested in the profile information usage for inlining. The decision on whether to inline a callee function is based on the relative frequency of function calls, their absolute number, and the caller function growth estimation done in target-independent manner and taking into account possible simple optimizations enabled by inlining.

We estimate function weight as follows:

```
FunctionWeight = NumofInstructions ×
InstructionPenalty – NumArguments ×
AllocaPenalty – NumofConstInstruction ×
ConstantPenalty,
```

where `InstructionPenalty=2` is the instruction cost, `AllocaPenalty=2` is the local variable cost, `ConstantPenalty=2` is the constant value cost. The function weight with taking profile information into account is calculated as follows:

```
NewWeight= NumCallProfileInfo/FunctionWeight
```

After function weight calculation we sort functions by increasing weight and inline them until the total weight exceeds the given threshold. The performance increase for SQLite, Expedite, Cray, and Coremark benchmarks is ~2%.

3.3. Function Outlining

The frequent pattern for writing functions is the function consisting of two parts: a relatively small hot part (e.g., a quick check for hot data and resulting fast calculations) and a large cold part (more expensive calculation when the data is absent or corrupt)[8]. It is desirable to outline the hot function part in a separate function, which then may be separately optimized (e.g. inlined), and the cold part may be placed in the separate executable section so that it does not interfere with the hot program parts.

First, we determine hot functions based on the k-means clusterization algorithm. We support three classes – hot, medium, and cold functions. For hot functions, we in turn determine cold control flow edges (relative to the other edge of the single conditional jump) and then we select the control flow region that can be executed only by transferring flow over cold edges (we select basic blocks that are dominated by the cold edge target blocks)[8][9].

The performance increase on SQLite, Expedite, Cray, and Coremark benchmarks is ~0.8%, and we get ~3% increase when combining this technique with function inlining. The code size increase is ~1-7% depending on the application.

3.3. Determining Dynamic Optimization Level

It is widely known that during dynamic optimization (JIT) only the hottest program parts should be heavily optimized, and the rest can be optimized only slightly as their performance does not affect significantly the total program performance. JIT compilers often utilize heuristic approaches to find out the proper optimization level for the given function [10],[11].

When testing our two-stage compilation approach on the target mobile ARM-based devices, we have implemented three variants of optimization levels like below:

- Minimal: no optimization (level zero) for cold functions and level two (so called standard optimizations) for hot functions;
- Medium: light optimizations (level one) for cold functions and aggressive optimizations (level three) for hot functions;
- Maximum: level two optimizations for cold functions and level three optimizations for hot functions.

When testing on SQLite benchmark, we have found that the minimal level saves up to 90% compile time with the performance similar to the standard level two optimizations. For medium level, we save 2-5% compile time with the performance increase of 1-3% compared with the level three optimizations. The maximum level produces performance increase of 3-4% when compared with the level three optimizations and saves 1-3% compile time.

4. CONCLUSION

The paper describes the two-stage compilation toolchain based on LLVM bitcode, which allows distributing applications in bitcode and optimizing them in the cloud application server with fully taking into account target-specific hardware features and user-specific profile information. Deploying application through the cloud server allows checking their bitcode form for security vulnerabilities and other defects. The toolchain is also capable in operating directly on the target device supporting either static code generation or just-in-time optimizations.

A number of profile-directed optimizations were implemented in the toolchain, providing performance increases between 1% and 7-10% depending on the application. The most beneficial optimizations include speculative devirtualization and function inlining/outlining.

REFERENCES

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

[2] Arutyun Avetisyan. "Two-stage compilation for optimizing and deploying programs in general purpose languages", Proceedings of the Institute for System Programming of RAS, volume 22, pp. 11-18. , 2012

[3] Arutyun Avetisyan, Andrey Belevantsev, Alexey Borodin, Vladimir Nesov, "Using static analysis for finding security vulnerabilities and critical errors in source code.", Proceedings of the Institute for System Programming of RAS, volume 21, pp. 23-38, 2011

[4] Kurmangaleev S.F. Korchagin V.P. Savchenko V.V. Sargsyan S.S. “Building obfuscation compiler based on LLVM infrastructure”. Proceedings of the Institute for System Programming of RAS, volume 23, pp. 77-92, 2012

[5] David F. Bacon and Peter F. Sweeny, “Fast Static Analysis of C++ Virtual Functuion Call”, OOPSLA '96 Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications pp 324-341, 1996

[6] GCC Free software foundation, http://gcc.gnu.org

[7] ARM Architecture Reference Manual, http://infocenter.arm.com

[8] Peng Zhao “Code and Data Outlining”, Doctor of Philosophy thesis, Edmonton, Alberta, 2005.

[9] Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon, Suhyun Kim ”Aggressive Function Splitting for Partial Inlining”, INTERACT’11 Proceedings of the 2011 15th Workshop on Interaction between Compilers and Computer Architectures, Seoul, South Korea, pp. 80-86, 2011.

[10] Da Silva, A. F. “Our Experiences with Optimizations in Sun’s Java Just-In-Time Compilers” Journal of Universal Computer Science. Vol. 12, pp. 788-810, 2006.

[11] M. Arnold, Stephen J. Fink, D. Grove, M. Hind, Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. Proceedings of IEEE, vol.93, No.2, pp. 449-466, 2005.