Retargeting Open64 to A RISC processor

-- A Student’s Perspective

Huimin Cui, Xiaobing Feng
Key Laboratory of Computer System and Architecture,
Institute of Computing Technology, CAS, 100190 Beijing, China
{cuihm, fxb}@ict.ac.cn

Abstract

This paper presents a student’s experience in Open64-Mips prototype development, we summarize three retargeting observations. Open64 is easy to be retargeted and the procedure takes only a short period. With the retarget procedure done, the compiler can achieve good and stable performance. Open64 also provides many supports for debugging, with which a beginner can debug the compiler without difficulty. We also share some experiences of our retarget, including methodology of verifying the compiler framework, attention to switches in Open64, importance of debugging and reading generated code.

1 Introduction

Open64 receives contributions from a number of compiler groups around the world, from industry as well as from academia. It is derived from the SGI MIPSPro64 compiler [1]. Good performance and industrial strength origin make the Open64 compiler a popular choice for research projects. The Open64 compiler supports C/C++ and Fortran95 languages and many researchers from industry and academia have contributed to the retargeting of the Open64 compiler [6].

Open64 has been retargeted to a number of architectures. Pathscale modified Open64 to create EkoPath, a compiler for the AMD64 and X8664 architecture. The University of Delaware's Computer Architecture and Parallel Systems Laboratory (CAPSL) modified Open64 to create the Kylin Compiler, a compiler for Intel's X-Scale architecture [1]. Besides the targets mentioned above, there are several other supported targets including PowerPC [6], NVISA [9], Simplight [10] and Qualcomm[11].

In this paper, we will discuss three issues when retargeting Open64: ease of retarget, performance, and debuggability. The discussion is based on our work in retargeting Open64 to the MIPS platform, using the Simplight branch as our starting point, which is a RISC style DSP with mixed 32/16 bit instruction set. During our discussion, we will use GCC (Mips target) for comparison. This retarget work is just a student project, so we have some non-goals, which will be discussed in section 5.

And we will also share some of our retargeting experiences, in four folds: 1) Steps to verify the compiler framework. 2) Turn on/off switches for special hardware features. 3) Debugging is very important, not only for retarget, but also for later extensive work. 4) Pay attention to the generated code if it doesn’t work

as expected.

The paper is organized as follows. Section 2 introduces the retarget procedure. Section 3 presents our major retarget results and analysis. Section 4 summarizes our experiences and expectations. Section 5 shows our non-goals and section 6 concludes the paper.

2 Suggested Retarget Procedure

This section presents our retargeting procedure, and it is applicable if one can find a similar processor in Open64's supporting targets. Here, letters (A,B,..) represent the file creation order, and numbers (1,2...) represent the building order.

(A). Add make rules for your target(<targ>) in top level Makefile.gsetup

(B). Create dir targia32_targ<targ>

This is the build directory for your new target. Host is specified as ia32.

(C). Create and set up Makefile in each subdir under targia32_targ<targ>

Make sure the BUILD_TARGET is set correctly so it goes to the right target directory for each sub component.

Set your BUILD_VENDOR correctly.

(1). Build include

(D). In common/com, create <targ>/ config_ targ.h

Add macros for Is_Target_xxx, xxx represents the supported processors of the new target, e.g, Is_Target_R10K() for MIPS.

Add ABI macros for the new target.

Set GP area size.

Set target debugging information.

In common/util, create <targ>c_qwmultu.c by copying from ia64, or as appropriate.

(2). Build libcmplrs, libiberty, libcomutil

Sometimes, it helps to show the compile line during building of the compiler, simply change gcommondefs and gcommonrules in linux/make/.

(E). In common/targ_info, create <targ> dir's and their files. Use this order:

isa/<targ>/isa.cxx
isa/<targ>/isa_properties.cxx
isa/<targ>/isa_subset.cxx
isa/<targ>/isa_registers.cxx
isa/<targ>/isa_enums.cxx
isa/<targ>/isa_lits.cxx
isa/<targ>/isa_operands.cxx
isa/<targ>/isa_hazards.cxx
isa/<targ>/isa_print.cxx
isa/<targ>/isa_pack.cxx
isa/<targ>/isa_bundle.cxx
isa/<targ>/isa_decode.cxx
isa/<targ>/isa_pseudo.cxx
abi/<targ>/abi_properties.cxx
proc/<targ>/proc.cxx
proc/<targ>/proc_properties.cxx
proc/<targ>/<targ>_si.cxx

This directory contains most of the basic setting for retarget purposes. Most retarget problems will be due to wrong settings in this directory. All these files should be written carefully.

(3). Build targ_info

(F). Deal with frontend in kgccfe and kg++fe directories.

In kgccfe/gnu, kgccfe/gnu/config, kg++fe/ gnu, kg++fe/gnu/config, create directory of <targ> by copying from ia64 or x8664 and fixing the values and the include file names as well as define statement.

Fix up the gnu_config.h to include the corresponding config.h file for your target.

Fix up the <targ>/config.h, <targ>/ hconfig.h, <targ>/tconfig.h, <targ>/tm_p.h, to include the corresponding file for your target.

Make sure Makefile and Makefile.gbase know to go to gnu/<targ> directory also. It requires one to check whether gnu/<targ> has been added into SRC_DIRS.

In include/elf.h, the #define for Elf32_Byte and Elf64_Byte can always be enabled.

If need to add intrinsic, the following files needs changing.

- intrinsic.def – defines

INTRINSIC_ID, property of intrinsic (order is important, indexed by INTRINSIC_ID)

- FE definition files, kgccfe/gnu
- Builtins.def – id, name, prototype, attribute
- Builtin-types.def – needed if new type is needed
- Wfe_expr.cxx – translate GNU builtin to WHIRL.

(G). Create the following files:

common/com/<targ>/config_platform.h
common/com/<targ>/config_asm.h
common/com/<targ>/config_cache_targ.cxx
common/com/<targ>/config_elf_targ.cxx
common/com/<targ>/config_host.c
common/com/<targ>/config_platform.c
common/com/<targ>/config_targ.cxx
common/com/<targ>/config_targ.h
common/com/<targ>/config_targ_opt.cxx
common/com/<targ>/config_targ_opt.h
common/com/<targ>/targ_const.cxx
common/com/<targ>/targ_const_private.h
common/com/<targ>/targ_ctrl.h
common/com/<targ>/targ_em_const.cxx
common/com/<targ>/targ_em_dwarf.cxx
common/com/<targ>/targ_em_dwarf.h
common/com/<targ>/targ_em_elf.cxx
common/com/<targ>/targ_em_elf.h
common/com/<targ>/targ_sim.cxx
common/com/<targ>/targ_sim.h

All these files can be copied from the directory of a similar target, and modified as required.

Make sure the endianness is set correctly. It is set in config_host.c, config_targ.cxx and common/com/config.cxx.

Make sure the assembly syntax for .s file output is set correctly. It is set in config_asm.c.

Make sure the calling convention and parameter passing are set correctly. They are set in targ_sim.h and targ_sim.cxx. If you need your own ABI definition, just change this file, and do not forget to modify targ_sim.cxx for consistency.

config_cache_targ.cxx sets cache models.

Targ_em_* files are for assembly output control, section types, relocations, dwarf etc. One only need to change these files when focus in BE portion.

Basically one needs to tailor files mentioned about and then continue building components as follows.

(4). Build gccfe

(5). Build g++fe

(6). Build ir_tools

(H). Create be/be/<targ> and be/com/<targ> directories

Create driver_targ.cxx, fill_align_targ.cxx in be/be/<targ> dir

Create betarget.cxx, sections.cxx in be/com/<targ> dir

There is no need to be optimal at this point, just make it work, and then make it perform. e.g. One can set Can_Do_Fast_Multiply to return FALSE, and return to that later.

(7). Build be

(8). Build libelf, libelfutil, libdwarf, libunwindP

(I). Create be/cg/<targ>/register_targ.h

be/cg/<targ>/tn_targ.h
be/cg/<targ>/op_targ.h

Create one's own target specific portion in lib/elf.h (or appropriate elf.h), define the right relocations, sections, etc.

(9). Build wopt

(J). Create files in be/cg/<targ>

Fix cgtarget_arch.h, such as CGTARG_ Copy_Op, ...

There might be needs to add more things in be/cg/op.h

Fix cgdwarf_targ.cxx, Find_Spill_TN, ..., unwind table in dwarf.

Copy other files from a similar processor's directory. Go over expand.cxx, whir2ops.cxx carefully. And exp_branch.cxx, exp_divrem.cxx, exp_loadstore.cxx, expand.cxx, entry_exit_targ. cxx needs to be changed or tailored substantially.

(10). Build cg

(11). Build driver
(12). Build ipl
(13). Build lno
(14). Build inline
(15). Build whirl2c
(16). Build ipa

3 Retarget Results & Discussion

3.1 Machine Assumption & benchmarks

Machine Assumption:

Processor: Loongson 2f, out of order
ISA: MIPS3
Frequency: 666MHz
Issues: 4
ALU: 2
FALU: 2
MEM Unit: 1

It is customary to use SPEC CPU2000 or 2006 to evaluate a compiler performance, but that is our non-goal (discussed in section 5). As a student project, we only consider the following benchmarks: Stanford benchmark, the abstraction penalty benchmark written by Stepanov [3], two programs chosen from SPEC CPU2000 - bzip2 and mcf.

Stanford benchmark [2] consists 7 small programs: hanoi, bubble, matmul, perm, qsort, queen, and sieve. The abstraction penalty benchmark summarizes the characteristics of generic programs; it measures the performance under different levels of abstraction for C++ programs. The baseline is a non-generic version of the kernel function, as shown by Figure 3.1.

```
double data[SIZE];
… …
for(int i = 0; i < iterations; ++i) {
      double result = 0;
      for (int n = 0; n < last - first; ++n)
          result += first[n];
      check(result);
}
```

Figure 3.1 non-generic version in abstract penalty benchmark

3.2 Main Results

Observation 1 (See also section 3.3.1). A student can complete the retarget procedure in about two months, if an appropriate supported target processor can be used to start the retargeting. In our case, the Simplight SL1 processor is a RISC based processor with 16 bit instructions and DSP extensions. We chose this as our starting point for its RISC basis as ISA.

Observation 2 (See also section 3.3.2). Performance of the generated Open64 compiler is satisfying, in two folds.

1. We expect the retargeted Open64 compiler will generate very good code due to the excellent middle ends such as Wopt and Lno components. Indeed, when we achieved competitive or better performance compared to a matured GCC compiler, for all the benchmarks we tested as a result of our effort.

2. Open64 provides obvious performance improvement with optimization option switched from –O2 to –O3, especially for C++ programs with higher abstraction levels.

Observation 3 (See also section 3.3.3). The debuggability of Open64 compiler and its generated code is good, it is not difficult for a beginner to learn and use.

3.3 Detailed Discussions

3.3.1 Ease of Retarget

Following the suggested retarget procedure, we completed our retargetting work in two months. In a short period, our compiler has passed all the benchmarks mentioned in section 3.1, with option from –O0 to -O3. It means that our compiler framework is reasonable and the major components work well.

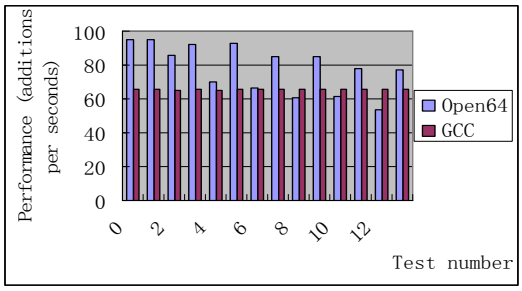
As we know, compiler development needs to deal with the issues of ISA, ABI, code generation, and some machine-dependent functions or methods. All these issues are covered in the suggested retarget procedure, as presented in Section 2.

Porting GCC is similar; it requires the developers to consider ISA and code generation in the .md (machine description) file. While ABI, processor information and machine-dependent subroutines should be considered in some machine-dependent header files and C source files [4][5]. These issues are common for both Open64 and GCC.

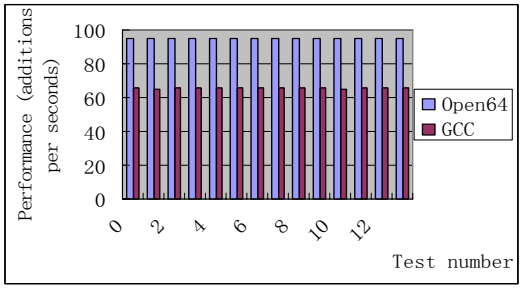
3.3.2 Performance Results and Discussions

In this subsection, we will present our performance comparison with GCC for the three benchmarks mentioned earlier. The performance is measured with just the retarget procedure done, it should be emphasized that we did NOT do any machine-dependent performance tuning of the Open64 compiler during this exercise.

Figure 3.2 shows the performance comparison for abstraction penalty benchmark between Open64 and GCC. We can see that Open64 provides better performance for both *–O2* and *–O3* options.



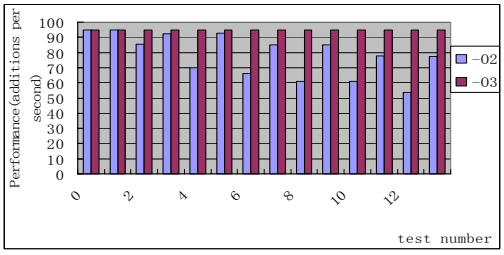
(a) performance comparison with *–O2*



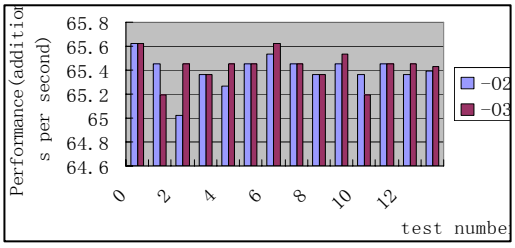
(b) performance comparison with *–O3*

Figure 3.2 Performance comparison for abstraction penalty benchmark

Figure 3.3 shows the performance comparison of -O2 and -O3 for Open64 and GCC. Open64 –O3 option provides obviously better and more stable performance for higher abstraction levels, comparing with –O2 option. While GCC has no obvious difference for the two options.



(a) Open64 performance



(b) Gcc performance

Figure 3.3 Performance comparison of -O2 and -O3, for abstraction penalty benchmark.

Figure 3.4 shows the performance comparison for Stanford benchmark. Open64 provides better performance than GCC, with options –O2 and –O3. The vertical axis is normalized execution time to GCC –O2. We can see that *matmul* shows dramatically performance increasing for Open64 –O3 option. The reason is that loop tiling is applied, which is effective for matrix multiplication, especially when the problem size is large.

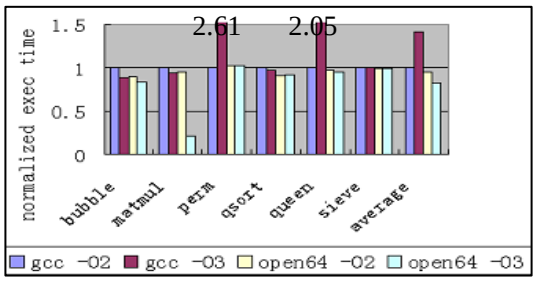


Figure 3.4 Performance of Stanford benchmark

Figure 3.5 shows the performance of Open64 and GCC for bzip2 and mcf, from SPEC CPU2000. Open64 and GCC have no obvious performance difference.

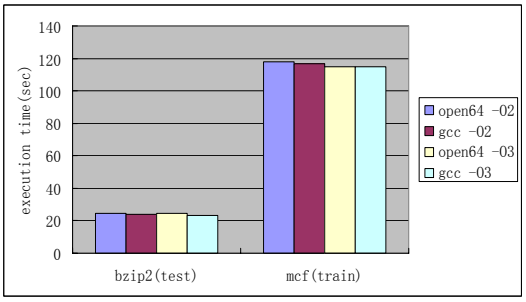


Figure 3.5 Performance of bzip2 and mcf.

3.3.3 Debuggability

Open64 provides many supports for debugging the compiler, including: 1) dumping out the program code and symbol table between phases. 2) tracing of optimization and analysis process. 3) builtin options to allow one to isolating program file that triggers the bug, and isolate the PU, the expression, basic block, etc that exhibits the bug. 4) some dump methods that can be called in GDB.

With these supports, the compiler can be debugged in an intuitive way. Here are our steps: 1) use isolation tools to find which PU in which file causes the bug. 2) read the assemble code of that PU carefully, to find out which BB or BBs are translated incorrectly. 3) dump out the intermediate representations between phases as a *.t* file, read the *.t* file carefully to find out the bad optimization phase. 4) Now we can know which optimization causes which BBs error. And can use GDB to debug the optimization phase. Dumping methods can be used in debugger, such as WHIRL dumper, BB dumper, etc.

Also Open64 provides friendly comments in the generated file, to help the reader understand the assemble code. For example, a loop would be commented with its line number in source file, nesting depth, estimated iterations, and unroll times. It clearly showed the loop transformations and helps the reader understand the assemble code more easily.

4 Experience and Suggestion

In this section, we will share our experience and our suggestion for Open64. We hope our experience might be useful for future developers to port Open64 in a short time. Some issues are related to the methodology of retargeting, and some might be trivial but useful.

Experience 1: How to verify the basic issues of the compiler framework step by step quickly? Our experience is choosing benchmarks from simple to complex. The following shows our steps.

Step 1: taking "hello world"(-O0) as the startup. If it works well, it means the modules of front-end \ ir-transformation\ code-generation\ lib-function-call are reasonable.

Step 2: focus on the variations of "hello world"(-O0). Change data type of the parameters, and change the number of parameters. Add some control-flow into the test-case, eg, branch and loop. If all the variations work well, it means the ABI and variant-as-parameter are reasonable. Remember to test for 64-bit immediate values if you are retargetting for a 64-bit platform.

Step 3: turn -O0 into -O2 and -O3 for hello-world and its variations. If it works, the optimization *FRAMEWORK* works. But optimization is a complex issue, and needs more efforts.

Step 4: taking "stanford benchmark" for test (-O0 to -O3). If the suite works well, it means the compiler works for multiple procedures. And some bugs on code generation and optimizations would be found and fixed during this step.

Step 5: testing for "abstract penalty" benchmark. It can check the c++ frontend, and loop optimizations.

Experience 2: Be familiar with Open64's switches.

There are many switches in Open64. If we need turn on/off some machine feature, search its

corresponding switch first of all. If the switch already existed, things would turn much simpler. For example, delay slot, multiply-and-add, divide instruction, etc.

Experience 3: Easy to debug is significant.

It is not easy to modify all the files correctly at first. We can fix the bugs by running, tracing, debugging the compiler. After familiar with that, we can do extensive changes to those files.

Debugging not only fixes bugs, but also helps get familiar with the compiler more quickly and efficiently.

Experience 4: Pay attention to the generated code.

Some files will not affect the correctness of the compiler if they are not modified appropriately, but they will affect the quality of generated code. For example, if the branch cost was set inaccurately in CGTARG_Compute_Branch_Parameters (be/cg/<targ>/cgtarget.cxx), the code layout of if-statement would not match that of the real machine. Remember to check the machine parameters if compiler generates codes with poor quality.

Suggestion 1. Can the flags distributed in multiple files be merged into one file?

For example, the endianness should be set in three files: config_host.c, config_targ.cxx and common/com/config.cxx. We do not know whether there are other similar instances, but this really brings some inconvenience to the retarget procedure. If a flag can be set only in one file, it would be more facile.

Suggestion 2. Can some graphic tools be developed for view IR, flow, etc?

Now the intermediate information is shown in text format, including WHIRL node, control flow graph, etc. It is time-consuming and requires the developer be familiar with all the data structures. If there are some graphic tools, this information can be shown in an intuitive way and helps the developers find the bug point more quickly.

5 Non-goals

We mentioned earlier that we are retargeting for an Open64-MIPS PROTOTYPE. So there are some issues we have not dealt with, and we will discuss these issues in this section.

Multiple ISA support. As we know, MIPS has several ISAs, say, MIPS2, MIPS3, MIPS4, MIPS64, etc. It should be controlled and chosen with the target machine, but we did not cover it. In our prototype, only MIPS3 is supported.

Dynamic Shared Object (DSO). DSO is a library that is linked in at runtime, and it requires generation of position independent code (PIC) and position independent data (PID). We have not tested the compilation for dynamic execution.

Production Quality. Production compiler requires elaborate code reviews and extensive tests; especially the code generator part, to assure it works well and generates correct code. Much more benchmarks and testsuites would be needed for test, such as, SuperTest, CPU2000, Perennial, etc.

As a student project, we only focused on the framework of the compiler, and the IR translation. Code generation modules still have rooms for improvement.

Optimization Tuning. After retarget procedure is done, each optimization should be tested for the new target, to assure that its expectation can be reached.

We only tested for –O2 and –O3 options, without paying attention to the individual optimizations. So in the previous sections, we said our optimization FRAMEWORK is reasonable. It does not mean the optimizations are at its best, especially the peephole optimizations in the code generator.

Machine-dependent Optimization. Extra machine-dependent optimization is an important step of the retarget, including the machine parameters, memory hierarchy organizations, etc. With these optimizations, the target machine's resources can be efficiently used.

6 Conclusion

This paper shows Open64's ease of retarget and good performance that can be obtained after retarget. Debuggability is also discussed with our retarget procedure. The paper also shares our retarget experiences, including methodology of verifying the compiler framework, attention to switches in Open64, importance of debugging and reading generated code. Also we present some suggestions for Open64 community.

Acknowledgement

Sun Chan guided our retarget procedure from start, told us the methodology and detailed steps of retargeting Open64. Fred Chow provided the basic document for retargeting procedure (section 2). Sun also helped us organize and revise the paper.

We give our thanks to Professor Guang. R. Gao for giving us this opportunity. Professor Gao also gave us invaluable encouragements and discussions during the retarget procedure.

We also thank to members from ICT and Simplight for their help and discussions, and thank to the reviewers for their valuable comments.

Reference

[1] Homepage of Open64. http://www.Open64.net/

[2] W. J. Price. A benchmark tutorial. IEEE Micro, 9(5):28--43, Oct. 1988

[3] Alex Stepanov. Abstraction Penalty Benchmark, version 1.2 (KAI). Silicon Graphics, Incorporated, 199?.

[4] Homepage of GCC, the GNU Compiler Collection. http://gcc.gnu.org/

[5] Soubhik Bhattacharya, Generation of GCC Backend from Sim-nML Processor Description, master thesis, 2001

[6] Ming Lin, Zhenyang Yu, Duo Zhang, Yunmin Zhu, Shengyuan Wang, Yuan Dong, "Retargeting the Open64 Compiler to PowerPC Processor," icesssymposia,pp.152-157, 2008 International Conference on Embedded Software and Systems Symposia, 2008

[7] Juha Haataja, Ville Savolainen, Cray T3E User's Guide, (Center for Scientific Computing, Finland, 1997).

[8] The Power Challenge Technical Report. On-line. as. http://www.sgi.com/Products/ hardware/ Power/index.html.

[9] Mike Murphy, NVIDIA's Experience with Open64, Open64 Workshop at CGO 2008.

[10] K. M. Lo and Lin Ma, Quantitative approach to ISA design and compilation for code size reduction, Open64 Workshop at CGO 2008.

[11] Subrato K De, Anshuman Dasgupta, Sundeep Kushwaha, Tony Linthicum, Susan Brownhill, Sergei Larin, Taylor Simpson, Development of an Efficient DSP Compiler Based on Open64, Open64 Workshop at CGO 2008.