A Standalone GCC-based Machine-Dependent Speed Optimizer for Embedded Applications

Sylvain Aguirre[1], Vaneet Aggarwal[2], and Daniel Mlynek[3]

[1] University of Applied Science, 1400 Yverdon, Switzerland,
`sylvain.aguirre@eivd.ch`,
[2] Indian Institute of Technology, Kanpur - 208 016 (UP), India,
`vaneet@iitk.ac.in`,
[3] Swiss Federal Institute of Technology, 1015 Lausanne, Switzerland,
`daniel.mlynek@epfl.ch`

Abstract. Architecture-dependent optimizations in GNU C Compiler (GCC) have been studied in order to increase superscalar processor performance. However, for cost and power consumption reasons, embedded systems are essentially based on scalar processors.
We present a software solution at the compilation level that improves the performance of single instruction issue, in-order processors at no cost and also helps save energy. More precisely, this work highlights the GCC compiler toolchain's weaknesses in terms of efficiency for scalar processors and proposes a standalone software tool to reduce program execution time by avoiding useless `stall` cycles thanks to improved scheduling.
A custom profiler based on the VMIPS simulator has shown that many clock cycles associated with the delay slots of `branch` and `load` instructions are lost. This is due to the GCC algorithm, and more precisely the one implemented in the assembler (GAS).
To avoid some of these wasted clock cycles, an independent software program, easily integrated into the compilation flow, attempts to fill in as many `load` delay slots as it can, in addition to those filled by the GAS algorithm.
After briefly describing the gas algorithm, we detail the one implemented in our tool. Experimental results on the MIPS architecture have been performed with gains of up to 9% at no cost.

1 Introduction

Because the market has an insatiable appetite for new functionality, performance is becoming an increasingly important factor. As mentioned in [3], CPU performance depends equally on three factors : the clock cycle time, the cycles per instruction (CPI) and the instruction count. Thus, the optimizer proposed in this paper helps reduce program execution time by tending to reduce the CPI and the instruction count without deteriorating the third factor.

GCC's optimizations are mainly architecture-independent and hence it is difficult for it to take advantage of specific architectures [13]. Thus, to improve efficiency and maintain the portability of GCC, this paper proposes a standalone machine-dependent optimizer. The benefits of high level languages such the C language are conserved by running on assembly code generated by the widespread GCC [8]. Moreover, it implies no hardware cost in the sense that it takes place at the compilation time for any applications and the hardware is kept unchanged.

Section 2 introduces background compiler concepts, whereas Sect. 3 exposes the proposed GCC-based optimizer. Section 4 describes the methodology followed to obtain the results presented in Sect. 5. Finally, Sect. 6 concludes this paper in suggesting the embedded applications that may have an interest in using such an optimizer and proposing to spread this work to other architectures.

2 Background and Prior Art

Scheduling is a part of the compiler technology that can be improved by reducing idle cycles generated by memory latencies in order to extract more performance from a microprocessor [5]. As shown in [3], `load` instructions are the operations most often executed in most programs (about 20%). Moreover, the architecture of scalar processors like MIPS [9, ?] solves data dependency issues by using `load` delay slots (another solution consists in interlocking the pipeline—e.g., SPARC). This manner of dealing with data dependencies is tightly associated with the efficiency of the compiler (or assembler) that has to fill the `load` delay slots in reorder sections—the parts of the assembly code that can be handled by the assembler, as opposed to noreorder sections—by useful instructions. If it is not able to do so, a number of stall cycles are introduced depending on both the pipeline depth and the structure—e.g., $CPI_{load} = 2$ for a 5-stage pipeline and 3 for a processor with one more data memory stage, in the worst case for a cache hit. Obviously, this has an important effect on the performance of the system as shown by the non-reduced formula for CPU_{time} [3]—where the three factors mentioned in Sect. 1 are highlighted and the fraction models the frequency of instruction i in a program and IC_i its number of occurences with its associated CPI_i:

$$CPU_{time} = \left(\sum_{i=1}^{n} \frac{IC_i}{Instructioncount} \times CPI_i \right) \times Instructioncount \times Clock\ cycle\ time \quad (1)$$

Whenever a data dependence has been detected between a `load` instruction and the following instruction, GAS resolves data dependencies by inserting a `nop` operation in the assembly code after a `load` instruction. Otherwise, the code is kept unchanged. Because the GCC toolchain scheduler is architecture-independent thanks to an encapsulation process that extracts machine specific infllncs.clsormation [11], an architecture-dependent optimizer will help to exploit a given architecture in a more efficient way [13].

Even if many sophisticated schedulers have been proposed [14, ?], this work will show that improvements are sometimes still needed and can be performed thanks to a simple algorithm.

3 GCC-based Speed Optimizer

3.1 Principles

Our Optimizer was designed upon the following principles : the tool source code must be easily understandable, quickly modifiable and adaptable to other microarchitecture.

It is in fact a static instruction scheduler and more precisely a standalone machine-dependent tool that can be part of the compiler tool chain. Being machine-dependent allows us to be close to the targeted microarchitecture and then to have more chances of performing an efficient optimization. It has been designed to avoid instructions that are part of the code only to solve data dependencies thanks to NO Operation(`nop`): these non-productive instructions tell the CPU to do nothing while waiting for data to arrive—i.e., forbid the immediate access to a destination register before the correct value arrives. Indeed, a customized profiler running embedded benchmarks showed that many `nop` instructions could be removed.

Even if the compilation time for small programs such as the ones used in embedded systems is a secondary criterion and is beyond the scope of this paper, the optimizer has been written in perl. The delay to generate the compiled code, which increases with the program size, it is not significant at all, especially for the embedded programs we have studied.

3.2 The Algorithm

To avoid some of the aforementioned useless operations, our scheduler first introduces all necessary `nop` instructions inside the reorder sections of the assembly code, generated by a compiler— GCC in our case—to solve data, control and structure dependencies according to the targeted machine.

It then scans the generated assembly code in order to identify all `load-nop` instruction patterns inside any reorder sections. For each of them, it will look for an independent instruction that can be placed after the analyzed `load` instruction. The elected instructions are chosen from a window of three instructions located below and two above the memory instruction in focus—i.e., up to five instructions can be analyzed to `fill in` the future `load` delay slot as illustrated in Fig. 1 where the `nop` on line 3 can be removed. Extending this scope did not help significantly to find more useful instructions as using a window of one more instruction produced an improvement of only about 0.1%. Finally, all the added `nops`, added at the beginning of the process, are removed to let GAS provide the binary code using its own optimization routine.

3.3 Features

The optimizer has a modest code size (about 800 lines of perl) in comparison to the ten thousand of C lines that compose GNU C. It is based on the MIPS 1

```
lh      v1,0(t1)        lh      v1,0(t1)
lh      v0,0(t2)        lh      v0,0(t2)
nop                 =>  addiu   t2,t2,2
mult    v1,v0           mult    v1,v0
addiu   t2,t2,2         addiu   t1,t1,2
addiu   t1,t1,2
```

Fig. 1. GCC's options results vs our optimizer effect. The code, on the left side, was generated by the standard GCC and its options whereas the code on the right side, comes from the output of the optimizer.

ISA and then has been designed to improve the performance of the MIPS R3000 architecture.

It is another layer in between GCC and GAS that attempts to increase the benefits brought by the GNU tool chain. What follows summarizes some important features of the tool :

- It is compatible with all the standard GCC tool chain optimizations.
- It does not break the reliability and the robustness of the GCC tool chain because the GNU source code is not touched. Thus, the user program and the machine-dependent scheduler itself can still be easily debugged.
- The user has the choice of whether to use the optimizer quasi-instantaneously or not. This goal is achieved by the compilation flow we propose, shown in Fig. 2). Indeed, a compilation can be performed without the optimizer because it is not part of the GNU source code.
- The scheduler is based on the GAS algorithm to avoid re-inventing the wheel—i.e., we use the fact that the assembler already introduces `nops` after `load` instructions, according to the `load` delay slot length [8] whenever it finds a data dependency between a memory instruction and the next instruction inside a reorder section.

We expect to apply our approach to other scalar processors than MIPS such as ARM, ARC, Xtensa, PowerPC, SPARC, etc.

4 Experimental Methodology

This section describes the environment in which the results presented in Sect. 5 were obtained—i.e., the compiler toolchain and its configuration, the chosen processor and its models—and then, the benchmark used to validate our work. The behavioral model of the targeted processor has been at the origin of this project. Indeed, it helped highlight some relevant statictics that were exploited to create the optimizer.

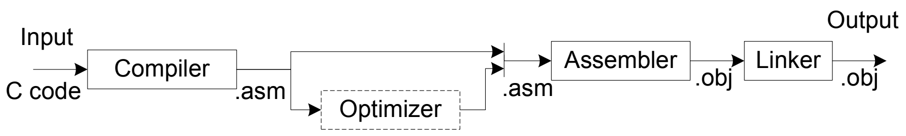

Fig. 2. Optimizer integration inside the compilation flow.

4.1 The Cross Tool Chain

In many system contexts, notably embedded ones, a cross tool chain may be necessary to compile a high level user code dedicated to run on a processor different from the host machine.
The cross tool chain we are talking about in this paper is based on GCC, a widespread free C compiler [15]. It is made up of six major components: the cross compiler itself (e.g., GCC version 2.95.2), an assembler (GAS) and a linker (GLD) based on various utilities (e.g., Binutils version 2.13), a library (e.g., newlib version 1.12.0) and a debugger (e.g., GDB version 6.0).

The standard compilation flow which consists in compiling, assembling and linking user programs is a bit different from the one we propose, as we will see Sect. 4.6.

4.2 Settings

Most of the time, physical size and cost are strong constraints in embedded systems and hence, the memory size is a fundamental factor that must be as small as possible. Moreover, as specified in [3], memory size has an impact on power consumption, which can be also a crucial factor in battery-powered systems. In this context, no loop unrolling optimization will be performed in the compilation steps of tested programs although great gains are also obtained.
All of the benchmarks were compiled with the `O3` and `Os` options: the former, tries to improve the program time execution whereas the latter tries to minimize the code size. Moreover, in order to avoid "relocation truncated" errors, the `-G 0` option was applied in all cases as specified in [10]. The libraries were generated in the same way. Two more runs, which included the optimizer this time, were also performed.

4.3 Reference Processor

The case study is a 5-stage single instruction issue, in-order RISC processor, implementing the MIPS R3000 architecture, in which a `load` instruction can be followed by any other instruction that does not use the destination register of that `load`. In this case, the `load` instruction can be considered as a 1-clock instruction, otherwise, two clock cycles are necessary to avoid data dependencies. This 32-bit load/store architecture does not contain any Floating Point capability and it is made up of 32 32-bit general-purpose registers.

4.4 VMIPS Modifications

VMIPS is MIPS R3000 simulator [2] that we have improved to make it compatible with any integer programs in order to validate our optimizer and the new toolchain through the EEMBC benchmarks [16]. Furthermore, it acts as a profiler by achieving data dependence analysis for `load` instructions to check whether an improvement may be considered and delivers many other statistics. It revealed weaknesses in GCC : indeed, it highlighted that many useless operations are executed after certain `load` instructions.

The analysis algorithm is the same as the one used by our optimizer (see Sect. 3).

4.5 EEMBC's Benchmarks

All of EEMBC's integer programs are used in this study—i.e., 42 out of 48 programs—to validate the new compilation flow and to demonstrate what gains this optimizer can bring. The application domains considered is this study are telecom, networking, automotive, office and consumer. The results described in Sect. 5 are based on what EEMBC calls the `regular` programs.

4.6 The Compilation Flow

Our approach keeps the standard cross tool chain unchanged, thus keeping the reliability, portability and optimization options of the GNU tools.
Indeed, the optimizer is a new step in the compilation flow that arises just after the assembly code generation as shown in Fig. 2. To do so :

- we first compile the program with GCC using the `-S` option in order to generate the assembly code (.S file),
- then we (can) run the optimizer on the produced assembly code. At this step, the code is re-scheduled.
- Then, the code is assembled using the `-o` option to generate an object file (.o file).
- The optimized object file can then be linked to other object files by the linker.

5 Results

In this section, we analyze and compare, for all of EEMBC's integer benchmarks, the performance of VMIPS used with the standard GCC tool chain and with the flow that takes our optimizer into account. We will see that the application has an impact on the speedup that the optimizer can bring. The chosen evaluation metric we worked on is the overall execution time. Indeed, thanks to the profiler, we can obtain the number of clock cycles needed to execute an integer program.

Only the results of integer EEMBC benchmarks that are improved by the optimizer are presented in this paper—i.e., 25 applications out of 42—others have a null gain and are not mentioned further.

In our simulations, the multiplication and division latencies were set to 5 clock cycles and 35 clock cycles respectively inside VMIPS, as suggested in [9]. We can then see the benefits brought by the optimizer on both the `Os` and the `O3` option in Fig. 3. For each benchmark, the simulation done with the `Os` option constitutes our speed reference which is modelled by the zero horizontal axis. Then, again for each program, three other simulations were achieved and compared to the suitable pre-cited reference.
The first one combines the `Os` option and our optimizer and indicates that the performance have increased with an average gain of 1.94%.
The second set of simulation wave takes only into account the `O3` option and illustrates an average speedup of 2.86% over `Os`. This gain is nearly doubled by running the optimizer in addition to the `O3` option in the last simulation group (`O3`-opt/`O3` average speedup of 2.64%).
The `O3` option combined with the optimizer mostly allows the best performance and in any cases, the optimizer does not deteriorate the gain of the `Os` or `O3` options.

The domain that takes the most advantage of the optimizer is the EEMBC telecom domain with a speed up of 3.85% for both `Os` and `O3` options whereas the EEMBC consumer benchmarks, not represented, received no improvement.

The best improvement is for a telecom application, autcor00data_2 (autocorrelation), that reaches 9.08% because the heart of this benchmark is based on two nested loops, made up of few instructions, in which a variable is written and read. This situation appears to be a difficult case for the GCC optimizer algorithm because it creates a `nop` to solve a data dependency that cannot be handled by `O3` probably due to a lack of instructions surrounding it. This `nop`, as part of the loops, is executed several times and it represents an important percentage of the performed instructions. Moreover, the input of this application has an impact on the number of iterations performed by these loops: the iterations are proportional to the input size. The biggest dataset is the one corresponding to the sine shape input signal (autcor00data_2) that is why it provides better results than its counterparts, the pulse (autcor00data_1) and the speech (autcor00data_3) input signal.

We can note a spectacular improvement of about 18.6% for the pntrch01 benchmark (Pointer Chasing). Although the compilation made with the `O3` already provides a great gain of 12.52%, the optimizer reaches a speedup of 18.6% and hence, a gain 6.08% higher.

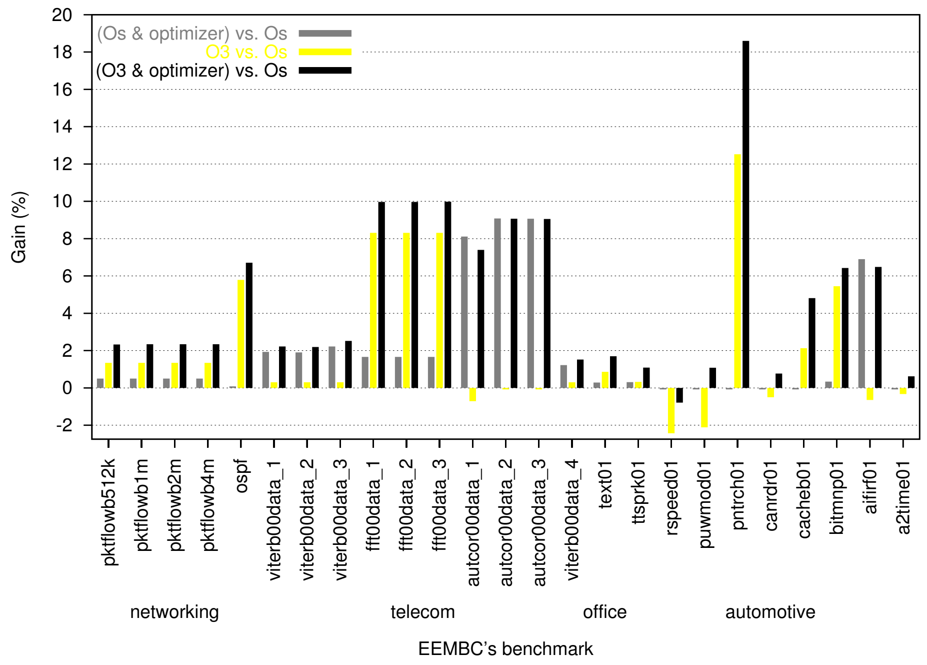

Fig. 3. Overall speedups brought by the optimizer for both `Os` and `O3` GCC options. Three columns are associated with each benchmark: the first one, in grey, represents the speedup that the `Os` compilation option combined with our optimizer brings upon only the `Os` compilation option. The second column, in light grey, shows the gain that the `O3` compilation option provides upon its `Os` counterpart. And finally, the last of the three columns, in black, shows the speedup brought by the `O3` and our optimizer always in comparison with the `Os` GCC option. The best improvement reaches 9.08% for the $autcor00data_2$ telecom application whereas about 40% of the EEMBC integer programs obtain no speedup and are not reported in this figure.

6 Conclusion and Future Work

We have developed a profiler based on a modified version of the VMIPS simulator, which is now benchmark compatible, in order to evaluate the potential speedup that can be achieved on the GCC tool chain using the EEMBC benchmarks.

Furthermore, this work highlights some types of embedded applications for which the performance can be increased by up to 9% for a telecom application running on a MIPS architecture.

As a consequence, energy is saved because there is less switching activity due to the removed `nops` and low power techniques—e.g., based on the power supply and the clock activity—could be applied to take advantage of the shorter execution time.

The scheduler is based on a very simple algorithm and the complexity of its implemention is relatively low (about 800 lines of perl code). We have shown that significant speedup can easily be obtained thanks to an independent machine specific tool, at no cost, without changing the reliability, the robustness, the compilation time of the GNU tool chain and preserving all of its debugging information.

A similar optimizer can be achieved for other load/store microarchitectures and different ISAs in which the same weakness, based on unfilled `load` delay slots, has been identified.

Even if the proposed optimizer is well suited for scalar processors dedicated for embedded applications, it could be applied for other processor architectures using delay slots and even interlocks. Indeed, much research has been done on parallel processing—i.e., how to use several ALUs or CPUs simultaneously—based on static [6] or dynamic [1] approaches. But even if this partitioning is well done [7], an issue is still there: the memory access latency that is associated with the system structure, notably memory one, pipelines structures. In cache hit cases, some clock cycles may be needed to avoid data, control or structure dependencies, which cause a loss of performance.

Moreover, the profiler has also identified that branch delay slots still need to be managed, which could lead to a new version of the optimizer.

7 Acknowledgments

We would like to thank the reviewers for their helpful and detailed comments, Markus Levy for using the EEMBC's benchmarks and the Swiss Commission for Technology and Innovation (CTI) for their support.

References

1. M. Epalza, P. Ienne, D. Mlynek, *Adding Limited Reconfigurability to Superscalar Processors*, in Proc. of the 13th Int'l Conference on (PACT'04) Parallel Architecture and Compilation Techniques, October 2003.
2. B. Gaeke, *The VMIPS Project*, Version 1.1.3, http://www.dgate.org/vmips
3. J. L. Hennessy, D. A. Patterson, *Computer Architecture : a quantitative approach*, Morgan Kaufmann Publishers, Inc., Third edition, 2003.
4. G. Kane, J. Heinrich, *MIPS RISC Architecture*, Prentice-Hall, Inc., New Jersey, 1988.
5. M. Levy, *Breaking Embedded System Barriers - Embedded Developers Discuss the Trends to Improve Performance*, Embedded Processor Forum, August, 2003.
6. A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, R. Guerrieri, *A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications*, in ISSCC Digest of Technical Papers, February 2003, pp. 250-251.

7. E. M. Panainte, K. Bertels, S. Vassiliadis, *Compiling for the Molen Programming Paradigm*, in Proc. of the 13th Int'l Conference on Field-Programmable Logic and Applications (FPL), vol 2778, Springer-Verlag Lecture Notes in Computer Science (LNCS), September 2003, pp. 900-910.
8. R. Stallman, *Using and Porting GCC*, Technical Report Documentation GNU, November 1995.
9. D. Sweetman, *See MIPS run*, Morgan Kaufmann Publishers, San Francisco, 1999.
10. D. Sweetman, N. Stephens, *IDT R30xx Family Software Reference Manual*, IDT Manual, 1994.
11. M. D. Tiemann, *The GNU Instruction Scheduler*, Technical report, Free Software Foundation, July 1989.
12. A. Unger and E. Zehendner, *Tuning the GNU Scheduler to Superscalar Microprocessors*, in 23rd EUROMICRO Conference '97 New Frontiers of Information Technology, September 1997.
13. L. Wang, B. Lu, L. Zhang, *The study and Implementation of Architecture-dependent Optimization in GCC*, Proc. of the Fourth International Conference/Exhibition, 2000.
14. K. Wilken, J. Liu, M. Heffernan, *Optimal instruction scheduling using integer programming*, ACM SIGPLAN' 00 Conference on Programming Language Design and Implementation (PLDI), June 2000.
15. GCC home page - cross compiler tool chain, http://gcc.gnu.org
16. The Embedded Microprocessor Benchmark Consortium,
http://www.eembc.hotdesk.com/home.jhtml