LLVM performance in HTTP Environments: Analysis and Optimization

John Jawed
j@jawed.name
5001 Great America Pkwy.
Santa Clara, CA, 95054, USA

ABSTRACT
Low Level Virtual Machine, or LLVM, is an open source compiler infrastructure [1]. LLVM allows optimization of software applications at the compile, link and run times. Equivalent front ends for languages supported by GNU's C Compiler are provided by the LLVM project. This document attempts to analyze the performance impact of using LLVM's C compiler in place of the GNU C Compiler with Hypertext Preprocessor (PHP).

Keywords
Low Level Virtual Machine, Performance Optimization, PHP, Apache, HTTP

1. INTRODUCTION
This document focuses on intrinsically measuring the performance impact of LLVM's C compiler on the PHP project and PHP applications.

In Section 2, LLVM components and internals are discussed. Section 3 and 4 describe data collected during benchmarks as well as procedure. Finally, Section 4 briefly discusses migrating existing environments and steps to utilize LLVM C Compiler.

2. LOW LEVEL VIRTUAL MACHINE
This section consists of a brief introduction to LLVM components and internals.

LLVM provides an efficient and well defined framework for compilers. LLVM libraries provide optimizations, a JIT compiler, GC, and link time optimization. The intermediate representation for LLVM is language and target independent. More importantly, LLVM provides an optimizer which works with the LLVM IR format. The most relevant component, for this document's purposes, the LLVM C compiler which is discussed in section 2.1.

All major platforms are supported by LLVM, however not all components of LLVM are supported on any given platform.

2.1 LLVM C
The LLVM C front end, `llvm-gcc`, is a modified but equivalent version of `gcc` which has the ability compile C programs into several different formats. By default, the behavior of `llvm-gcc` is equivalent to `gcc` in which the programs are compiled into native object files (*.o). Alternatively, programs can be compiled into LLVM bit code or ASM. LLVM's C front end also has support for GNU C extensions.

`llvm-gcc` initially compiles the program into LLVM IR, a mid-level optimizer is run on the resulting LLVM IR which is finally run through an arch-specific code generator. Optionally, LLVM enables another pass through the mid-level optimizer at link time, referred to as Link Time Optimization (LTO). LTO is not supported on all platforms, at the time this document was written there was no native support for LTO in `llvm-gcc` on non-Darwin hosts (use the `gold` plugin to achieve this). LLVM's mid-level optimizer can be used at link time which allows an extra level of optimizations not available without tight integration with the linker.

3. BENCHMARKING
This section discusses the methodology, environment and process in which benchmarks used to measure performance are conducted.

The goal is to ensure a consistent environment by not modifying any control variables. Except where noted, all timings are the best of 3 samples.

3.1 ENVIRONMENT
The following describes hardware and software variables which are considered to be controlled. Ethernet performance is considered to be relatively controlled.

The system used consists of the following hardware: Intel Core 2 Duo, 1024MB of RAM, 1.07 GHz bus speed. The system software consists of Linux 2.6.23, Apache 2.2.9, MySQL 5.0.67, GCC 4.2 (EABI), LLVM 2.5, LLVM gcc 4.2 (bootstrapped), PHP 5.3 (CVS). Except for tests with `llvm-gcc` (which does not add the `-msse2` compiler flag, unless noted) all compiler flags are: `-O3 -fomit-frame-pointer -Wall`. Additional dependencies which were compiled with `llvm-gcc` in LLVM tests are listed in Appendix A.

3.2 PERFORMANCE TEST #1
This performance test consists of running the PHP test suite (`make tests`).

The main purpose of this test is to seek out any perceived performance benefits, thus only the running of the test suite is timed (with the `time` command). This approach serves as a good initial and litmus test but it is not practical enough to be applied outside this document. The test suite consists of 9386 cases which test functionality, security and language constructs. Wall time improved by 32% with `llvm-gcc` when running the test suite.

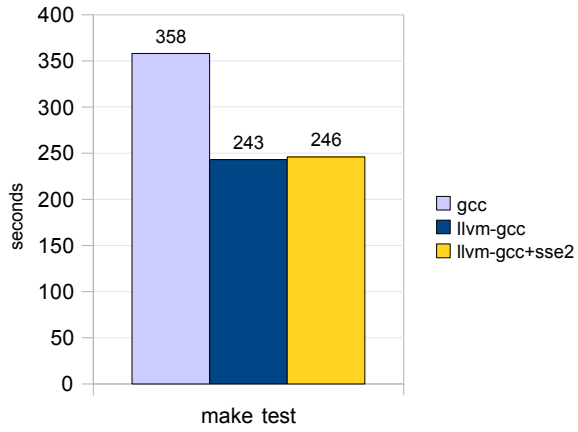


Figure 1. Run times of *make test* with *llvm-gcc* and vanilla *gcc*

3.3 PERFORMANCE TEST #2

This performance test consists of running the standard benchmark suite included with the PHP distribution.

The main purpose of this test is to investigate and analyze which areas experience the most improvements with `llvm-gcc`.

To facilitate analysis and add greater contrast to the timings, the standard Zend benchmark is modified to be executed 3 times in sequence.

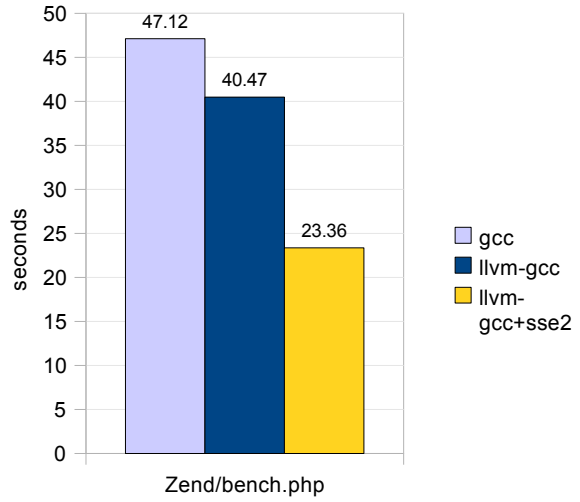


Figure 2. *Zend/bench.php* run times with PHP compiled by *llvm-gcc* and vanilla *gcc*

The `Zend/bench.php` benchmark executes a variety of timed tests which measure the performance of common algorithms, various control structures, built-in and user functions.

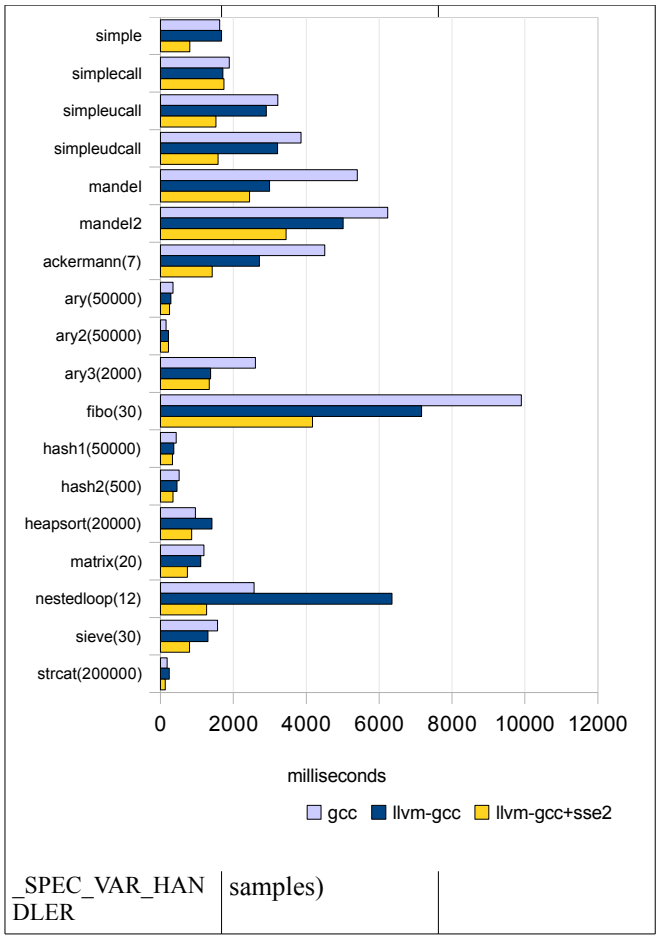
Analysis of the individual tests indicate performance gain was most evident for complex algorithms when the PHP binary was compiled with `llvm-gcc` versus `gcc`. However, runtime performance declined by more than a factor of 2 when benchmarking `nestedloops(12)` without the use of SSE2 instructions.

Upon further investigation the symbol responsible for the `nestedloop(12)` slowdown is determined, `ZEND_POST_INC_SPEC_CV_HANDLER`, which increments the value and goes to the next opcode in the Zend Engine. With `llvm-gcc` `ZEND_POST_INC_SPEC_CV_HANDLER` is sampled more than 16x than the corresponding `gcc` code generation (sampled via timer interrupts at 100HZ).

Debug information for sampling symbols is enabled with the compiler flags `-g` exclusively of the timing tests.

Table 1. Timer interrupt based samples of each symbol in *llvm-gcc* vs *gcc* PHP binaries for *nestedloop(12)*

Symbol	`llvm-gcc` samples	`gcc` samples
ZEND_POST_INC_SPEC_CV_HANDLER	2182 (61% of all samples)	137 (10% of all samples)
ZEND_POST_DEC	440 (11% of all	<1 (-)



_SPEC_VAR_HANDLER	samples)	

Figure 4. Individual performance timings for *Zend/bench.php*

The next step in this research is to disassemble the instructions for the symbol ZEND_POST_INC_SPEC_CV_HANDLER and try to locate any differences.

Because the slowdown is within the Zend VM's value increment handling symbol during the `nestedloop(12)` tests, `zend_execute.c` x86 assembly would contain the `ZEND_POST_INC_SPEC_CV_HANDLER` function. At first observation, there are fewer instructions generated by the `llvm-gcc` code generator vs the `gcc` code generator. However, the following FP math instructions contained in the `llvm-gcc` code generation appear to be the root cause:

```
…
movl   executor_globals+20, %ecx
fldl   (%ecx)
fstpl  (%eax)
movl   8(%ecx), %edx
…
```

The corresponding assembly generated by the `gcc` code generator does not use the `fldl`/`fstpl` instructions. Instead, the `gcc` code generator uses the `Streaming SIMD Extensions (SSE)` to handle the FP math and avoid the otherwise [relatively] expensive instructions of copying, pushing and popping registers on/off the FPU stack. It is probable that the `llvm-gcc` assembly would be slower considering the `fldl`/`fstpl` instructions are looped in each loop.

After mentioning this document's findings to the LLVM authors the addition of the `-msse2` flag was suggested, which explicitly asks `llvm-gcc` to use the `SSE2` extended instruction sets available in the test environment CPU (`cat /proc/cpuinfo | grep`

flags). Using -msse2 will imply the usage -mfpmath=sse. By default, gcc enables -mfpmath=sse with the x86_64 version of their compiler.
With the -msse2 compiler flag nestedloop(12) timings decreased to 1265ms.

3.4 PERFORMANCE TEST #3

This performance test consists of benchmarking a widely used web application which utilizes PHP.

This benchmark uses WordPress 2.7, an open source content publishing system built on PHP and MySQL. The HTTP benchmarking with be conducted with siege, an open source utility which provides detailed analysis of stimulated HTTP loads.

siege is run on an external hardware host which is on the same wired ethernet Virtual Local Area Network. Alternate PHP Cache is enabled and PHP is compiled as an Apache module.

The HTTP request will be to a dynamic page preloaded with data imported from the web.

For the HTTP loads, initially 5 concurrent connections are established and are making requests for 60 seconds, each subsequent test load increases the number of concurrent connections by 5.

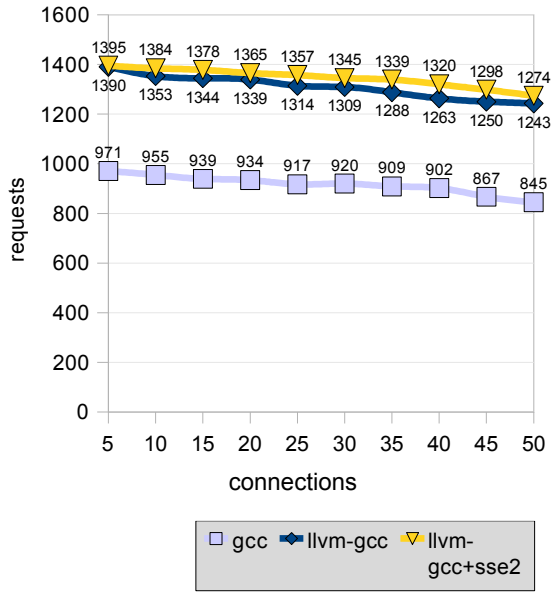


Figure 5. Concurrent requests handled over a period of 60 seconds

A WordPress environment compiled with llvm-gcc rather than gcc provides a 43% performance gain, and provided sustained 40-46% performance gains. A WordPress environment compiled with llvm-gcc and SSE2 instructions increased performance by 43-50% when compared to gcc. After 50 concurrent connections the test environment server could not handle all requests reliably (for both llvm-gcc and gcc).

4. MIGRATION

This section attempts to provide insight on migration aspects for an environment or project which currently uses GNU's C compiler to LLVM's equivalent C compiler.

It is important to emphasize that, as with all performance benchmarks, any performance/capacity gains vary or may even be non-existent in different environments.
The following sections do not cover all aspects of any compiler migration; they only serves to highlight aspects which became apparent in the course of publishing this document or to facilitate further investigation.

4.1 BUILD

In most cases updating build tools will be a matter of switching the C compiler (often times represented by the CC variable in GNU Make files) from gcc to llvm-gcc. In section 3.3, a significant performance drop was encountered due to suboptimal compiler flags for llvm-gcc. Default floating point math behavior for gcc appeared to be different with llvm-gcc without specifying the -mfpmath=sse (implied by using -msse2) flag. Alternatively, LLVM can be configured using --with-arch which can contain the environment architecture that supports SSE2 (or SSE3). Migrated environments should examine application flows to ensure there no regressions in performance or otherwise.

Not all code which compiles with gcc will compile without modification with llvm-gcc, however most code should compile cleanly.

4.2 SECURITY

Not all compilers are created equal, thus, not all compilers optimize code in the same manner. In fact, even different versions of the same compiler software may optimize code differently. While the differences maybe inconsequential there is a potential optimizations could lead to new security holes. Consider the following C code which checks to see if some data length does not exceed the range of data:

```
…
unsigned long length_of_data;
char data[MAX_LENGTH];
char *end_of_data = data + MAX_LENGTH;

if (
data + length_of_data >= end_of_data
||
data + length_of_data < data)
    fprintf(stderr,”Buffer overflow”);
…
```

In the code snippet above, length_of_data, could be the length of data specified by a caller. The code is not C standard compliant, in fact, it is very poorly written but valid nonetheless. The first check tries to make sure that the buffer fits within the data buffer, since the pointer may overflow the second check ensures length_of_data is within the range of data. Most compilers will optimize out the second check and introduce a buffer overflow. The C standard allows compilers to do this because pointer addition does not yield a pointer value outside data.

C89/99 [standard] compliant code should not have any new issues with different compilers but application integrity should be re-evaluated in migrated environments.

5. CONCLUSION

This document shows there is potential for the LLVM based C compiler to provide more performance than the GNU C compiler. Building a WordPress environment with llvm-gcc as opposed to gcc improved performance in all benchmarks mentioned in this document. WordPress request handling capacity increased by 40-46%. When using llvm-gcc with SSE2 instructions performance increased by 43-50% for WordPress. This document's findings are

supported by past work on other projects, notably the work done with the Perl language which exhibited a 15% performance gain for PerlBench.
Each application and environment is unique, the findings of this document should not serve as a baseline comparison for implementations. The only goal of this document, is to facilitate further investigation of potential benefits for application vendors by pursuing an LLVM C compiler based solution.

Furthermore this document did not explore the advantage of LLVM's LTO, it is reasonable to expect that one may be able to achieve further performance gains with LTO, or a smaller binary with less machine code.

6. ACKNOWLEDGMENTS

Appreciation for guidance during the authoring of this document goes to the LLVM contributors and authors. In particular, Török Edwin and Chris Lattner for their assistance regarding LLVM internals and feedback in publishing this document.

7. REFERENCES

[1] Chris Lattner and Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.* IEEE Computer Society, 2004.
[2] Chris Lattner and Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.* IEEE Computer Society, 2004.