An Optimizing C Compiler for the GMICRO/500 Microprocessor

Yugo Kashiwagi†1
Yasuhiro Tawarat2
Hideaki Chakit3
Kouji Yamadat2
Masahiro Kainagat2
Tatsuo Isobet3

†1 Semiconductor Design and Development Center, Hitachi, Ltd.
Kodaira, TOKYO JAPAN
†2 System Development Laboratory, Hitachi, Ltd.
Kawasaki, KANAGAWA JAPAN
†3 Hitachi Software Engineering Co. Ltd.
Yokohama, KANAGAWA JAPAN

Abstract
The GMICRO/500 is a superscalar microprocessor based on the TRON specification. An optimizing C compiler for the microprocessor is now under development.

The compiler implements both hardware-independent optimizations and GMICRO/500-specific optimizations with a performance target of 100 MIPS at 50 MHz.

The hardware-independent optimizations gave a good basis for the effective improvement of hardware/software performance. On the other hand, GMICRO-specific optimizations was a challenging problem because of the complexity of the superscalar CISC architecture.

This paper describes these optimization techniques with emphasis on the implementation strategy of the GMICRO/500-specific superscalar optimization.

1. Introduction
The GMICRO/500 is a superscalar microprocessor based on the TRON specification [1]. An optimizing C compiler for the GMICRO/500 is now under development based on the GMICRO/200 optimizing C compiler.

Since the first publication of the paper on the GMICRO/200 C compiler [2], the authors have implemented more sophisticated optimizations, resulting in 98% performance improvement (4.7 MIPS to 9.3 MIPS). The current version of the GMICRO/200 C compiler is Version 3.0.

The design of GMICRO/500 optimizing C compiler started with this GMICRO/200 optimizing C compiler. This is an important point both in architecture and compiler design. A good optimizing compiler, implementing all the machine independent optimizations, leaves essential points to be implemented by the hardware and machine dependent optimizations of the compiler.

Running the object code of the GMICRO/200 C compiler on the GMICRO/500 instruction simulator gives the performance of 71.3 MIPS at 50 MHz (1 MIPS=1757 dhrystone loops/sec). An improvement of 150% is achieved by clock speed (20 MHz to 50 MHz), and 207% by the parallelism of GMICRO/500 superscalar architecture.

Analysis of the benchmark programs has shown that the most effective performance improvements can be achieved by reducing the overhead of procedure calls and maximizing the parallelism of GMICRO/500 Superscalar architecture. These two issues are the main target of hardware/software performance improvement.

This paper describes hardware independent optimizations in GMICRO/200 C compiler and GMICRO/500 specific optimizations in GMICRO/500 C compiler. The former shows the starting point of the authors' development, and the latter, superscalar optimizations which can also be applied to superscalar CISC architectures in general.

2. Hardware independent optimizations
This section describes the optimizations implemented in the GMICRO/200 C compiler and its refinements to be implemented by the GMICRO/500 C compiler.

The current version of the GMICRO/200 C compiler implements all the known optimizations described in standard compiler textbooks [3]. In addition, the compiler performs global register allocation, and generates instructions tuned for the the GMICRO series instruction set.

Stripping off almost all the redundancies of source programs, the optimizing C compiler reveals the essential operations to be improved, giving a good starting point for both architecture and compiler design. The following section describes the optimizations already implemented in the GMICRO/200 optimizing C compiler.

2.1 Optimizations of GMICRO/200 C compiler
The purpose of the hardware independent optimizations is to strip off the redundancies of the source program, sometimes intrinsic in the language specification. The effective optimization techniques are as follows:
(1) Constant folding
(2) Common expression elimination
(3) Copy propagation
(4) Elimination of redundant expressions
(5) Loop unrolling
(6) Strength reduction of loop counter expression
(7) Use of algebraic laws
(8) Tail recursion

These hardware independent optimizations, together with global register allocation and GMICRO-specific optimal code generation, achieved 98\% of the performance improvement from the first version of the compiler on GMICRO/200, evaluated by Dhrystone Benchmark Version 1.1[4]. This gives the performance of 71.3 MIPS on GMICRO/500.

The Dhrystone benchmark is used to measure the effect of new optimizations for GMICRO/500 described in the following sections. More realistic benchmark programs, such as SPEC benchmarks [5] would be appropriate to measure the system performance. But at this early stage of development, where there is no working compiler for GMICRO/500, the measurement with SPEC benchmarks is expensive.

2.2 Refinements of hardware-independent optimizations

The study of the object code of GMICRO/200 C compiler has shown that the overhead of procedure calls is still a major source of potential performance improvement. This is also an important issue from the standpoint of software engineering. Low overhead procedure call encourages the modularization of programs, which improves the productivity of software systems.

GMICRO/200 C compiler already implements tail recursion optimization, parameter passing through registers, and optimizations of leaf procedures. However there are still more optimizations possible.

A part of the problem is solved in GMICRO/500 hardware by speeding up JSR instructions, and implementing a return address cache. The compiler implements the following two new optimizations for GMICRO/500.

(1) Extended tail recursion

Tail recursion optimization described in the last section is not so effective for real-world embedded programs, because recursive programs, which use an indefinite amount of stack, are not common in embedded systems.

The new compiler extends the tail recursion so that it can handle the calls of other functions at the tail position, which are common in embedded systems.

Example.

```
f() -> f()
{
    g();
    goto g;
}
```

The above example is not a valid C program, but it illustrates the basic idea of extended tail recursion. This conversion eliminates the save/restore of return address generated by the call of "g". This optimization when parameters are specified provided that they are allocated on registers.

(2) Inline expansion

Short procedures are expanded in-line into their caller. To avoid program size explosion, inline expansion is applied only to very short procedures without further procedure call.

Adding these optimizations, the object code of the GMICRO/200 C compiler can run at 83.7 MIPS at 50 MHz on GMICRO/500, which is 17\% performance improvement. This performance gain includes the reduction of procedure call overheads, and optimization opportunities newly introduced by inline expansion.

3. GMICRO/500 specific optimizations

This chapter describes hardware-dependent optimizations for the GMICRO/500. The detailed description of the GMICRO/500 architecture, which resulted in the 207\% speedup of the GMICRO/200 C compiler object code is described in [6].

The hardware-dependent optimizations are done by the following steps:

(1) Register allocation
(2) Instruction selection
(3) Branch optimization
(4) Instruction scheduling

Among these tasks, (1) and (2) are almost solved in existing GMICRO/200 C compiler, because they apply to the GMICRO series architecture in general instead of specifically to GMICRO/500 architecture.

However, the following compiler parameters for these tasks are adjusted to promote superscalar parallelism:

(a) Penalty for the allocation of fresh registers
The use of a fresh register is encouraged to avoid register conflict.

(b) Preference for simple instructions
The preference for selecting simple instructions is promoted, because simple instructions have more potential to be executed in parallel.

The optimizations (3) and (4), making full use of GMICRO/500-specific hardware resources, are the major source of performance improvement. Following sections discuss them in detail.

3.1 Branch optimization

Branch optimization makes use of two hardware resources of the GMICRO/500: the instruction prefetch queue and the branch window.

The instruction prefetch queue prefetches instructions by 64 bits, aligned at 8-byte boundary. To make the most of this hardware, the compiler aligns the target labels of BRA and JSR instructions at 8-byte boundaries. This minimizes the number of instruction fetches after an unconditional branch.
The branch window is the hardware mechanism which executes unconditional branch instructions (BRA and JMP) in 0 cycle when the branch window is hit. The branch window is almost always hit if the program is executing the innermost loop. So it is safe to assume that unconditional branch instructions takes 0 cycle in the compiler optimization.

Given this fact, the branch optimization appears quite different compared to branch optimization of the GMICRO/200 C compiler. The following is an example to illustrate this:

Example:

Source program

```c
while (i<10){
    s+=i;
    i++;
}
```

Object program by GMICRO series C compiler

```assembly
BRA L2
L1:
    ADD @I, @S
    ADD #1, @I
L2:
    CMP #10, @I
    BLT L1
```

Object program by GMICRO/500 C compiler

```assembly
L1:
    CMP #10, @I
    BGE L2
    ADD @I, @S
    ADD #1, @I
    BRA L1
L2:
```

Suppose the initial value of "i" is 0. Then the first object executes 1 unconditional branch, 10 conditional branches taken, and 1 conditional branch not taken. The second executes 10 unconditional branches, 1 conditional branch taken and 10 unconditional branches not taken. The total number of instructions mattered in the GMICRO/200 C compiler. But in GMICRO/500, the number of conditional branches taken, which stops the pipeline flow, really matters. So the compiler minimizes the number of conditional branches taken by introducing extra 0-cycle unconditional branches for compensation.

These two optimization, 8-byte alignment of labels and generation of loop instructions assuming 0-cycle unconditional branches, minimizes the overhead of pipeline hazards caused by branch instructions.

3.2 Instruction scheduling

The GMICRO/500 architecture, being a CISC, makes the problem of instruction scheduling more complex than ordinary RISC processors. The compiler implements three ideas to solve this problem.

The first idea is to classify instructions by their grain size. Instructions of different grain sizes are scheduled at different stage of the scheduling algorithm, because instructions of larger grain size are always preferred to gain parallelism.

The second idea is to trace the assignment of pipe (0 or 1) to instructions during instruction scheduling for accurate pipeline simulation.

The third idea is to simplify the pipeline simulator by adopting simpler pipeline model.

The following sections describe these ideas in detail.

3.2.1 Granularity of instruction parallelism

The granularity of instruction parallelism corresponds to the following three levels of hardware parallelism:

1. ALU/FPU parallelism
   Integer instructions are executed by the arithmetic logical unit, and floating point instructions are executed in the floating point unit. These two units execute in parallel. FPU instructions, after it is decoded, occupies FPU typically for 10 cycles.

2. CPU/memory parallelism
   Memory interface executes in parallel with the central processing unit. When a datum is stored into memory, subsequent instructions can be executed in parallel with the store operation.

3. Superscalar parallelism
   Superscalar pipeline executes two instructions in parallel.
   Among the parallelisms above, (1) has the coarsest granularity and (3) has the finest. The compiler gives the highest preference for the floating point instructions with the coarsest granularity, then store instructions, and if none of these can be found, it schedules instructions with the finest granularity. When the hardware with the coarser grain size is busy, the instructions for the hardware is given lower preference.

   For example, the floating point instruction FADD takes 3 cycles to execute after it is decoded by the CPU decoder. When FADD and other CPU instructions are to be scheduled, the compiler schedules FADD first. Then the FPU becomes busy for the next 3 cycles, and the compiler schedules CPU instructions for this period.

3.2.2 Behavior of GMICRO/500 superscalar pipeline

Before going into the detailed description of the pipeline scheduling of the GMICRO/500 C compiler, this section shows the typical behavior of the GMICRO/500 superscalar pipeline from a standpoint of compiler implementation. The pipeline has two pipes (pipe 0 and pipe 1), each of which has five units: instruction fetch (I), decode (D), execute (E), access (A) and store back (S). An instruction goes through these five stages to be executed.

Figure 1 shows the execution model of the GMICRO/500 superscalar pipeline at peak speed, in which
each instruction takes only 0.5 cycle on average. Each row is the execution of a single instruction, and each column is operations performed in a cycle.

(2) Data Conflict
When a datum is set to a register, it is not available until the store back stage. Figure III shows an example of data conflict. When a datum is loaded to a register, it is not available until the store back stage. So the second instruction is delayed by 3 cycles until the datum is loaded to the register.

Figure I. GMICRO/500 pipeline execution model

There are three major sources of pipeline hazard and one hardware mechanism to reduce them:
(1) The timing of decode
The superscalar pipeline of the GMICRO/500 decodes two instruction in parallel, provided that the first instruction is a 2-byte instruction. This is not always the case, because the GMICRO/500 has a CISC instruction set. If the first instruction has the length longer than 2 byte, only the first instruction is fed into the pipeline. This is one of the reason why the code generator is designed to issue shorter instructions if possible.

This mechanism of simultaneous decode has a problem of committing the execution of the second instruction. Once an instruction is decoded in pipe 1, the instruction commits its execution in pipe 1 even if the execution of the first instruction ends in pipe 1.

Figure II shows this problem. Two instruction, both of 2-byte length are decoded simultaneously. But the first instruction occupies two pipes. The second instruction, which has already committed to execution in pipe 1 cannot start its execution in pipe 0 of the next step, leaving one empty pipe.

(3) Access Conflict
When pipe 0 and pipe 1 try to access memory in the same cycle, the instruction in pipe 1 is delayed by 1 cycle. Figure IV shows an example of access conflict.

(4) Register bypassing
GMICRO/500 architecture provides a mechanism to reduce the delay caused by data conflict, which is the greatest source of the pipeline hazard. The datum, still on the way to be stored back into the register, is bypassed to the next instruction at earlier stage. This is called register bypassing.

The data size of the operand in the second instruction must be long-word size, because the bypassing mechanism does not include type conversion.

Figure V shows an example of register bypassing. Register R1 is not set at the execution stage of the
instruction. But the hardware bypasses its value so that the next instruction receives it at earlier stage.

If the first instruction loads the datum from the memory, the bypassing occurs in the access stage of the first instruction, where the value of the operand is available, resulting in the delay by 2 cycles.

3.2.3 Pipeline scheduling

The algorithm of pipeline scheduling has been already implemented in the GMICRO/200 C compiler [2]. The compiler constructs a dependence graph of instruction sequence in each basic block (a sequence of instructions without jump-ins and jump-outs), and then rearranges the instruction order to minimize the execution time of the basic block, whilst retaining the dependence among instructions. A survey of the techniques of instruction scheduling, especially for superscalar microprocessors can be found in [7]. The GMICRO/500 C compiler implements the scheduling based on greedy algorithm, in which locally optimal instruction is preferred for the next candidate.

To apply pipeline scheduling algorithm to GMICRO/500, the compiler must know the assignment of pipes to the instructions. Figure VI illustrates this problem.

Consider two instances of register bypassing in the figure. In (a), where the first instruction starts at even pipe (pipe 0), there are two pipes left open. In (b), where the first instruction starts at odd pipe, no pipes are lost. This shows the importance of the assignment of parity (even or odd) of pipes to the instructions.

The assignment of pipes is of dynamic nature, and cannot be determined completely at compile time. The following example illustrates the situation:

**Example.**

MOV.W #10,R1
L1:
SUB.W #1,R1
...  
CMP.W #0,R1
BNE L1

**Figure V. Register bypassing**

**Figure VI. Assignment of pipes**

Suppose the execution of the first instruction of the example above (MOV) ends in pipe 0. The second instruction (SUB) is executed in pipe 1. On the other hand, when the SUB instruction is executed after the BNE instruction, the instruction prefetch queue is reset and the SUB instruction is executed in pipe 0. So we cannot determine the pipe in which the SUB instruction is executed.

In such cases, it is safe to assume that the SUB instruction is almost always executed in pipe 0, because the SUB instruction is almost always executed after the BNE instruction, because this is a loop.

In general, the compiler uses the following rules to determine the parity of pipes.

1. **Beginning of a procedure**
   The JSR instruction always ends its execution in pipe 1. So the compiler assumes that the first instruction of a procedure starts in pipe 0.

2. **Inside a procedure**
   Inside a procedure, the compiler performs flow analysis to determine the parity of the first instructions of each basic block. This determines the starting pipe of each instruction except the merging points of control flows.

3. **Merging points of control flows**
   If there is a merging point of control flow and the last pipe of the preceding basic blocks disagree, the parity of the following basic block is not statically determined. The compiler takes the control flow which jumps to the point (which ends in pipe 1, in most cases), because this flow is more frequent as shown in the last example.

The parity of pipes introduces a new scheduling technique, NOP insertion. The behavior of pipelines described in the last section shows that the execution time of the instructions can be improved by inserting NOPs...

67
and adjusting the timing of decode of instructions. This technique can be applied to eliminate both data conflicts and access conflicts. Of course, it would be better if these conflicts can be eliminated by just exchanging instructions without inserting NOPs. This technique is applied by the compiler when all the other tries fail.

**Figure VII** shows the effect of NOP insertion for the data conflict. In the first case, the second instruction, commits the execution in pipe 1, is delayed by the data conflict, and executed in the next pipe 1. In the second case, where a NOP is inserted, the second MOV instruction, decoded in pipe 0, opens the second pipe 1 for the next instruction.

(a) Before NOP Insertion

| MOV R0, R1 |
| MOV R1, R2 |

(b) After NOP Insertion

| MOV R0, R1 |
| NOP |
| MOV R1, R2 |
| Next Instruction |

**Figure VII. NOP insertion**

### 3.2.4 Implementation of the pipeline simulation

To implement the pipeline scheduling in the last section, the accurate simulation of the pipeline behavior is the most important. This can be done by the canonical simulation of the five stages of the pipeline, which is adopted by the simulator of GMICRO/500. But the cost of simulating all the pipeline resources is too much for the compiler, because every instruction generated by the compiler goes through this process. The variety of CISC instruction set makes this problem more serious.

When the compiler examines the next instruction for scheduling, it requires only one datum, the delay resulting from selecting the instruction. To implement this efficiently, the authors devised a one-stage model of the instruction scheduling (**Figure VIII**). The one-stage model is, so to speak, a slice of the five-stage model at the decode stage. The reference point of each instruction should be the decode stage, where the execution of each instruction is committed.

As far as the calculation of the delay time is concerned, these two models are logically equivalent and there is no loss of the accuracy of simulation. The one-stage model is an encoded version of the five-stage model tailored for the instruction scheduler.

The status of pipeline is encoded into the following data:

1. M: The timing of the newest memory access.
2. Cn: The timing of the newest change of register, Rn.
3. P: The pipe for the next instruction.

For each instruction, the behavior of the instruction is encoded into the following data:

1. L: The number of pipes occupied for the execution of the instruction.
2. Rn: The timing of the read of register, Rn.
3. Sn: The timing of the write of the register Rn.
4. N0: The timing of the first memory access.
5. N1: The timing of the last memory access.

Essentially, these data are sufficient to give the precise calculation of the delay time. All the pipeline behavior in the section 3.2.2 can be simulated with these data.

As an example, the simulation of NOP insertion described in the last section, is simulated in one stage model in **Figure IX**.
Figure IX. Sample simulation using one stage model

(a) Before NOP Insertion

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
\text{MOV R0,R1} & \text{MOV(1)} & \text{MOV(2)} & \text{MOV(3)} & \text{MOV(4)} \\
\text{MOV R1,R2} & \text{NOP} & \text{NOP} & \text{NOP} & \text{NOP} \\
\end{array}
\]

delay=1 cycle

(b) After NOP Insertion

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
\text{MOV R0,R1} & \text{MOV(1)} & \text{MOV(2)} & \text{NOP} & \text{NOP} \\
\text{MOV R1,R2} & \text{NOP} & \text{NOP} & \text{NOP} & \text{NOP} \\
\end{array}
\]

Figure IX shows the performance improvement of the compiler.

4. Conclusion

The GMICRO/SOO C compiler, implementing all of the optimizations described here is now under development. The Dhrystone Benchmark Version 1.1 [4] compiled with the above optimizations except pipeline scheduling shows 300 cycles/loop, which is equivalent to 94.9 MIPS at 50 MHz (1 MIPS=1757 Dhrystone loops/sec). The effect of pipeline scheduling is not evaluated yet, because its precise evaluation requires the full implementation. There is still 5% room for improvement by the pipeline optimization, as the object code scheduled by hand shows the performance of 100 MIPS at 50 MHz, which can be reached by globally optimal pipeline scheduling algorithm. It is yet to be shown how far from the optimal the authors' strategy is. Figure X shows the performance improvement of the compiler.

Bibliography