# Build Linux kernel with Open64

Qing Zhu[1], Tao Wang[2], Gang Yu[1], Kun Ling[2], Jian-xin Lai[1]

[1]Unix System Lab, Hewlett-Packard Company

[2] State Key Laboratory of Computer Architecture, Institute of Computing Technology

[1]{qing.zhu2, yu.gang, jianxin.lai}@hp.com, [2]{wangtao2010, lingkun}@ict.ac.cn

## Abstract

Open64 is an open source, optimizing compiler. The goal of the compiler is to be an alternative back-end of GCC. To improve Open64's code quality, enhance GCC-compatibility and attract more users to the Open64 community, it is important to build significant applications like Linux kernel with Open64. This paper describes the work of building and running up Linux kernel with Open64. As an experience report, the general build, triage approach and tackling method for commonly issues on building kernel with non-gnu are discussed. Now, the officially maintained open64.net trunk on x8664 target and the Loongcc[1] branch on MIPS64 target have all successfully built up and run the Linux kernel.

***Keywords*** Open64, Linux kernel, Loongcc, X86-64, MIPS-64, QEMU, BOOT

## 1. Introduction

Open64 is an open source, optimizing compiler. It supports Fortran 90/95 and C/C++, as well as the shared memory programming model OpenMP. The C and C++ front ends are based on GNU C compiler; the Fortran90/95 front end is the SGI Pro64 (Cray) Fortran front end. The GNU Compiler Collection or GCC (formerly GNU C Compiler) is an open source compiler for C/C++, Fortran77, Ada and Java. Compared to GCC, Open64 features advanced interprocedural optimizations, loop nest optimizations, global scalar optimizations, and code generation with advanced global register allocation and software pipelining, and designed to be a high performance compiler. However, GCC is the de-facto industry standard compiler, it is much more popular than Open64, and most of the GNU and open source software use GCC as the default compiler. To improve the quality and GCC-compatibility of Open64, one urgent task for Open64 is to build and run well-known prominent applications.

Linux kernel[5] is an operating system kernel used by the Linux family of Unix-like operating systems. It is one of the most prominent and complex applications in open source software. It's well-known the Linux kernel relies on the GCC-extensions heavily. To our knowledge, besides GCC, only Intel C/C++ Compiler can build an earlier kernel version[10, 11] around 2006. Recently, Clang [12] can compile and boot up the Linux kernel with limited functionality.

This paper describes the work of building and running Linux kernel as the target application for Open64. We have at least three goals:

- Improve Open64 code quality and GCC-compatibility by fixing the bugs found in compiling and running the kernel.
- Attract more potential users to Open64 community by showing Open64's quality and GNU-compatibility. We choose X8664 and mips64 platform as kernel building targets, since these two architectures are principal and popular in IT industry, proving the success on these two platforms will impress more people on Open64.
- Provide a good research platform to improve the kernel's performance by advanced optimizations. Especially, to enable IPA/IPO (Inter-Procedure Analysis and Optimization) on kernel build, show how much performance improvement that IPA/IPO will give us on standard system software like operating system kernel.

In this paper, we make a summary of general practices in kernel building process on different architectures. These approaches and experience may help other targets to conduct kernel building and build other Open64 support platforms effectively. The paper is intended to contribute the general approaches and experiences on building kernel-like system software using a non-GNU compiler like Open64. It is organized as follows: section 2 will introduce the steps to build an boot Linux kernel, we will show the detailed triaging/debugging environment information and our current achievement. Then, section 3 will discuss the issues on build and run the kernel using Open64. We also discuss our understandings and tackling methods for these issues. Finally, we make the summary on our work in section 4 and also discuss possible future work continually enhancing Open64 on system softwares.

## 2. Steps to Build and Boot Linux Kernel with Open64

This section describes the general steps to build and boot kernel, including how to configure, Open64 option control, boot tool, and the general triage methods of build and boot issues.

### 2.1 Configure

The configuration method is as the usual way that we configure for kernel. There are a lot of configuration methods: menuconfig(menu based), xconfig(QT based), gconfig(GTK based),etc. No matter which method to choose, the config info is finally written to ".config" file, we can easily change the value to "y"(built in to kernel),"m"(built as module) or do not set it.

Since there are so many configuration values, it is hard for us to make sure on each value, and also we don't want to miss the chances to compile more modules and drivers using Open64. We choose the default configurations for X8664 and MIPS, i.e, "make x86_64_defconfig" for X8664 and malta motherboard default config for MIPS.

### 2.2 Build

Build is also quite simple, just types "make -jN"(Where N is the number of jobs allow at one time). But before actually build, we have to solve the option control issue.

For the historical reasons, Linux kernel is quite gcc-oriented. In order to successfully build up the kernel by non-gnu compilers, it is necessary to collect the kernel used options and filter out those specific to gcc or extremely unportable. At the same time it is also necessary to add some tuning options for kernel-like big systems code bases.

Dongyuan's team from Tsinghua University had already made some initial efforts on the options selecting and pushing by a script. This work was committed at 2010 around for kernel 2.6.27 on IA32 target. Although a little out-dated, it still provides us an initial solution to control the options and filter out those undesired warnings or fake errors.

For general approach, the options are divided into 3 categories:

1) Non-gcc options that need to be added for kernel building: since the kernel is quite a big code-base and intensively using some skilled (maybe quite gnu oriented) and matured coding patterns. It needs some special treats for kernel building. Luckily, due to Open64's good compatibility for various code-bases and high gcc compatible design goals, not too many options have to be pushed. Only 4 options added to the build. They are "-OPT:swap=off", "-OPT:olimit=0", "-WOPT:warn_unit=0" and "-Wno-used". Especially, "-OPT:olimit=0" is pushed because the source code size is somehow exceeding the Open64's default optimizing capacities. We enable this with the risk of resources exhausting, but luckily on the latest machines with big memories, we successfully build up kernel without compiler internal resource errors.
2) Options that are gcc specific, not meaningful on non-gnu compilers or extra undesired warnings: We choose to ignore these options. Gcc special options are mainly stack operations and checks, special x86 handling, compilation controls. Due to different stack usage and x86 porting, we disable all these on Open64 building. Gcc's "-Wall" warning is frontend supported, however, it makes the compilation too much undesired warnings, we also choose to disable it.
3) Options that are supported and meaningful to non-gnu compilers: we choose to accept these options and direct the compiler do kernel hackers' intensions.

On the Loongcc's building effort, we extend the build script to enable *Option Tracking and Complete Failure Reporting*. Option Tracking records the compiling command line once Open64 fails to compile a file. Meanwhile, Complete Failure Reporting takes measures to make Loongcc continue to compile the next file, so that all files can be built once by a single *m*ake. The critical mechanism of Complete Failure Reporting is based on a well-known fact that Gcc and Open64 are ABI-compatible, their object files can be linked seamlessly into one executable. Thus, once Loongcc fails to compile a file, Gcc will be invoked to compile the same file to get corresponding object. In this way, no changes need to be made to Linux kernel original Building system, with which we are unfamiliar. Finally, the report will be generated and inform us how many files have failed with specific command line options. This report not only provides an overview of the compiling process, but also make it possible to fix multiple bugs at the same time, thus quicken the pace of bug-fixing with team-work's strength.

### 2.3 Boot

We choose simulator QEMU [3] to boot the kernel. QEMU is a processor emulator that relies on dynamic binary translation to achieve a reasonable speed while being easy to port on new host CPU architectures. It can save and restore the state of the virtual machine with all programs running. It supports the emulation of various architectures, including IA-32 (x86) PCs, x86-64 PCs, MIPS R4000, etc. QEMU has a built in gdb, which is convenient for debugging. Choosing virtual machine(QEMU) instead of real machine has following advantages:

1) Low cost on access and deployment. If we boot kernel on real machines, we need to reboot the machine, and once the kernel corrupts, it's inconvenient to get the failed log and hard to find the root cause. Open64 built kernel must have a lot of problems while boot from the beginning, it is more wise to choose the virtual machines.
2) QEMU provides real machine simulations
3) Full debug support, QEMU supports snap-shots of boot process, stack dump when fail, and remote debug support.

For X8664 target, we use "qemu-system-x86_64" command to invoke QEMU while using "qemu-system-mips64el" for MIPS64.

### 2.4 Triage/debug method

When kernel fails at build or runtime, we have to figure out a common method to triage and debug. Debugging kernel runtime problems is a black art. It is well-known that Kernel debugging is hard. Besides the complexity of operating systems, unlike user-level programming, establishing a debugging environment isn't as simple as point-and-click. This paper introduces an environment to debug kernel and some debugging techniques. We also designed and implemented tiny tools for automatic triaging both build and runtime failures. These tools help us to find the root cause quickly.

#### 2.4.1 Build issues

Build failures are often easy to triage. With the help of *Option Tracking and Complete Failure Reporting*(introduced at section 2.2), this becomes even easier for Loongcc. The general steps are as follows:

1) Build the kernel and save log file
2) Grep error message from log file and gets the corresponding build command. While build, kernel will save the build command of each object to a file at the same directory of the object file, and named as ".*.cmd".
3) Use Delta [4] tool to generate small case. Delta is a tool which assists engineers in minimizing "interesting" files. Usually "interesting" means a file causing a particular error as input to a program. When Open64 fails to compile one file, we can write a script which include the build command and a grep for the error message, then let delta call the script. Delta will cut the input file if script return 0 until it cannot be minimized any more. For debugging or bug-reporting purposes we would like to find the minimized file, which is very helpful to triage the root cause due to its small scale.
4) Follow normal steps of debugging Open64 to find the root cause with the minimized case.

#### 2.4.2 Runtime issues

***Debugging environment*** Figure 1 gives a high-level overview of kernel debugging environment, it uses two machines, Machine1 use QEMU simulator to load kernel image and invoke gdbserver to listen gdb connection on specified tcp port(default port is 1234); Machine2 use emacs & gdb attached to the remote gdbserver on Machine1 to debug kernel source. Surely, machine1 and machine2 can be one physical machine while the logical relationships between QEMU and GDB client is still the same. Running QEMU with "-S -s" option will freeze CPU at startup status until the gdbserver is attached by a gdb thread from the previously specified tcp port. Then, enter 'c' to start the execution of Linux kernel.

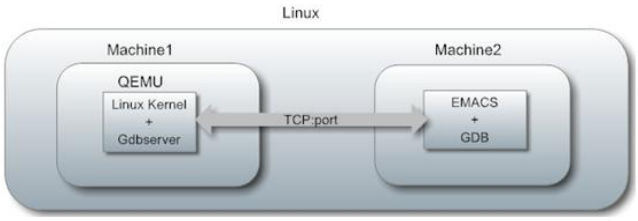



**Figure 1.** Kernel runtime debugging environment

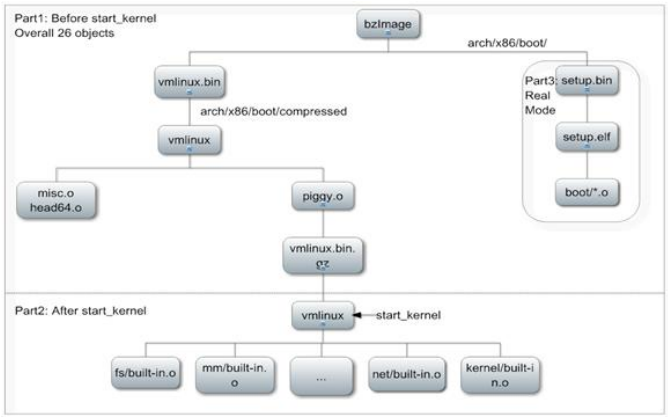



**Figure 2.** Kernel image construction on X8664

***Debugging Techniques*** According to kernel source structure and boot sequence(see Figure 2 for X8664 kernel image construction), this paper divides kernel runtime problems into two categories according booting stages:

1) Failures before call "start_kernel"
2) Failures after call "start_kernel"

There are 3 reasons:

1) "start_kernel" is the first break point that can be set under gdb
2) The runtime address of code running in real mode(Figure 2, Part3) is unknown, it is determined by the design of boot loader. So we can not set breakpoints even at fixed physical address. As to MIPS64, there is only one file writing in assembly code. And before start_kernel, only its object counterpart would be executed.
3) The number of source files and code scale is small in Part1 (only 26 objects for x8664 and a single object for MIPS64), so it is easy to debug even without gdb.

*Triage and debug failures before "start_kernel"* For x8664, there are 26 objects for Part1, in which 23 objects for real mode at dir (arch/x86/boot) and 3 objects (arch/x86/boot/compressed) for decompress kernel. When kernel fails before function "start_kernel", generally follow below steps to triage:

1) Prepare GCC built kernel object files, since gcc and Open64 use the same ABI calling conventions, their objects can be mixed linked
2) Use following pseudo-algorithm to find the buggy objects, the buggy objects = X - Triage(X), where X is the full list of kernel objects. Triage(X) will return the correct objects built by Open64, it is a recursive process. We implemented this algorithm in a tool naming *BuggerInspector*. It can also triage all the runtime buggy objects in one execution.

```
Safe_objects  Triage(object_list X)
{
  Safe_objects Y = NULL;
  while ( X is not empty )
  do{
       divide X to X1 X2;
       build X1 with Open64;
       build X2 with GCC;
       mix link X1 X2 and rebuild kernel;
       if ( kernel boot ok) {
         //X1 is safe;
         X = X - X1;
         Y = Y + X1;
       }
       else{
         if ( X2 is empty )
           return NULL;
         else {
           Y = Y + Triage(X1);
           X = X - X1;
         }
       }
  }
  return Y;
}
```

3) Choose the smallest object file as debug focus first, sometimes there are a lot of erroneous objects, maybe they are caused by the same reason. To reduce workload, we choose the smallest object first.
   i) Build that object file with Open64, others with gcc, read the source code and add "print" to get the value of variables we interests and to determine the location of errors.
   ii) Compare the different behavior with GCC-built kernel. For ex, if some key values are different, we know an error may occur.
   iii) Compare assembler files built from gcc and opencc is also a straight forward way.

For MIPS64, there is one single object whose source file is head.S at dir (arch/mips/kernel), so if kernel fails before "start_kernel", the broken file is determined(head.S).

*Triage and debug failures after "start_kernel"* For kernel failures after "start_kernel", break points can be set, and return to the normal "debugging world", except that the code scale becomes larger and more complex. The general method to debug is as follows:

1) Find the buggy file or function. We use the same algorithm listed above.
2) Debug Open64 built and gcc built kernel at the same time, compare the behavior; add printk to source code, which will print the message or values we wanted to the console; read Oops info, it is the kernel message outputted when kernel occurs fatal error. It mainly include the error overview, loaded module, register info and backtrace info, these will help a lot in find the place of error.
3) Generate reproducible small case.
4) Analyze the root reason and make fix or workaround

| Bug Type | Bug number |
|---|---|
| Common bugs | 6 |
| X8664 specific | 30 |
| MIPS specific | 44 |

**Table 1.** Bug statistical data based on target

| Bug Type | Loongcc bugs | Open64 bugs |
|---|---|---|
| Special incompatible Options | 3 | 4 |
| The incompatibility to gcc | 6 | 5 |
| GNU C extension support | 29 | 6 |
| Other bugs | 12 | 21 |

**Table 2.** Bug statistical data based on bug type

# 3. Discussion on the problems found during build/boot

To successfully build and boot Linux kernel on either X8664 or MIPS, we have to fix all issues found during build and runtime.

In our endeavor, about 36 bugs were found in open64 and 50 bugs in Loongson 3B. We divide the bugs in to 3 main categories:

1) Common bugs exists on both Open64 and Loongson
2) X8664 specific issues
3) MIPS specific issues

Table 1 shows the statistical information of the bugs found in Loongcc and Open64 when build and boot Linux kernel. There are 6 common bugs in both Open64 and Loongcc, which mainly follows to below aspects:

1) The O0 CFG build-up and clean problem
2) The O0/O1 inline invoking problem
3) ASM_INPUT copy-propagation problem
4) Support for GCC extension Variable-Length array in Structure
5) The GNU ASM alias issue

There are about 36 Open64 bugs, in which 30 bugs are X8664 specific, Figure 2 shows there are 4 bugs related with options unsupported by Open64; 5 bugs are incompatible with gcc, such as Open64 didn't do code clean up and inline at opt-level 0; 6 bugs are about GNU C extension support, such as embedded assembly, variable-length array; The remaining 21 bugs are found in other fields, such as front-end, wopt and cg.

There are about 50 Loongcc bugs, in which 44 are MIPS specific. As Linux kernel is closely Co-designed with Gcc, we note that more than half of them are related to target-specific GNU C language extensions[13], like MIPS embedded assembly language support. There are also several Loongcc defects exposed. Among them, there are only 4 original Open64 defects, like WGEN GNU alias attribute, WGEN GNU VLS, WOPT unsafe strength reduction transformation, Inline empty variable argument functions elimination, and the remaining bugs are more or less associating with the Loongcc re-targeting, like position-dependent code generation support, N64 ABI support, Whirl2ops whirl node expansion missing, asm statement support in GRA and LRA in CG, builtin function support, etc.

In the following sections, we will introduce the detailed method to analyze and fix the typical cases or problems listed above.

## 3.1 Common issues

### 3.1.1 The O0 CFG build-up and clean problem

Gcc regards control flow graph building as an essential step in the compilation. This is also for portability that applied to a broad range of distinct architectures. And CFG build-up is default enabled at all optimization levels, including O0. Open64, on the other hand, believes O0 means only front-end whirl generation and back-end code expansions. So, Open64's O0 level keeps those unreachable target-specific code for other architectures until the CG phase which of course breaks the compilation.

We have made many efforts to resolve this problem, including: open CG cflow optimization for O0, add CG CFG build up phase, WGEN expansion optimizations, etc. However, for the reasons of :

1) We still don't want introduce any optimizations to O0
2) We believe introduce new phases will take more efforts and not a good practice regarding software re-use
3) We still want to solve the problem in the back-end for the integrity consideration

Finally, we discarded these immature ideas and resort to invoking restricted WOPT at O0 or O1. Inspired by preopt du-only phase, we add phase "q" for O0 or O1. It does only restricted copy-propagation and dead code elimination (unreachable code elimination). We handle carefully on the optimization flags to make sure unintended optimizations would not be done. We think this approach operating at a relatively high level and re-using the matured WOPT code coincide with good practice principles. Problem here is that opening WOPT at O0/O1 levels will introduce long compile-time expense, since some infrastructure operations like BB/D-U construction, Alias Analysis, HSSA constructions are essential parts of WOPT. Another issue is that we may lose some dwarf-info during WOPT processing. We have a preliminary work patch for this issue, but we are still working on improve it to be a matured solution.

### 3.1.2 The O0/O1 inline invoking problem

Gcc believes call graph building is an essential step in the compilation. So, gcc deletes non-called file-scope routines with inline attribute at any levels. Open64 still makes conservative handling on inliner, the behind critiea is that inliner should only be called at optimization levels and also the call graph based dead function elimination functionality is coupled with the inliner. This causes the non-called inliner routines which are not designed for current architecture unable to build.

We solved this problem by calling inliner even at O0. At gcc spin to whirl transformation phase, if there are inline attributed routine, we launch the inliner which then calls DFE first, after that we make carefully handling on the command line arguments. Only if there are explicit instructions to inline, we open the inline processing, otherwise, we do no inline.

### 3.1.3 ASM_INPUT copy-propagation problem

As well-known, Open64 disables copy-propagation into ASM_INPUT argument due to the constraint required be in specific form. However, the disabled copy-propagation makes the unexpected self assignments deleted in WOPT which then causes the wrong compilation. Fixing the bug by keeping specific self assignments live is not consistent to the WOPT design since we may want distinct flags to indicate which self assignment are to keep, this is not general and convincible. Finally, we solved this problem by a test and approve approach, i.e, we try propagate to ASM_INPUT's arguments first, if we find the propagated expression is the same form to original one, we allow this propagation.

Direct result of this fix is that we both get performance upgrade and compile time reduces. In a test for the whole kernel build, we get 12560 test/replace out of 428777 tries in 3845 files compiling. And we also find some optimizations originally disabled due to ASM_INPUT as the copy-prop barrier, now enabled again, and shows its power[8].

#### 3.1.4 Support for GCC extension Variable-Length array in Structure

Variable-Length Arrays (VLA) are c99-introduced[9] feature which means a procedure scope array can be variable length, i.e, allocated during run-time. VLA can be considered as a language standard to substitute for alloc call. GCC extends this feature to enable aggregates like structs/unions containing VLA, called Variable-Length array in Structure (VLS). A typical VLS form as follows:

```
struct {

struct sd_shash;

char ctx[csize(tfm)];

} desc;
```

Although, VLA is get commonly supported and accepted as the language standard, VLS is still vague and not quite commonly understand for the following reasons:

- It is completely undocumented: although GCC supports this feature, but there are few document specify this feature at the GCC official site. We can only guess that this feature is used to facilitate the pattern of combination of structs with flexible array members and VLA. In the above example, struct *desc* has a flexible-array-member field *ctx*, which now allocates the succeeding *csize(tfm)* bytes behind the structure, so desc.ctx can be easily access the succeeding data, without hard coded allocation and flexible-array-members definition. Although we can guess this usage for VLS, but GNU does not make confirm and documentation on it. So, implement this feature is quite unsure for non-GNU compilers.
- there are semantic extensions for this feature, c99 standard[9], page 102, describes the elements of aggregate as: *A member of a structure or union may have any object type other than a variably modified type.* This means each element of the struct or union is non variably modified object. We can also deduce that elements's offset from the head of struct/union is an known constant, i.e, access each field of the struct does not affected by the other fields. However, see the below VLS:

  ```
  struct {

  int a;

  int b[x];

  int c;

  } desc;
  ```

  field a and field b are accessed at fixed offset, while field c is accessed at the (1+x) integer from the head of the struct. Now, access of field c can be affected by a variable x, which is belonged to variably modified field b. Semantically, access structure field now can be affected by other variables, so this is an unapproved fundamental change for the elements of the language.
- The type size problem. VLS makes structure a variably modified type with unknown size. Although GCC dumps VLS structure size as -1 for unknown in the dwarf symbol info. This is not standard, other compilers may have symbol verify difficulties with this -1 for structure size. it means the structure/union are now unknown size, unable to verify the fields are accessed/aligned with the right value. And also, other frontend may have sizeof difficulties for VLS.

Although we have many debates and discussions on the VLS support in Open64, we finally decide to support VLS since Open64 uses the GCC frontend. Luckily, GCC helps up much in the SPIN tree dump: the offset computation, the *sizeof* issue, the transformation and protection of access negative indexed array ... etc are all considered by GCC, and despite the semantic question, we successfully implemented VLS in Open64. Currently, we enable -1 for VLS size and haven't find any issues on the symbol verification in Open64. We don't consider too much on the passing VLS as parameters since this is undocumented ABI.

#### 3.1.5 The GNU ASM alias issue

In Open64, an assertion occurs in WGEN phase when two objects which have been declared to be alias to each other, but actually have two different storage types. For example, an array is declared as an alias to a function. More accurately, this bug is caused by an unsupported GNU C extension, which is used in Linux kernel for page management. In ASNI C, this case is strictly treated as an error; while in GNU C Extension, it is legally supported by Gcc. Open64 initially threw an assertion fail in ST_Verify_Fields, we finally support this extension by modifying the function WGEN_Assemble_Alias.

### 3.2 X8664 specific issues

#### 3.2.1 Front-end bugs

Currently, Open64 front-end compilation includes two steps:

1) Invoke gcc to generate spin tree. A spin tree is a tree expression of the Abstract Syntax Tree (AST), which is very similar to gcc's internal tree expression. The spin tree is generated by patching in the GCC code, which is quite hard to read and maintain.
2) Wgen translates spin tree to whirl binary files including the whirl code and symbol table. Although the WGEN skeleton code is in readable shape, too many un-structured detail handlings have made the whole WGEN code quite difficult to maintain and update.

Although many efforts have been put in wgen and spin, we still find serious logic bugs in spin generation and wgen, including: incorrect logic to update nested field id (wgen), missed updating struct align properties in the right context(spin), incorrect setting alias attribute(wgen). We believe it will need much more effort to focus and fix the unknown front-end bugs, since GCC's internal structures are quite changed from version to version and the patching development mode is not good software practice in the long run. So, we strongly suggest Open64 switch to individual front-end for continuously develop and upgrade. Clang[6] has shown its mature and production quality, which is a good candidate for future development.

#### 3.2.2 Incompatible option fix

Besides option control made in section 2.2, we still find some option related bugs:

1) "-print-file-name=library": This option is used to get the path of compiler library header files.
2) "-m elf_i386/-m elf_x86_64": Open64 treat them as a group, thus was wrongly transferred to linker.
3) "-mregparm=num": this option is used to control the number of registers used to pass integer arguments. We found boot failures caused by it, there are some hard coded asm in kernel which designed to use specific registers to store the parameters, and this is based on -mregparm=num is correctly set. But Open64 did not support it in the driver, and also we need it works for built-in functions. We fixed it by enable -mregparm=num in OPTIONS and -TENV:mregparm=num for built-in functions.

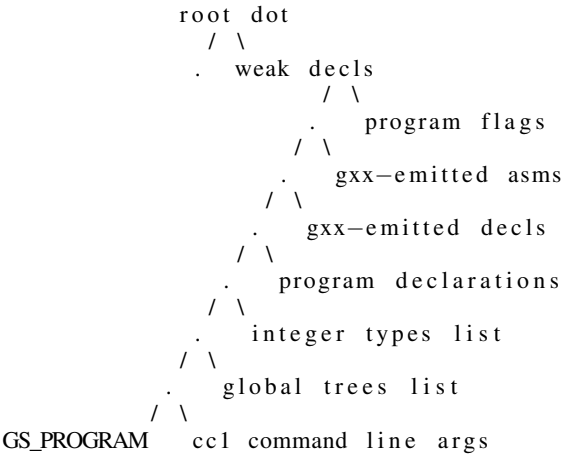
#### 3.2.3 Conflict between Kernel code and Open64

During debug kernel problems, there are some conflicts found between kernel code and Open64 implementation. That is if fix the problem at Open64 will break original design principle or will cost a lot. i.e. Open64 will generate a lot of temporary variables during optimize phase, this causes the stack usage is often larger than gcc, one bug found during runtime is the kernel corrupts due to stack overflow. To fix this kind of problems, we make patches on kernel code. But this is just a short-term solution, for the long run we need to consider how to solve it in open64.

#### 3.2.4 The global file scope asm

There is one serious bug been fixed related with global file scope asm. Open64 always put global asms code to the end of assembler file. But sometimes the global asms have code directives transferred to assembler, and should be put where it defines, otherwise the meaning is different. The buggy case we found in kernel is when using asm(".code16gcc") at the begging of a file, ".code16gcc" is an code directives, it specifies the code being assembled was generated by GCC and therefore is 32-bit assembly code that will run in a 16-bit segment.

To fix this problem, we changed the place where emit asm strings, the dot structure for GS_PROGRAM as follows, in previous version of Open64, the global asm strings were emitted at global trees list(the last field of GS_PROGRAM), that is why we see it at the end of gspin and assembler file. And we found when parse declarations, it just add the asm node to the top-level asm statement node list, and postpone to emit them at the end. Our fix is to break this, once there is an asm declaration, we emitted it as soon as possible.




After some testing, we found regressions caused by above fix, the root reason is when use global asm, compiler do know the correct origin of objects to be allocated, so we should not emit .org at this time, this is implemented by adding a new file info flag:FI_HAS_GLOBAL_ASM, and this flag is set at wgen phase when analyze gspin declarations.

With above two fixes, the global file scope asm problem was finally resolved.

#### 3.2.5 Other issues and fixes

Besides the above work, we also find the following issues and fixes:

- Implement the asm constraint "p" for memory load and push.
- fixed two wopt bugs due to typo and vague boundary constant handling.
- fixed many CG bugs in the quite sensitive part, for example, the wrong EBO peephole optimization, the wrong handling of some asm constraint, improved asm constraint based register allocation, . . . , etc.

### 3.3 MIPS-specific issues

#### 3.3.1 Incompatiable option and C Language extensions

1) Embedded assembly: As previously noted, there are about 29 bugs in this category. Especially, we note that bugs caused by unsupported embedded assembly extension occupy much more percentage (25 out of 50) than other C language extensions. This phenomenon results in that embedded assembly extensions for MIPS target are not well-documented and Linux kernel uses lots of features that the only document is GCC source code, which makes even harder for us to resolve corresponding bugs. We have to use a test-driven framework for enabling embedded assembly support for MIPS. By using the assertion mechanism of Loongcc, comparing the assembly generated by Loongcc and GCC and referring to source code of Gcc, we finally make it work. These bugs are mainly about new MIPS embedded assembly input/output constraints support, including the value range assertion of the immediate, the data-type of the input/output variables, ways of register allocation, etc. These supports are mainly done in the WHIRL to OPS phase in CG when handling ASM relative stuff. Besides, there are also modifications in other phases of Open64 for assembly support, including copy-propagation supporting for specified constraint of asm statement in WOPT, asm WHIRL generation support in WGEN, and assembly modification for specified constraints in CG emit.
2) Special incompatible Options : Besides option control made in section 2.2, we enhance Loongcc to support option "-print-file-name=include", this option is used by Gcc or Open64 to get the path of compiler library header files. Because Loongcc is a X86-MIPS cross compiler, the library path is different from Open64. The way Open64 using is not suitable for Loongcc. We add NAMEPREFIX information to help Loongcc to invoke mips64el-st-Linux-gnu-gcc to get the right path.
3) N64 ABI support: Loongcc was originally designed for 32-bit user-space high performance computing applications named N32 ABI(Application Binary Interface) on Loongson3[15, 16], which is a MIPS-64 compatible 8-core CPU. It can only generate N32-ABI compatible codes. However, to fully exploit 64-bit CPU performance, only MIPS N64 compatible Linux kernel is in practical use on Loongson platform. Thus, we have to enhance Loongcc with N64 ABI support for Linux kernel building. Besides, MIPS N64 feature is a critical step for more 64-bits HPC applications to run on Loongson-based servers. This support is been done by carefully modifying the target information and dedicated TN relocation in Loongcc.
4) Position dependent code support: Besides the N64 ABI feature of the available Linux Kernel release on Loongson3, this

release is also position dependent, that means the kernel executable must be loaded into fixed-physical memory address for successfully booting. Loongcc formerly only supported user-space position independent code. Therefore, position dependent feature must be developed firstly. We extended the symbol flag for MIPS relocation information, which is used in function Exp_Ldst, to generate an offset relative to the address of this flag. In addition, there are also some modifications in function call WHILR_expanding. In this way, Loongcc does not turn to the original gp-relative mechanism for locating a symbol.

5) Built-in function: Another Loongcc defect exposed is caused by the reference of Gcc built-in function __builtin_return_address. For better performance, Gcc provides a set of built-in functions. One of these functions is __builtin_return_address, which returns the return-address of the current function. In Open64, it is translated into a built-in symbol "__return_address" which is located in the frame controlled by FP. As a result, the FP register $fp is introduced after code generation. But in the localization phase, it is considered as illegal. The solution is, in CG local register allocation phase, we modified Loongcc to not allocate dedicated FP register for this function.

### 3.3.2 Loongcc defect

1) Inline : For Loonson, Open64 does not inline a variable argument function with empty body even if the function has inline qualifier. On the contrary, Gcc always inlines such a function, neglecting whether the function has an empty body. There was a bug triggered by this different feature. Function pr_debug has reference to a structure pptp_msg_name which is only defined when macro DEBUG is turned on. In release mode, although the reference to undefined pptp_msg_name still exists, after inlining there is no reference generated because pr_debug's body is empty. However, as Open64 does not inline this variable argument function pr_debug, references to the undefined structure pptp_msg_name still exists and thus a link error occurs.We fixed this defect by taking the same strategy of Gcc.

2) WOPT unsafe strength reduction transformation: We have solved a serious defect that misused homomorphism map between modular integer rings with radix $2^{32}$ and $2^{64}$. The problematic code was as below.

```
int32 done = 0;
int64 data = 0;
do
{
 done += 64;
 src = data + done;
} while (done + 63 < len);
```

When doing invariant variable hosting, Loongcc first identified "data" that is an invariant variable having the potential to be hoisted. Thus, the compiler has mistakenly merged two loop statements into one, namely yield the code as below:

```
int32 done = 0;
int64 data = 0;
do
{
 src = (data + done) + 64;
} while (done + 63 < len);
```

As known, the variable "data" is 64-bit integer in the ring of Z/nZ, where n is $2^{64}$; and "one" is 32-bit integer in the ring of Z/mZ, where m is $2^{32}$. And, the homomorphism map

$$\text{CVTU8U4(A+B)=CVTU8U4(A)+CVTU8U4(B)}$$

does not hold, where CVTU8U4 is an integer conversion from 32-bit to 64-bit. For example, assume A=$2^{32}$-1, B=1. So, we have

$$\text{CVTU8U4(A)+CVTU8U4(B)} = 2^{32}$$
$$\text{CVTU8U4(A+B)} = 0$$

Obviously, the assumption of homomorphism was wrong. This defect caused a memory corruption, since the variable "done" had kept a wrong memory address. Our fix have removed all this kind of wrong logics.

3) CG: In CG function Assign_Registers_For_OP, Loongcc invokes function OP_side_effects to check whether an OP has side effects or not. However, OP_side_effects omitted OP attribute volatile which implicates a possible side effects. In the following code segment from MIPS Linux kernel, asm function called by atomic_add actually has side effects on variable temp. Because temp is used in the atomic addition operation in asm function. Despite the asm function has a volatile attribute, Loongcc still misjudged the case. Thus, in the later local register allocation phase, Loongcc allocate register $0 for variable temp. On MIPS platform, Register $0 always holds value zero. Therefore an infinite loop was formed. The solution is to replace OP_side_effects with OP_has_implicit_interactions which checks volatile flags of an OP.

//Side effect asm function in arch/mips/include/asm/atomic.h

```
static __inline__ __attribute__((
    always_inline))
void atomic_add(int i,atomic_t * v)
{
    //other code omitted ...
    int temp;

    __asm__ __volatile__(
    "       .set    mips3\n"
    "1:     ll      %0, %1   # atomic_add\n"
    "       addu    %0, %2\n"
    "       sc      %0, %1\n"
    "       beqz    %0, 2f\n"
    "       .subsection 2\n"
    "2:     b       1b\n"
    "       .previous\n"
    "       .set    mips0\n"
    : "=&r" (temp), "=m" (v->counter)
    : "Ir" (i), "m" (v->counter));
    //other code omitted ...
}
```

//assembly code wrongly generated by Loongcc

//for asm segment of atomic_add

```
1:  ll    $0, ($25) # atomic_add
    addu  $0, $31
    sc    $0, ($25)
    beqz  $0, 2f    # $0 never change
                    # to a non-zero value
    .subsection 2
2:  b     1b
    .previous
    .set mips0
```

## 4. Current status

### 4.1 Loongson

After much effort, Linux kernel 2.6.35 is successfully compiled by Loongcc and booting on a virtualized MIPS64 machine, which is simulated with Qemu 0.12.5. And this Qemu has been modified to support Loongson3 special hardware features. The version of the GDB client is mips64el-unknown-Linux-gnu-GDB 7.3.1 with a patch[14] which enable the 32-bit GDB client work correctly with the 64-bit GDB server built-in Qemu. Malta mother board embedded a MIPS64 CPU is simulated, and memory size is 256 MBs.

### 4.2 Open64

Linux Kernel versions 2.6.32.6 successfully built and boot on x8664 at O2. About 36 bugs reported at Open64 bugzilla[7]. All bugs have been fixed at Open64 trunk or have workaround fix.

### 4.3 Future work

In future, there are a lot of interesting works.

***For Loongcc*** Robustness and performance for a full MIPS(Loongson) Linux distribution will be enhanced continually, of which the most compelling is shown below:

1) Merge update on Loongcc branch to Open64 trunk.
2) Build Linux kernels on all Loongson products and boot on real machines.
3) Use Loongcc for building more basic GNU/Linux system components, key applications, and finally a full Linux distribution. We have successfully built binutils, gcc, Mplayer and bootstrap.
4) Improve the command line compatibility between open64 and Gcc. Before any critical decisions made by OSG on the front-end of Open64, compatiability with Gcc is still a key feature.
5) Optimize the performance of Linux Kernel and key applications, such as firefox, Apache and the like.

***For Open64*** There are at least 3 areas need further work:

1) Improve Open64 and remove workaround fix. Some of the workarounds have not been approved by the community, we are working continuously to improve the workarounds to formal bug fix. We believe these upcoming fix for bugs will enhance the Open64's stability more.
2) Build and boot kernel at more optimize level. Currently, we have only successfully build and run Linux kernel on O2 optimization. Other levels, just like the "O0 -g" to get the debug build kernel, IPA/IPO customized kernel for performance and O3 level for aggressive optimization have not been built-up yet. Working on these levels will benefit the study for both the Open64 compiler and Linux kernel. We wish we can gain more experiences and contributes more to the open source community.
3) Boot Open64 built kernel on real machines, install packages and do more tests. All the triage and run up tests have been performed under the QEMU vm machine, we have not prove the Open64-built kernel successfully run on real machines, this should be a short-term goal. Unless the kernel run stably and optimized on real machines, we can say we achieve our initial goal. Only by showing the production quality running the kernel stuff, we can attract more people get interested in Open64's ability in system software are. As a long-term goal, we'd like to compile and run large volumes open source softwares under Open64's native kernel. We have already done things on this area, we will work on to bring up Open64 as an alternative compiler for a Linux distribution.

## Acknowledgments

Tao and Kun's team are grateful for Loongcc team members: Hu Shiwen, Huang Lei, Li Xin, Lian Ruiqi, Liu Ying, Lu Tingyu, Zhao Hongjian and Zhang Mang, who fixed most of the bugs in Loongcc MIPS64 Linux kernel building process.

Qing, Gang and Jian-Xin would like to thank the Open64 community, especially Sun Chan for their helpful reviews, comments and suggestions.

Lastly, honours should be ascribed to developers of MIPSPro, Pathscale and ORC for their contributions to the open source society.

## References

[1] Zhou Shuchang, Liu Ying, Lv Fang, Yin Le, Huang Lei, Li Shuai, Ma Chunhui, Gao Zhitao, Lian Ruiqi, Open64 on MIPS: porting and enhancing Open64 for Loongson II, Open64 Workshop at CGO 2009.

[2] https://events.Linuxfoundation.org/slides/2011/lfcs/lfcs2011_llvm_lelbach.pdf

[3] The QEMU project. http://qemu.org

[4] Delta page. http://delta.tigris.org

[5] Linux Kernel Project. http://www.kernel.org/

[6] The clang Project. http://clang.llvm.org

[7] The Open64 bugzilla. https://bugs.open64.net/

[8] http://sourceforge.net/mailarchive/forum.php?thread_name=CAEwdn1C

[9] The ISO/IEC 9899:TC3: Programming Language C. www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

[10] The Impact of Compiling a LINUX Kernel with INTEL C/C++ Compiler on Computer Clusters Used by Science. Problems of Engineering Cybernetics and Robotics,ISSN:0204-9848, Volume 56,2006. http://www.iit.bas.bg/pecr/56/78-85.pdf

[11] The Linux Kernel build white paper. http://software.intel.com/file/6390

[12] [ANNOUNCE] Clang builds a working Linux Kernel (Boots to RL5 with SMP, networking and X, self hosts), http://lists.cs.uiuc.edu/pipermail/cfe-dev/2010-October/011711.html

[13] GNU C Language Extensions. http://tigcc.ticalc.org/doc/gnuexts.html. Dec 20th, 2011

[14] Patching GDB 7.3 for QEMU remote kernel debug. http://torokerneleng.blogspot.com/2011/08/patching-gdb-73-for-qemu-remote-kernel.html. August 25th, 2011

[15] Loongson 3B CPU details. http://www.loongson.cn/EN/product_info.php?id=33. Nov 15th, 2011

[16] HU W, WANG R, CHEN Y, et al. Godson-3B: A 1GHz 40W 8-core 128GFLOPS processor in 65nm CMOS. Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International.2011