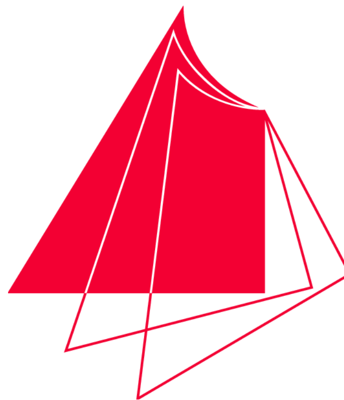
Clang/LLVM

Maturity Evaluation Report



Author: B.Sc. Dominic Fandrey

Supervisor: Prof. Dr. rer. nat. Thomas Fuchß

Institute: Karlsruhe University of Applied Sciences
Computer Science department

Completed: Sunday 6 June 2010

Table of Contents

1.Introduction....................5
1.1.Motivation....................5
1.2.What is a Compiler....................7
1.3.Compilation Tasks....................7
1.4.Compiler Architecture....................8
1.5.Introduction to Clang/LLVM....................11
2.The Makings of a Good Compiler....................19
2.1.Compiler Performance....................19
2.2.Binary Performance....................20
2.3.C/C++ Compiler Performance Check List....................21
3.Clang/GCC Compile Time Comparisons....................23
3.1.ClangBSD Compile Time....................23
3.1.1.Understanding Charts....................27
3.1.2.ClangBSD SVN r207367 buildworld....................28
3.1.3.ClangBSD SVN r207367 buildkernel....................33
3.2.Reasons for Differences in Compile Time....................37
4.Clang/GCC Binary Performance Measurements....................39
5.Conclusion....................41
6.Appendix....................43

1. Introduction

This report describes the current maturity state of the Clang/LLVM C/C++ compiler. The main focus is on compile time and memory consumption. Other aspects of compiler quality have already been subject to sufficient independent research. Select examples are summarized and referenced to ease further studying of the subject.

1.1. Motivation

For many years the GNU Compiler Collection (GCC) was the de facto standard complier of the open source community. In 2007 the Free Software Foundation released version 3 of their popular GPL license [GnuGpl]. The GCC project uses this license from the GCC4.3 branch on.

The GPL version 3 introduces new licensing restrictions for patent holders that some open source operating system projects do not want to impose on their users. These, commonly BSD-style licensed systems, are currently stuck with the unmaintained 4.2 branch of GCC[1] and in desperate need for a compiler replacement.

One possible candidate is Clang/LLVM. The FreeBSD Status Reports January - March, 2009, published by Brad Davis on freebsd-announce@FreeBSD.org [FbsdAn2009] announced:

> … . The FreeBSD project is looking at the possibility to replace GCC with Clang as a system compiler. It can compile 99% of the FreeBSD world and can compile booting kernel on i386/amd64 but it still contains bugs and its C++ support is still immature.

LLVM has been gaining public attention during the last years, which is well documented on the LLVM publications page [LlvmPubs]:

1 27 April 2010 the latest release branch was GCC 4.5.0, GCC 4.6.0 was in active development and the GCC 4.2. branch was no longer maintained (http://gcc.gnu.org/).

1.Introduction

Clang has recently had some major breakthroughs. Doug Gregor wrote on the 4th February, 2010 on the LLVM Project Blog [Gregor201002]:

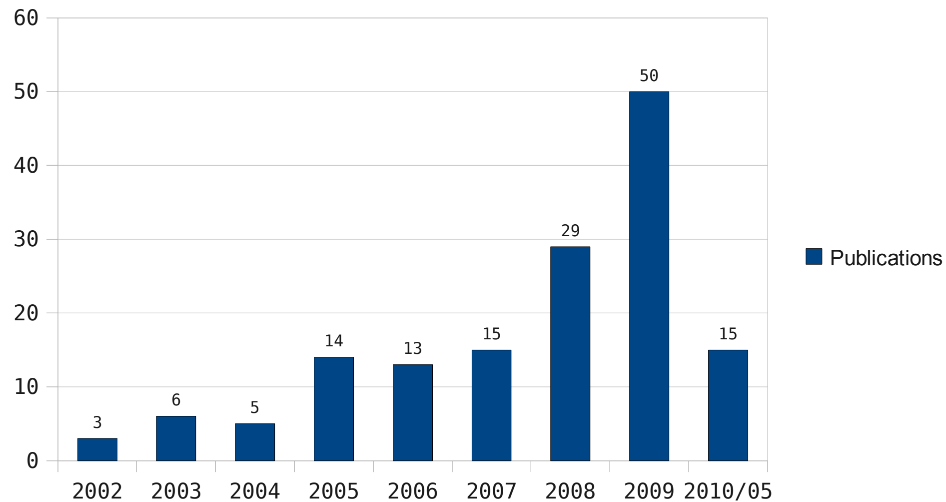


Illustration 1: Known LLVM related publications

> Today, Clang completed its first complete self-host! We built all of LLVM and Clang with Clang (over 550k lines of C++ code). The resulting binaries passed all of Clang and LLVM's regression test suites, and the Clang-built Clang could then build all of LLVM and Clang again. The third-stage Clang was also fully-functional, completing the bootstrap.

On the 29th of May, 2010 Roman Divacky announced on freebsd-current@FreeBSD.org, that the Clang testing branch of FreeBSD would soon be merged with the major FreeBSD development branch [Divacky201005]:

> ClangBSD was updated to LLVM/clang revision 104832 which is what we aim to import into HEAD in roughly a week.

1.2. What is a Compiler

On page 5 of the introduction to “Compiler Construction” [Wirth1996], Niklas Wirth answers the basic question of what constitutes a compiler:

Computer programs are formulated in a programming language and specify classes of computing processes. Computers, however, interpret sequences of particular instructions, but not program texts. Therefore, the program text must be translated into a suitable instruction sequence before it can be processed by a computer. This translation can be automated, which implies that it can be formulated as a program itself. The translation program is called a compiler, and the text to be translated is called source text (or sometimes source code).

1.3. Compilation Tasks

The book then continues to divide compilation into four tasks:

1. Lexical analysis, the source code is divided into tokens
2. Syntax analysis, the syntax tree is built from the tokens
3. Type checking, operators and operands are checked for compatibility
4. Code generation, the machine readable code is generated

In the case of C/C++, a preprocessor step has to be performed prior to step 1. The C preprocessor performs header inclusion, code macro substitution and in the case of C also defines constants. Furthermore the preprocessor is used to trigger platform specific hacks, to work around non-portable code.

The fourth step is in the case of C/C++ normally divided into three steps:

a) Assembler generation, the abstract representation is translated into a platform specific assembler dialect (.d)

b) Assemble, the code is converted into binary object files (.o)

c) Link, the object files are linked to executable binaries and libraries (.so | bin on *nix, .dll | .exe on MS Windows platforms)

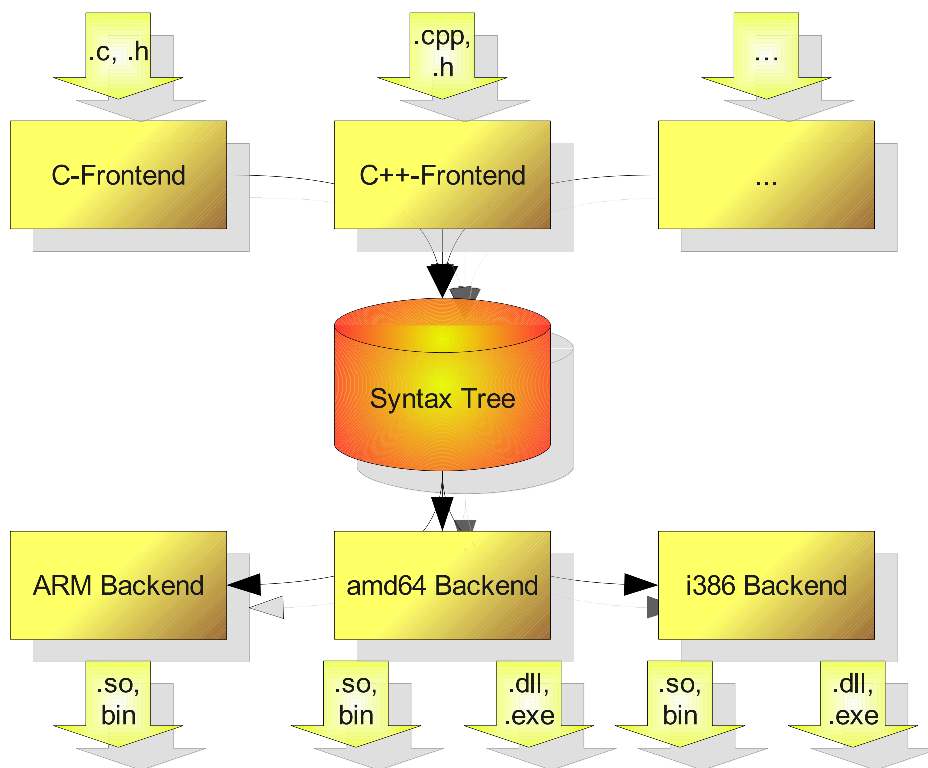
1.4. Compiler Architecture

On page 7 Wirth recommends a two component architecture:

> A wise compromise exists in the form of a compiler with two parts, namely a front end and a back end. The first part comprises lexical and syntax analyses and type checking, and it generates a tree representing the syntactic structure of the source text. This tree is held in main store and constitutes the interface to the second part which handles code generation. The main advantage of this solution lies in the independence of the front end of the target computer and its instruction set. This advantage is inestimable if compilers for the same language and for various computers must be constructed, because the same front end serves them all.

Drawing 1 illustrates the data flow of such a frontend, backend solution.

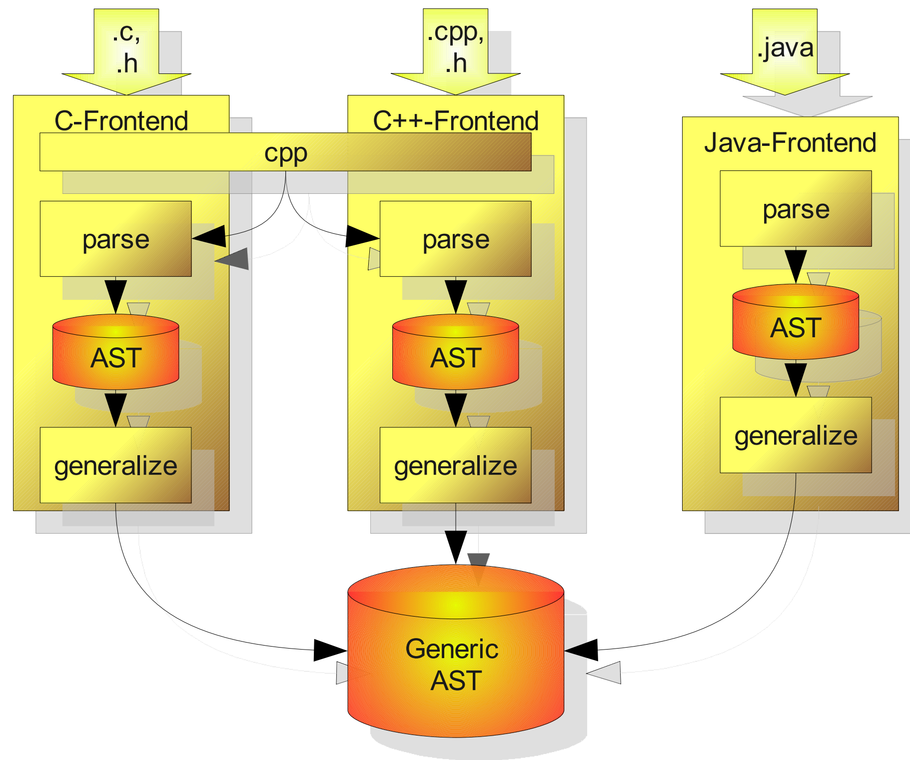
Labelled arrows stand for input and output files, lines with arrow heads present the data flow within compiler components. Dotted lines represent the execution order, where data is just worked on instead of being converted.



Drawing 1: Compiler reference architecture

The GNU Compiler Collection divides the backend into a universal middle end and an architecture specific backend. Drawing 2 and Drawing 3 are both based on a diagram in the WikiBooks [WikiBooks] article “GNU C Compiler Internals” [WBGccArch], which can be discovered following the links in the documentation of the official GNU GCC project home page [GnuGcc].

The GCC frontend performs two tasks, parsing an input file into an Abstract Syntax Tree (AST), and the conversion into a generic representation of the AST.

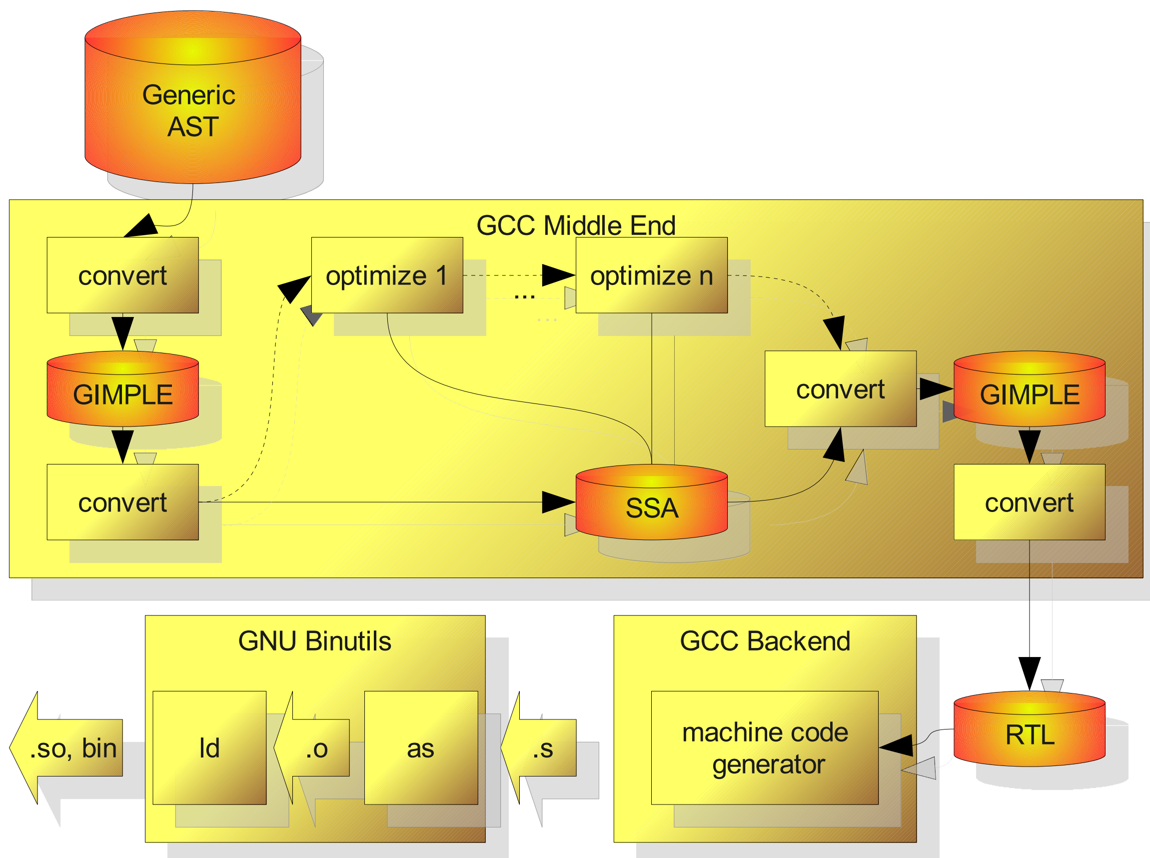


Drawing 2: The GCC frontend

The generic AST representation is passed on to the GCC middle end. There it is converted into a representation called GIMPLE, which is described as “a convenient representation for optimizing the source code” by the same WikiBooks article.

From there it is converted into a so called Static Single Assignment (SSA) representation, upon which more than 20 optional optimization passes can be performed. After the optimization passes the SSA representation is converted back into GIMPLE and from there into the Register Transfer Language (RTL).

The RTL is an assembler language for an imaginary processor architecture. It is passed on to the backend, which converts it into the machine code for the target architecture.



Drawing 3: GCC middle and backend

1.5. Introduction to Clang/LLVM

Being a young project not much effort has been spent on recording the history of Clang/LLVM and its major creators and contributors. One of the few valuable sources available is the article “Apple's other open secret: the LLVM Compiler” by Prince McLean [McLean2008], which was published in the online magazine AppleInsider [AppleInsider].

The LLVM project started as a research project of Chris Lattner at the University of Illinois in 2000 and was first released in 2003. Lattner later caught the attention of Apple, which started contributing to the project in 2005 and later employing Lattner to fund his work.

1.Introduction

LLVM has since been subject to a lot of attention in university research, whereas Apple has started incorporating LLVM into its products. Some of these uses are detailed in the aforementioned article.

Illustration 2: The LLVM logo

In order to understand the architecture of the Clang frontend it is first necessary to understand some aspects of the LLVM backend. In 2004 Chris Lattner and Vikram Adve wrote released the paper “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformationin” [LattnerAdve2004], which describes the LLVM architecture in detail:

> This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support transparent, life-long program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs.

A basic design principle of LLVM is that all components work on the same intermediate representation (LLVM IR), which is described in Chapter 2 of the paper:

> The code representation is one of the key factors that differentiates LLVM from other systems. The representation is designed to provide high-level information about programs that is needed to support sophisticated analyses and transformations, while being low-level enough to represent arbitrary programs and to permit extensive optimization in static compilers.

Similar to the GCC optimizer the LLVM backend uses an SSA form, the main difference is that this SSA form is the intermediate language used throughout the entire backend:

> LLVM uses SSA form as its primary code representation, i.e., each virtual register is written in exactly one instruction, and each use of a register is dominated by its definition. Memory locations in LLVM are not in SSA form because many possible locations may be modified at a single store through a pointer, making it difficult to construct a reasonably compact, explicit SSA code representation for such locations.

To suit different purposes, the LLVM IR exists in three equivalent representations:

1.Introduction

> The LLVM representation is a first class language which defines equivalent textual, binary, and in-memory (i.e., compiler’s internal) representations. The instruction set is designed to serve effectively both as a persistent, offline code representation and as a compiler internal representation, with no semantic conversions needed between the two. Being able to convert LLVM code between these representations without information loss makes debugging transformations much simpler, allows test cases to be written easily, and decreases the amount of time required to understand the in-memory representation.

Chapter 3.2 describes the general layout of an LLVM compiler frontend:

> External static LLVM compilers (referred to as front-ends) translate source-language programs into the LLVM virtual instruction set. Each static compiler can perform three key tasks, of which the first and third are optional: (1) Perform language-specific optimizations, e.g., optimizing closures in languages with higher-order functions. (2) Translate source programs to LLVM code, synthesizing as much useful LLVM type information as possible, especially to expose pointers, structures, and arrays. (3) Invoke LLVM passes for global or interprocedural optimizations at the module level. The LLVM optimizations are built into libraries, making it easy for front-ends to use them.

The “"Clang" CFE Internals Manual” [ClangInternals] describes the Clang architecture in detail. Currently Clang consists of 11 libraries and the Clang driver tool.

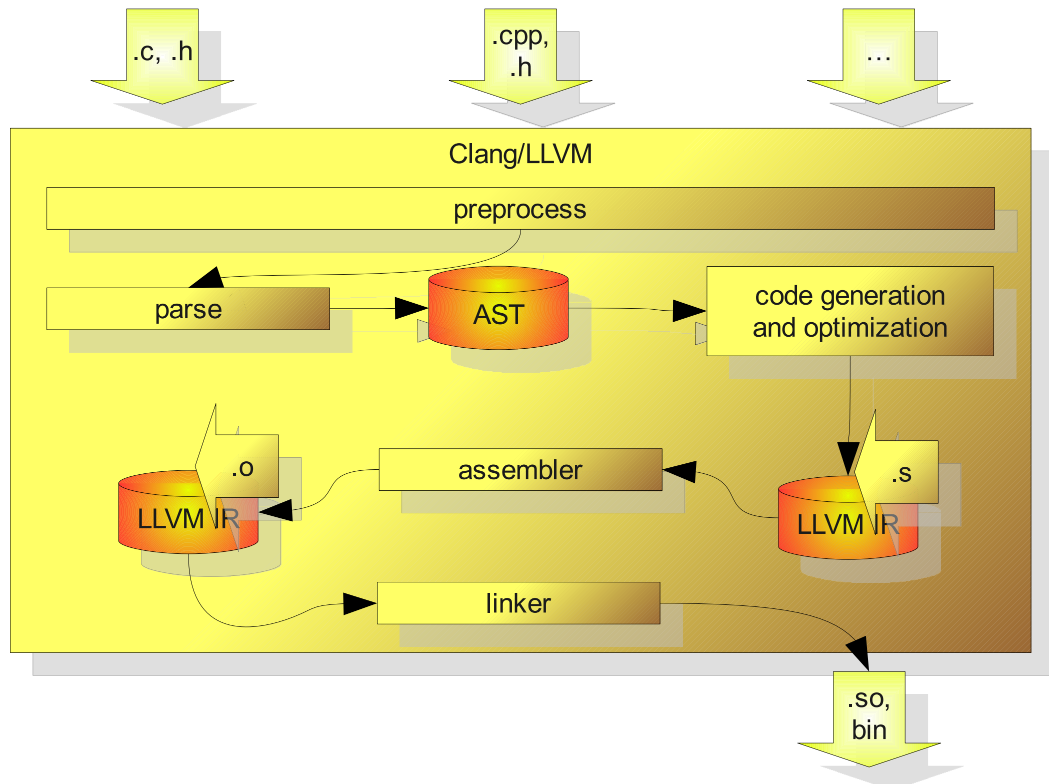
The “Clang Driver Manual” [ClangDriver] describes the purpose of the driver:

> The Clang driver is intended to be a production quality compiler driver providing access to the Clang compiler and tools, with a command line interface which is compatible with the gcc driver.

When invoked to compile C/C++/ObjC code, performs the following steps:

1. Parse command line parameters
2. Invoke Clang preprocessor
3. Invoke Clang compiler
    1. Parse code into AST
    2. Lower the AST into LLVM IR
4. Call the LLVM optimizer, linker and other libraries according to the given parameters

The following illustration shows the the way the frontend operates is very similar to GCC:



Drawing 4: Clang/LLVM compilation

At this point the process steps back into the domain described in the Lattner, Adve paper [LattnerAdve2004]. The border between the Clang frontend and the LLVM backend runs right through the code generation stage. The first step of that stage is to transform the AST into the LLVM IR. Everything that follows is performed by LLVM components. However the entire compile pass is controlled by Clang.

The most notable difference to the GCC backend is that the LLVM IR encompasses the entire compile run time. Because the LLVM IR is available at link time, the linker can perform additional optimizations across object files.

The entire process is more conservative than originally envisioned in the Lattner, Adve paper [LattnerAdve2004]. This is most likely due to Clang's development goal to function as a drop in replacement for the GCC.

According to the paper LLVM would be capable to attach the LLVM IR to the final binaries, allowing on the fly relinking when depending libraries are updated or to apply new optimizations passes, e.g. when the compiler is updated or when new profiling data is available, that suggests that specialized versions of certain functions would benefit runtime. All components to realize this are already in place, all it would need was another Clang driver without the GCC legacy architecture and an operating environment that tightly integrates LLVM.

2. The Makings of a Good Compiler

Before going into detail about the features of Clang/LLVM, it is necessary to define what constitutes a good compiler. Later chapters will then explore these established aspects of compiler quality.

There are two levels at which a compiler's quality has to be measured. Compilers are software and at the same time they produce binary representations of software, that can be interpreted by a virtual machine or directly by a micro controller or processor. So quality measures need to be established for how well the compiler software performs and for how well the compiled software, i.e. binary representations, perform. For the sake of simplicity, the term binary will be used for compiled software within this document.

Micro controllers and processors will be abbreviated CPU for central processing unit.

2.1. Compiler Performance

The significant aspects of compiler performance are usability and speed.

The usability of a compiler depends on the expressiveness of warnings and errors, debugging capabilities, its interfaces to embed it into other software and the availability of supporting tools and the integration into established development tools, e.g. IDEs.

The speed of a compiler depends on its execution time and memory footprint. The latter is especially significant with modern multi core CPUs. Compilation is a process that can easily be split into parallel tasks.

Because the required amount of memory scales linearly with the number of running processes, memory use can severely impact performance. If the system runs out of memory and starts swapping, execution time can easily increase by unacceptable factors.

The author of this document regularly builds OpenOffice.org packages for the FreeBSD operating system and only recently has (i.e. March, 2010) switched to parallel OpenOffice.org builds after upgrading to 8GB RAM from 2GB.

2.2. Binary Performance

The significant aspects of binary performance in respect to the compiler are speed and correctness.

Correctness is the minimum requirement of a binary. It means that the binary, interpreted by a CPU or virtual machine, performs the logic specified in the source code. Considering the availability of *compatible* CPU families for a single platform like i386 or AMD64, all with their own specific bugs a compiler has to circumvent, and the extensive code optimizations performed by a compiler, this is a non-trivial matter.

The speed at which a binary performs the coded logic depends on how well the compiler optimizes for a CPU architecture or a specific CPU family. Optimizing for a specific family has the downturn of potentially reducing the performance with other architecture compatible CPUs, but is a significant requirement for high performance applications like clustered physics simulations. Generally optimizing for a specific CPU family only makes sense, when the user of a binary has access to the source code and can thus recompile it.

2.3. C/C++ Compiler Performance Check List

- ✔ Compile time
- ✔ Maximum memory use
- ✔ Features
- ✔ Feature integration
- ✔ Binary correctness
- ✔ Binary performance

As mentioned in the introduction only compile time and memory use were extensively tested for this report, the remaining aspects are mentioned in passing.

3. Clang/GCC Compile Time Comparisons

This chapter contains compile time measurements from tests performed for this paper[2]. The LLVM home page [LLVM] boldly states on the first page:

> Clang is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles (e.g. about 3x faster than GCC when compiling Objective-C code in a debug configuration), …

The project offers several examples backing these bold claims. However these testaments to the superiority of Clang/LLVM were obviously hand picked to represent the project. This chapter provides numbers based on a real world use scenario, the source code of the FreeBSD operating system.

3.1. ClangBSD Compile Time

ClangBSD is a branch of FreeBSD that aims to integrate Clang/LLVM into the FreeBSD operating system in an effort to break the dependency on GCC.

The following compile time measurements were performed on a FreeBSD/amd64 8.0-STABLE system:

```
FreeBSD mobileKamikaze.norad 8.0-STABLE FreeBSD 8.0-STABLE #0: Mon
Apr  5 12:45:41 CEST 2010
root@mobileKamikaze.norad:/usr/obj/HP6510b-
8/amd64/usr/src/sys/HP6510b-8  amd64
```

The hardware is a HP Compaq 6510b notebook with the following parameters:

- `Intel(R) Core(TM)2 Duo CPU     T7700  @ 2.40GHz (2394.02-MHz K8-class CPU)`

2 In the course of this research more than 50 hours of real compile time were spent.

3.Clang/GCC Compile Time Comparisons

- `8192 MB RAM (2 x 4GB DDR2)`
- `FSB 800MHz`

The following is a list of the used host system compilers.

clang	clang version 1.5 (trunk) Target: x86_64-portbld-freebsd8.0 Thread model: posix
gcc	Using built-in specs. Target: amd64-undermydesk-freebsd Configured with: FreeBSD/amd64 system compiler Thread model: posix gcc version 4.2.1 20070719 [FreeBSD]

Table 1: Used compiler version information

The following charts display measurements for different compiler settings. To understand what these charts mean one has to understand the general layout of the FreeBSD source tree and its bootstrapping process in very broad terms.

Chapter 24 Updating and Upgrading FreeBSD of the FreeBSD 7.3-RELEASE and FreeBSD 8.0-RELEASE Handbook [FbsdHandbook] describes the build process in two steps:

1. `make buildworld`

   This first compiles the new compiler and a few related tools, then uses the new compiler to compile the rest of the new world. The result ends up in `/usr/obj`.

2. `make buildkernel`

   Unlike the older approach, using config(8) and make(1), this uses the *new* compiler residing in `/usr/obj`. This protects you against compiler-kernel mismatches.

I.e. the host system compilers will only be used in the first **buildworld** step. Which compiler is used can be influenced by setting the CC and CXX variables and passing them on to make[3]. **CC** controls the C compiler and **CXX** the C++ compiler used.

The charts list three different compiler settings, **gcc**, **cc** and **clang**, which are abbreviations for different **CC** and **CXX** settings. The following table shows how each setting is mapped to them:

	CC	**CXX**
gcc	gcc	g++
cc	cc	c++
clang	clang	clang++

Table 2: C and C++ compiler table

The **cc** case is special, because it refers to the *default* compiler. On a FreeBSD-8 host system the default compiler is **gcc**:

```
> ll `which cc` `which gcc`
-r-xr-xr-x  2 root  wheel   401K  5 Apr 12:56 /usr/bin/cc*
-r-xr-xr-x  2 root  wheel   401K  5 Apr 12:56 /usr/bin/gcc*
> md5 `which cc` `which gcc`
MD5 (/usr/bin/cc) = 5cf8f8a2031c09ed9d7e9fb05046d245
MD5 (/usr/bin/gcc) = 5cf8f8a2031c09ed9d7e9fb05046d245
```

However after the bootstrap the ClangBSD default compiler **clang** will be used. Thus the build is done by two entirely different compilers.

> This particular case actually caused the first test runs to fail. Depending on whether **gcc** or **clang** are used, the build has to be performed with different system headers and the ClangBSD build infrastructure of the SVN revision 207220, which was used for the first preliminary tests, did not handle the CC=cc case, resulting in a build failure.

3 There are numerous ways of doing this, such as exporting them to the environment, setting them in a configuration file or passing them on to make as parameters.

3.Clang/GCC Compile Time Comparisons

> After a post on the **freebsd-current** mailing list, the author was contacted by ClangBSD developer Roman Divacky, who after a short mail exchange pointed out the responsible section of the build infrastructure and provided a first patch attempt, that tried to *ask the compiler* whether it was actually **clang** instead of guessing it from the values of **CC** and **CXX**.
>
> Though this patch did not work, due to PMake performing all conditional and variable evaluation as a preprocessing step of makefile parsing (see PMake – A Tutorial by Adam de Boor [Boor1988]) and thus before the bootstrapping took place, the idea proved right when it led to a working patch, which circumvented the issue by injecting the check as runtime executed shell code.
>
> The patch was committed to SVN revision 207367 [ClangBSD207333], which was used for the measurements described in this paper.

The **buildworld** and **buildkernel** targets were measured separately, because they represent very different code layouts. The **buildkernel** step compiles the operating system kernel, which has a very clear and simple structure. Most of the code was originally developed for FreeBSD. Even code originating from different projects (called contributed code) is strongly changed to fit in.

The **buildworld** target on the other hand compiles a conglomeration of FreeBSD specific and contributed code from a vast number of different projects. All integrated into a single source tree, that makes up the FreeBSD operating system.

To check for the impact of potentially different file system access patterns, each test was performed in a temporary hard disk (HD) space and on a memory disk (MD).

Further measurements are suffixed with **-j3**. This implies that the make call was performed with the parameter **-j3**. The **make(1)** [MAN1MAKE] manual page states:

```
-j max_jobs
   Specify the maximum number of jobs that make may have running at
   any one time.  Turns compatibility mode off, unless the -B flag
   is also specified.
```

The documentation to the **-B** parameter elaborates that serial execution is the default behaviour:

```
-B Try to be backwards compatible by executing a single shell per
   command and by executing the commands to make the sources of a
   dependency line in sequence.  This is turned on by default unless
   -j is used.
```

The purpose of calling make with the **-j3** parameter is to allow it to perform several independent compilations at once to make use of all available CPU cores. The number 3 was chosen, because the system has 2 cores. The additional process is there to avoid idle time while a process is waiting for I/O operations.

3.1.1. Understanding Charts

The measurements were performed with the command **time -l**, which invokes the commands given as its parameter as a child process and uses the **getrusage(2)** [MAN2GETRUSAGE] system call to collect information about the performed operations.

Time measurements are given in seconds, memory measurements in kb (as in 2^{10} bytes):

- **real**

  The real time that passed from command execution to termination. On a single core system this amounts to *user* + *sys* + the time lost to other processes (which can never be completely avoided on a multi tasking system) or waiting for I/O.

- **user**

  The CPU time spent in the user context. Note that the CPU time is counted for each core, so this can amount to more than the real time if more than one CPU is involved.

- **sys**

  The CPU time spent in the system context. This is time spent in the kernel for things like locking and device access.

- **maxrss**

  The maximum resident set size is the highest amount of memory that was in use during the entire run time. It is a good indicator for how much system memory is required. This value refers to the amount of memory really used, not by the amount of memory allocated.

3.1.2. ClangBSD SVN r207367 buildworld

The following chart shows the time measurements for the **buildworld** target:

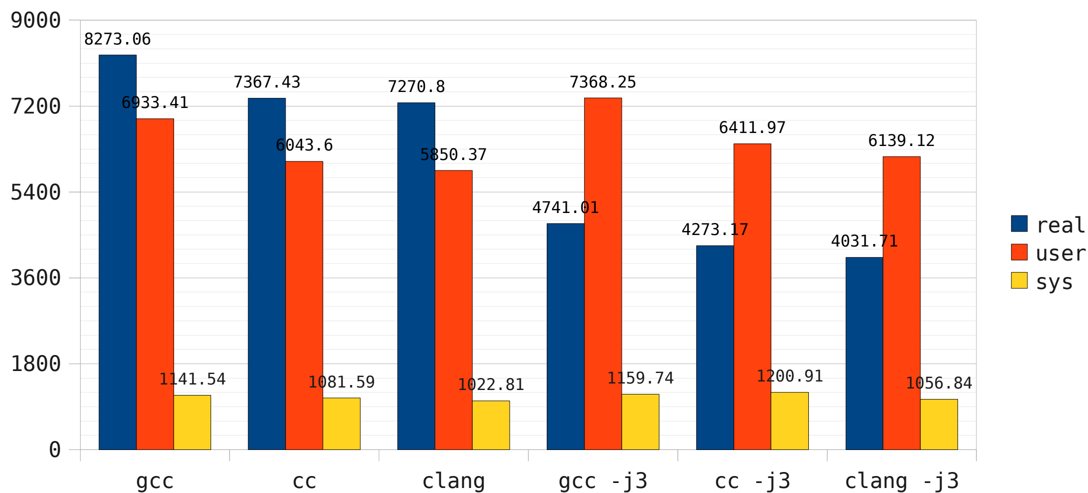


Illustration 3: **buildworld** times on a HD

As can be seen in serial execution mode **clang** beats **gcc** by more than 16 minutes, which means that it took **gcc** 13.78% more time to complete. With the **cc** setup compilation is nearly as fast, which can probably be attributed to **cc** mostly being **clang**.

In parallel build mode **clang** beats **gcc** by 11 minutes, which means that **gcc** took even 17.59% longer to complete.

The next chart shows the same measurements performed on a memory disk:

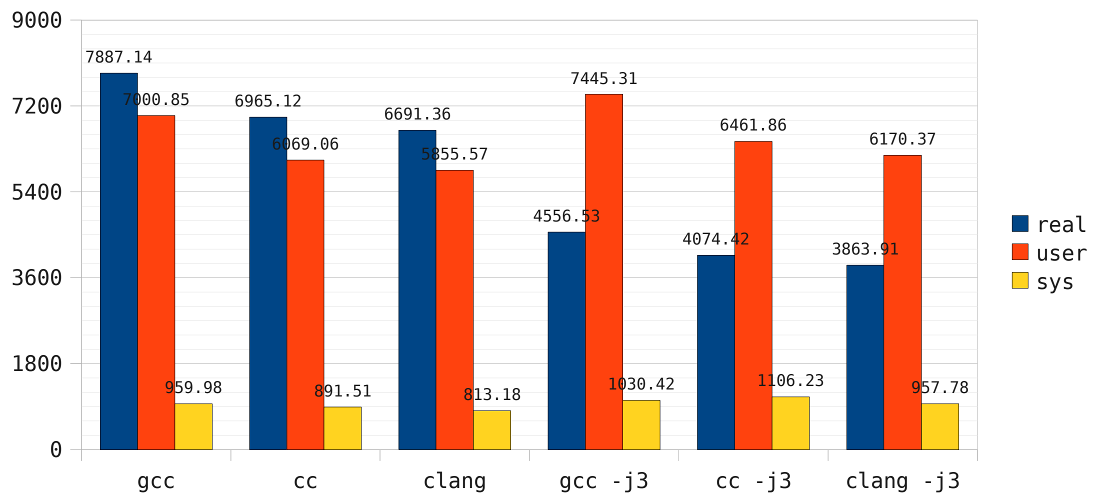


Illustration 4: **buildworld** times on a MD

In serial mode the **gcc** compiler was 4.66% faster than on a hard disk, which amounts to over 6 minutes. However this is still slower than **clang** or **cc** on a hard disk. On a memory disk **clang** was 7.97% faster than on a hard disk, more than 9 minutes faster than it was on a hard disk. This means that on a memory disk **gcc** took 17.87% more time to complete than **clang**.

In parallel mode **clang** was again more than 11 minutes faster, which means that **gcc** took 17.93% longer.

In parallel mode **clang** was only 4.16% faster than on a hard disk, similar to **gcc**, which was 3.89% faster. This means the effect of running on a memory disk is much smaller than in serial mode, which confirms the assumption that performing parallel builds using more parallel processes than processing units, helps circumventing I/O delays.

The next chart shows the maximum memory used during each build run:

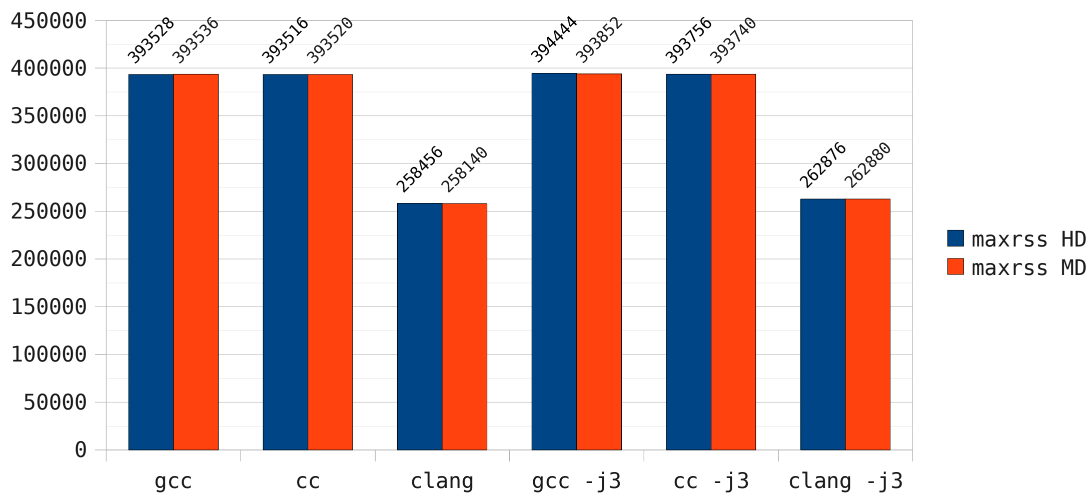


Illustration 5: **buildworld** memory consumption

This chart confirms what one would naturally expect. The I/O device (i.e. hard disk or memory disk) does not influence memory use.

Unlike with the build times the **cc** setup follows the **gcc** compiler setup in maximum memory use. This implies that the bootstrapping process requires the most memory. The most significant part of the bootstrapping process is probably the compilation of the compilers **clang** and **gcc**, both are good suspects to attribute the maximum memory use to.

In all cases the maximum memory use of **clang** was around 65% of **gcc**'s memory use. In other words, **gcc** required up to 384mb of memory to do the same job **clang** did with 252mb.

The implications are not immediately obvious, because both numbers do not look very high on modern systems (e.g. the notebook the test ran on has 8gb of RAM at its disposal). However maxrss measures the largest process in the entire build run, i.e. in the **-j3** case the build process might have used up to 3 times the memory, so 1152mb for **gcc** and 757mb for **clang**. The implication is that a machine with 1gb of RAM a **gcc** build should be limited to 2 parallel processes, whereas it would provide sufficient resources for 3 parallel processes using **clang**.

The implications become even more dire, when considering that there are much more memory demanding build processes around. The following notice is printed when attempting to build OpenOffice.org 3.2.0_2 with the FreeBSD Ports Tree[4] [FPortsOOo3]:

```
NOTICE:

To build OOo, you should have a lot
of free diskspace (~ 11GB) and memory (~ 2GB).
```

4 The software package building infrastructure of FreeBSD.

3.1.3. ClangBSD SVN r207367 buildkernel

The **buildkernel** stage is less interesting than the **buildworld** stage insofar that the kernel sources neither reach the size nor the diversity of the world sources. However it allows the comparison of host system compilers to the bootstrapped compilers.

The first chart shows the regular compilation performance with the bootstrapped compilers.

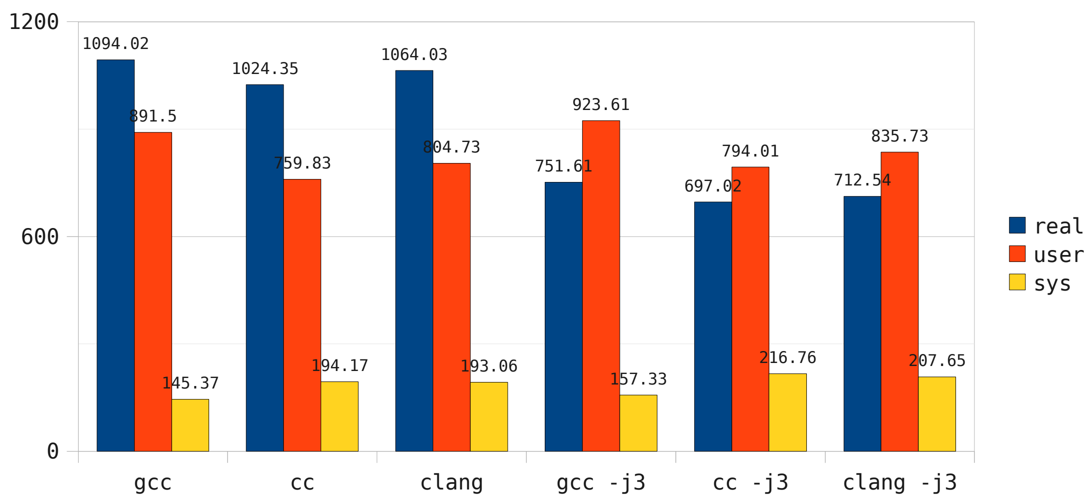


Illustration 6: **buildkernel** on a HD

As the chart shows **clang** only gains a 2.82% advantage over **gcc** in serial compilation mode. In parallel mode the distance grows to 5.48%, still significantly below the 17.59% when performing a **buildworld**. The values of **cc** and **clang** are very close, this can be attributed to **cc** being an alias for **clang** in this case. Differences have to be attributed to different treatment by the build system and the general load every multi-tasking system is inevitable subject to.

The following chart depicts the same measurements performed with the host compilers.

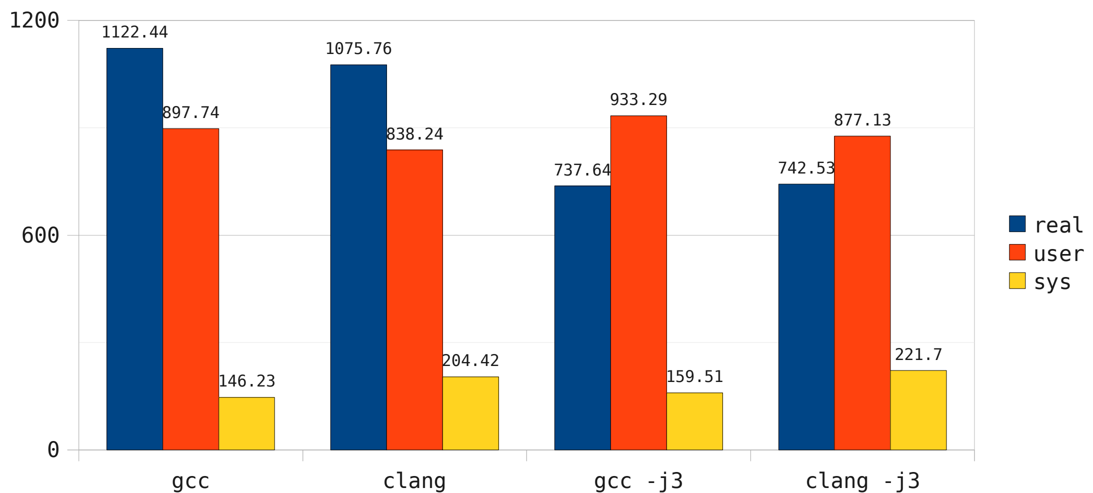


Illustration 7: **buildkernel** with host compilers on a HD

With the notable exception of **gcc -j3** the host system compilers perform slower. A probable explanation is that the bootstrapped compilers were linked against a recent FreeBSD 9 development environment (i.e. ClangBSD), whereas the host system was a stable FreeBSD release engineering branch.

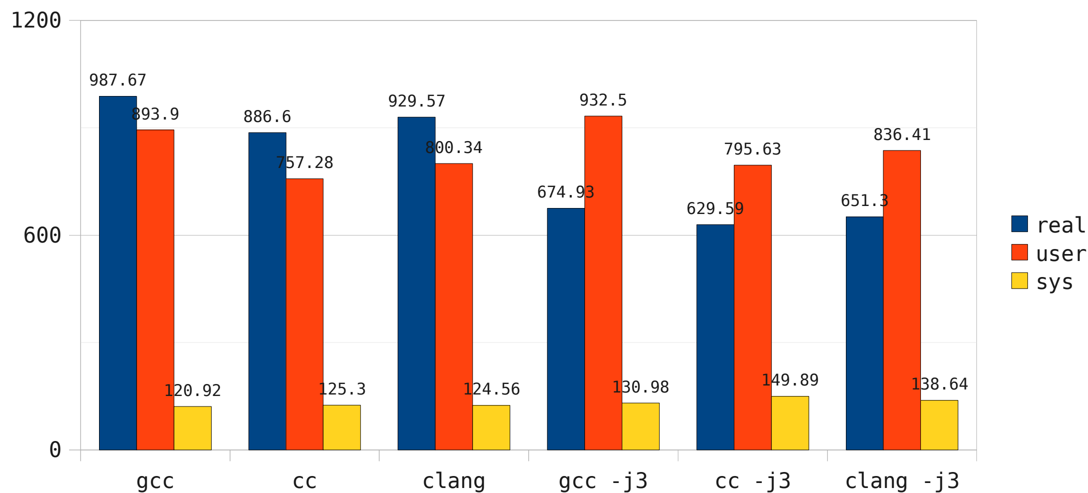


Illustration 8: **buildkernel** on a MD

On a memory disk **gcc** built 9.73% faster than on the hard disk. Again **cc** gains a little more with 13.44% performs and **clang** gains 12.64%. The parallel **gcc -j3** build actually gains slightly more from the move to to a memory disk with 10.21%, **cc** with 9.03% and **clang** with 8.59% don't gain quite as much, but still a lot more than during **buildworld**. This means that the parallel build process does not compensate hard disk I/O delays quite as well as during a **builworld**.

The next chart also shows how the host compilers perform on a memory disk.

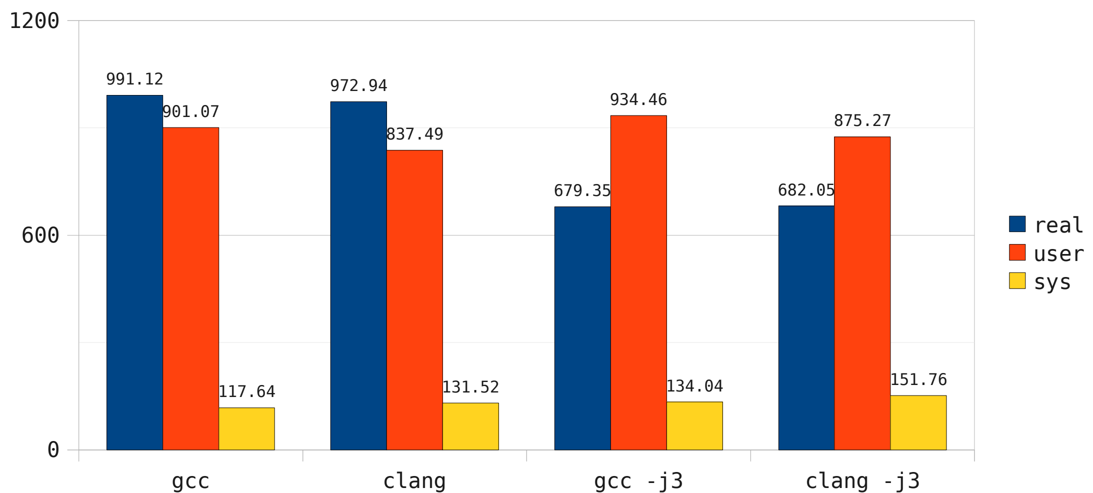


Illustration 9: **buildkernel** with host compilers on a MD

With 11.70% the host system **gcc** performance improves similar to the bootstrapped **gcc**. However it's only 0.35% slower than the build system **gcc** on a memory disk. Unlike on a hard disk, where the **gcc -j3** host compiler test case was actually faster than the bootstrapped compiler, the parallel **gcc -j3** build on a memory disk, performed 0.65% slower. For **clang** the disadvantage of the host system compiler, compared to the bootstrapped compiler is less significant with 4.67%. The parallel **clang -j3** build had a similar disadvantage of 4.72%.

The last chart of this chapter shows the maximum memory consumption during buildkernel.

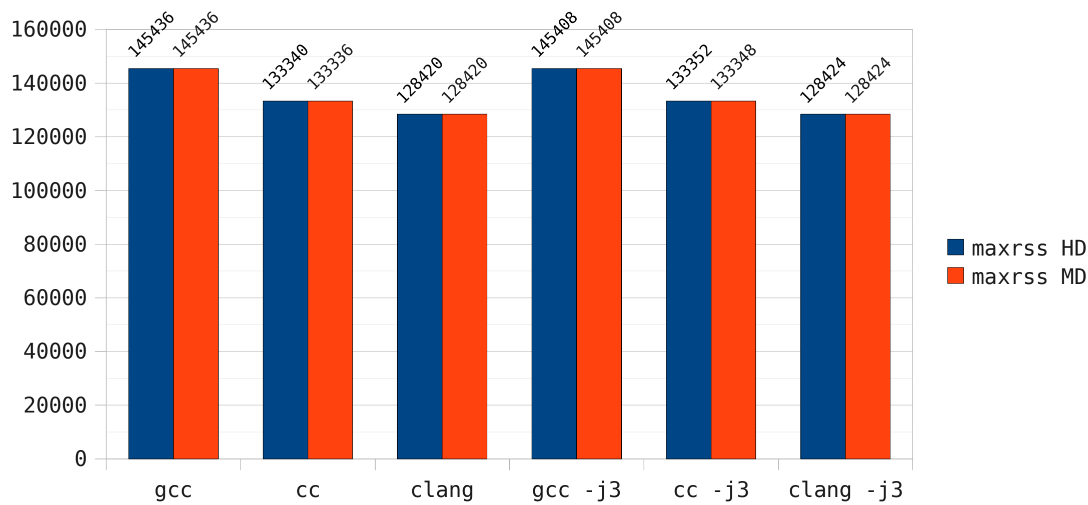


Illustration 10: **buildkernel** memory consumption

With 11.7% less memory than **gcc**, **clang** does not nearly gain as much of an advantage as during the buildworld tests. The complexity of kernel code probably does not suffice to unveil memory issues.

3.2. Reasons for Differences in Compile Time

These, partly rather significant, compile time differences between Clang/LLVM and GCC, might be attributed to several different features. The reasons laid out in this subsection should be considered speculations until further investigation confirms or denounces them.

The “Clang – Features and Goals” [ClangFeatures] manual states that the clang frontend's preprocessor and AST building is significantly faster:

> In our measurements, we find that clang's preprocessor is consistently 40% faster than GCCs, and the parser + AST builder is ~4x faster than GCC's.

Apart from that, looking at the architecture layouts of Clang and GCC laid out in section 1.4, Compiler Architecture (page 8) and 1.5, Introduction to Clang/LLVM (page 11) show that the internal code representation of GCC goes through several conversions that are not necessary in the LLVM backend, because code is always kept in the LLVM IR there.

4. Clang/GCC Binary Performance Measurements

Binary performance is not the focus of this report. So this chapter just provides a starting point for those willing to research. The source used in this chapter is Phoronix [Phoronix], an online magazine that regularly publishes articles with performance measurements, normally produced with the Phoronix Test Suite [PhoronixTS].

To pre-empt the conclusion of the Phoronix article, ioquake3 SVN r1784 performance built with Clang or GCC does not differ significantly on the author's computer system. Ioquake3 is a project that maintains a branch of the id Tech 3 engine [IOQ3], which is the basis for several standalone projects such as OpenArena [OpenArena] or Urban Terror [UT] and can still be used to play Quake 3 Arena [Q3A]. The author of this report currently maintains the FreeBSD ports of ioquake3 [FPortsIoq3][FPortsIoq3dev].

On its final page, the article “Benchmarking LLVM & Clang Against GCC 4.5” written by Michael Larabel and published on 21st April, 2010 [Larabel2010] concludes:

4.Clang/GCC Binary Performance Measurements

> While using LLVM is faster at building code than GCC (except for the ImageMagick application), in most instances the GCC 4.5 built binaries had performed better than LLVM-GCC or Clang. Clang did deliver a surprising lead over GCC 4.5 and LLVM-GCC with the Apache benchmark where the Clang-built Apache managed to handle 9% more requests per second. There was also significant benefits for LLVM-GCC and Clang with the BYTE Unix Benchmark running the Dhrystone 2 test, but in the rest of the tests the performance was either close to that of GCC or well behind. In some tests, the performance of the Clang generated binaries was simply awful.

This statement illustrates both the potential of Clang as a C/C++ compiler as well as its lack of maturity. Maturity in this context has to be defined as a two-way relationship. Not only has the GCC had several decades more time to mature, but also the software compiled with it had the time to be optimized for performing well with GCC.

However, the Clang project emits new success messages at an astounding frequency. On 20th May, 2010 Doug Gregor wrote on the LLVM Project Blog that “Clang++ Builds Boost!” [Gregor201005]:

> This morning, Clang++ had its first fully-successful Boost regression test run, passing every applicable C++ test on the Boost release branch [*]. According to today's results, Clang is successfully compiling more of Boost than other, established compilers for which Boost has historically been tailored (through various workarounds and configuration switches). In fact, Clang's compiler configuration in Boost is completely free of any of Boost's C++98/03 defect macros.

Boost is a C++ library collection in wide spread use, the project homepage [Boost] provides more information.

What this signifies is that Clang currently matures very fast.

5. Conclusion

All in all, Clang/LLVM is not yet usable as a drop in replacement for the GNU Compiler Collection, however its current progress suggests that it soon will be.

In other areas, that were not really part of this report, it already outperforms GCC. The modular library architecture of the underlying LLVM allows a tight integration into IDEs, runtime optimizers and similar tools. Apart from the project's financial backer Apple, other projects like Mono also integrate parts of LLVM [MonoLLVM]:

> Mono from SVN is now able to use LLVM as a backend for code generation in addition to Mono's built-in JIT compiler.
>
> This allows Mono to benefit from all of the compiler optimizations done in LLVM. For example the SciMark score goes from 482 to 610.

While not yet able to fit into a large production environment like a package building cluster, due to the sheer amount of C++ code that does not yet compile with Clang, Clang/LLVM is already a valuable developer tool. Its widely advertised, clear error and warning messages are eye opening for people who are used to gleam meaning from the obscure error output produced by GCC[5], tools like KLEE provide valuable information to developers, as Daniel Dunbar explains on the LLVM Project Blog [Dunbar201004]:

> If you aren't familiar with it, KLEE is a tool for symbolic execution of LLVM code. It is way too complicated to explain here, but for the purposes of this example all you need to know is that it tries to explore all possible paths through a program.

5 This statement is based on personal experience.

5.Conclusion

From an architecture standpoint LLVM rises and falls with the quality of its intermediate representation. The intended long life cycle of the LLVM IR, encompassing the entire life time of the produced binary, instead of just the short lived compile time forces developers to always stay compatible to older versions. Whether this is a burden or not remains to be seen.

That Clang/LLVM falls back into the realms of obscurity seems rather unlikely at this point. Enthusiasm for the project is wide spread and a steady stream of success messages, combined with Apple providing financial backing point towards an exciting future for the project.

6. Appendix

Bibliography

[AppleInsider] AppleInsider Staff, Apple Insider News and Analysis, 2010, http://www.appleinsider.com/

[Boor1988] Adam de Boor, PMake – A Tutorial, 1988, Berkeley Softworks

[Boost] Beman Dawes, David Abrahams, Rene Rivera, Boost C++ Libraries, 2010, http://www.boost.org/

[ClangBSD207333] Roman Divacky, ClangBSD SVN Revision 207333, 2010, http://svn.freebsd.org/viewvc/base?view=revision&revision=207333

[ClangDriver] The Clang/LLVM Project, Clang Driver Manual, 2010, The LLVM Project

[ClangFeatures] The Clang/LLVM Project, Clang - Features and Goals, 2010, The LLVM Project

[ClangInternals] The Clang/LLVM Project, "Clang" CFE Internals Manual, 2010, The LLVM Project

[Divacky201005] Roman Divacky, ClangBSD branch needs testing before the import to HEAD, 2010, http://docs.freebsd.org/cgi/mid.cgi?20100529130240.GA99732

[Dunbar201004] Daniel Dunbar, What's wrong with this code?, 2010, http://blog.llvm.org/2010/04/whats-wrong-with-this-code.html

[FbsdAn2009] Ed Schouten, Roman Divacky, Brooks Davis, Pawel Worach, FreeBSD Status Reports January - March, 2009, 2009, http://docs.freebsd.org/cgi/mid.cgi?20090509191339.GE40521

[FbsdHandbook] The FreeBSD Documentation Project, FreeBSD Handbook, 2010, The FreeBSD Project

[FPortsIoq3] Dominic Fandrey, games/ioquake3, 2010, http://www.freshports.org/games/ioquake3

[FPortsIoq3dev] Dominic Fandrey, games/ioquake3-devel, 2010, http://www.freshports.org/games/ioquake3-devel

[FPortsOOo3] openoffice@FreeBSD.org, editors/openoffice.org-3, 2010, http://www.freshports.org/editors/openoffice.org-3

[GnuGcc] The GCC Team, GCC, the GNU Compiler Collection, 2010, http://gcc.gnu.org/

[GnuGpl] Free Software Foundation, GNU General Public License, 2007, http://www.gnu.org/licenses/gpl.html

[Gregor201002] Doug Gregor, Clang Successfully Self-Hosts!, 2010, http://blog.llvm.org/2010/02/clang-successfully-self-hosts.html

[Gregor201005] Doug Gregor, Clang++ Builds Boost!, 2010, http://blog.llvm.org/2010/05/clang-builds-boost.html

[IOQ3] Zachary Slater, ioquake3, 2010, http://ioquake3.org/

[Larabel2010] Michael Larabel, Benchmarking LLVM & Clang Against GCC 4.5, 2010, http://www.phoronix.com/scan.php?page=article&item=gcc_llvm_clang

[LattnerAdve2004] Chris Lattner, Vikram Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, 2004

[LLVM] Chris Lattner, The LLVM Compiler Infrastructure Project, 2010, http://llvm.org/

[LlvmPubs] Chris Lattner, LLVM Related Publications, 2010, http://www.llvm.org/pubs/

[MAN1MAKE] The FreeBSD Project, make(1) – maintain program dependencies, 2008, The FreeBSD Project

[MAN2GETRUSAGE] The FreeBSD Project, getrusage(2) – get information about resource utilization, 2010, The FreeBSD Project

[McLean2008] Prince McLean, Apple's other open secret: the LLVM Compiler, 2008, http://www.appleinsider.com/articles/08/06/20/apples_other_open_secret_the_llvm_complier.html

[MonoLLVM] The Mono Project, Mono LLVM, 2010, http://www.mono-project.com/Mono_LLVM

[OpenArena] OA Team, OpenArena, 2010, http://openarena.ws/

6.Appendix

[Phoronix] Phoronix Media, Phoronix – Linux Hardware Reviews, Benchmarking, & Gaming, 2010, http://www.phoronix.com/

[PhoronixTS] Phoronix Media, Phoronix Test Suite - Linux Testing & Benchmarking Platform, Automated Testing Framework, Open-Source Benchmarking, 2010, http://www.phoronix-test-suite.com/

[Q3A] Id Software LLC, id Software, 2010, http://www.quake3arena.com/

[UT] FrozenSand, Urban Terror, 2010, http://www.urbanterror.info

[WBGccArch] Alexey Smirnov, GNU C Compiler Internals/GNU C Compiler Architecture, 2010, http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture

[WikiBooks] Various Authors, WikiBooks - Open books for an open world, 2010, http://en.wikibooks.org/

[Wirth1996] Niklaus Wirth, Compiler Construction, 1996, ISBN 0-201-40353-6