Compiling for the CRISP Microprocessor

Sumit Bandyopadhyay
Vimal S. Begwani
Robert B. Murray

AT&T

190 River Road, Summit, NJ 07901

ABSTRACT

In this paper we present code generation considerations for the AT&T CRISP microprocessor. The architecture of CRISP is different from either current CISC or RISC machines in a number of ways. This opens the opportunity for several new types of optimizations. The implications for C language compilation will also be discussed.

1. Introduction

We have implemented a prototype optimizing compiler for the new AT&T CRISP[1] microprocessor. This optimizing compiler implements the most common traditional optimizations, plus several optimizations which are specific to the CRISP chip. In this paper we discuss some code generation and optimization issues for CRISP and present some results to show the effectiveness of these optimizations.

The rest of the paper is organized as follows: Section two gives a brief description of CRISP microprocessor. Section three describes in detail the specific optimizations we have implemented. Section four gives some implementation details. Some benchmark results are presented in section five. The implications of these optimizations on compilation are discussed in section six.

2. CRISP Architecture

CRISP is a high performance next generation microprocessor chip being developed at AT&T[1]. It has a simple instruction set (compared to today's CISCs), and supports simple addressing modes. It is designed to support modern block structured languages by providing a fast procedure call/return mechanism.

CRISP has no programmer visible general purpose registers *per se*; instead, the top 128 bytes of the stack are kept in an on-chip Stack Cache.[2] This cache is managed using a callee-save, caller-restore mechanism. Upon entry, each function allocates space in the cache for its stack frame; this allocation may cause data already in the stack cache to be flushed to memory. When the function returns, the calling function usually executes a *catch* instruction; if a part of the caller's stack frame has been flushed, this instruction reads that part back into the cache.

CRISP is a highly pipelined machine. Instructions are processed by two basic units, a prefetch and decode unit (PDU) and an execution unit (EU). The two units run asynchronously. There is an on-chip 512 byte prefetch buffer cache. There is also a 32 entry decoded instruction cache, which sits between the PDU and EU.

CRISP handles conditional branch related delays differently than other pipelined machines. Each conditional branch has a static *prediction* bit (set by the compiler) associated with it. This prediction bit tells the PDU and the EU the most likely outcome of that conditional branch. The PDU prefetches along this path and decoded instructions are loaded into the instruction pipe according to this suggestion. If the branch was predicted incorrectly, there is a delay while the pipeline is flushed and restarted. This scheme puts the onus of branch prediction on the compiler.

More details about CRISP chip can be found in [1], [3] and [4].

3. Target Optimizations

Before designing a prototype compiler, we identified the optimizations that looked most useful. The implemented optimizations can be divided into two sets. The first set contains most traditional optimizations which are machine independent. We will not discuss these optimizations in detail; further explanations of these optimizations can be found in [5]. The second set contains optimizations which are

specific to the CRISP microprocessor.

3.1 Traditional optimizations

3.1.1 Local common subexpressions: We identify common subexpressions within basic blocks; these are only evaluated once. These are often address calculations.

3.1.2 Constant Propagation/Constant Folding: This involves keeping track of expressions that have constant value, and doing the constant arithmetic at compile time.

3.1.3 Jump optimizations: A jump to a jump can be rewritten as a single jump. A jump to the next instruction can be eliminated.

3.1.4 Loop Invariant Code Motion: A computation which gives the same result on each successive execution of the loop is moved outside of the loop, where it will only be executed once.

3.1.5 Dead Store/Dead Code Elimination: Stores into memory locations that will never be used are eliminated.

3.2 CRISP-Specific Optimizations

3.2.1 Branch Prediction: In a pipelined machine, branch instructions tend to interrupt the smooth flow of instructions through the instruction pipeline. The penalty for an incorrect prediction is a flush of the pipeline, which can cost up to three cycles on CRISP. Since conditional branches can account for up to a sixth of all instructions executed, [6] [7] predicting the most likely path for conditional branches is one of the most important optimizations in the CRISP compiler.

A number of hardware strategies to predict the most likely outcome of conditional branches have been employed in other highly pipelined machines[8] [9]. On CRISP, branch prediction is done by the compiler, not the hardware. Our current branch prediction strategy uses the context of the comparison (e.g. is this the test of a *for* loop), the comparison operator (e.g. equal to, less than), and the C language types of the operands. (Note that the information about the loop type, and the C operand types, would not be available if we were using a hardware prediction scheme.) We instrumented compiled code for several UNIX® system programs (nroff, diff, pr, etc.) and analyzed these data to determine which type of branches were usually taken and which were not.

Based on these data we found that branches that are based on loop tests should be predicted to stay in the loop (which is what we had expected). Branch prediction for an *if* statement is a little more complex and involves a table lookup (based on the comparison operator and the comparison operand types) to predict if the body of the *if* will be executed or not.

This simple scheme gives good results for most programs we have run. For the UNIX system programs we have tried, the above strategy correctly predicted 90% of all the conditional branches.

We believe this demonstrates that the branch prediction by the compiler is effective. A 90% success rate compares favorably with the success rates of hardware branch prediction schemes,[8][9] and has the advantage of not requiring any space on the chip for branch prediction logic.

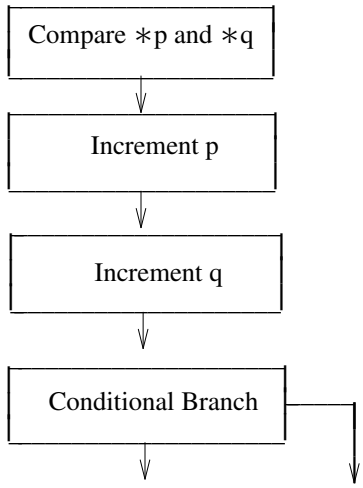
3.2.2 Branch Spreading / Instruction Scheduling: The instruction scheduler has two goals. First, it tries to separate a compare and its corresponding conditional branch instruction as far apart as possible (this is called *branch spreading* in the rest of this paper). Branch spreading can reduce or eliminate the penalty when the conditional branch was predicted incorrectly.

The compare instruction is the only instruction that sets or perturbs the condition code. Therefore, as soon as the EU has finished executing the compare instruction, it knows whether any of the conditional branches that have already been fetched were predicted incorrectly. If a branch has been predicted incorrectly then the pipeline after the branch is flushed. The advantage to putting code between the compare and the branch is that there will be less code in the pipeline after the branch, therefore fewer stages of the pipeline will be flushed.

This optimization is most commonly used by code that uses the C postfix increment and decrement operators. For instance, consider the following fragment of C code:

```
if (*p++ == *q++)
```

Here, the two increments must be done regardless of the outcome of the branch, so they can be put after the compare:



Here, the EU will have executed the compare when the conditional branch is in the first stage of the pipeline. On the next cycle the EU knows whether the prediction was correct; if it was incorrect the instruction after the conditional branch (which was fetched on the same cycle) is the only one that is flushed. Because of the spreading, the penalty for an incorrect branch prediction is one cycle instead of three.

If there is enough code between the compare and the branch, the EU may have finished executing the compare by the time it prefetches the branch. In this case, the EU will look at the actual value of the condition code to determine which way the branch will go (it can ignore the branch prediction bit).

Secondly, the instruction scheduler tries to put multiple references to the same variable as close as possible to each other. This strategy is opposite of the one used by [10]. In most pipelined architectures, if an instruction sets a memory location which is referenced in the next few instructions, then there is a data hazard. Some processor implementations do not have data hazard bypass, and therefore the compilers for these machines try to separate an assignment to, and the references of, the same symbol as far apart as possible. Since CRISP has data hazard bypass, we can put assignment and references to a symbol as close as possible. By *creating* a data hazard, the second reference is resolved by the data hazard bypass logic on the CRISP chip. This process, known as *read cancelling*,[4] avoids a memory reference.

Collecting all references to a symbol as close as possible also helps stack compression (see next section); if we can reduce the range over which an automatic is live, we may be able to put more symbols into a given stack location.

The instruction scheduling algorithm assigns a priority to each instruction. The highest priority is assigned to a compare instruction. Instructions generating values being used by a compare instruction are given the next highest priority. Branches are given the lowest priority. Instructions with the same priority are arranged such that all the references to each symbol are put close together.

3.2.3 Stack compression: In the current CRISP implementation, only the top 32 words of the stack are in the stack cache. We try to bind automatic variables to locations in this cache in a way that maximizes the percentage of references that are in the stack cache. This process is akin to register allocation on most machines. Stack compression has multiple benefits:

- By maximizing the number of symbols in the stack cache, we reduce the program execution time (references to the cache are faster than references to memory).
- A smaller stack frame reduces the cost of stack flushes/reads during procedure call/returns.
- A smaller stack frame reduces the cost of saving the stack cache on a process context switch.

The stack compressor achieves these goals by attempting to reuse stack locations whenever possible. For example, in CRISP, the stack frame layout requires the allocation of space for all of the outgoing parameters on function entry. However, between two function calls, this space can be used for local variables and temporary variables.

The stack compressor does live/dead analysis on all non-static symbols (compiler-generated temporaries, automatics, input parameters, and outgoing parameters). Using live/dead information, all the dead stores are eliminated. After that, it uses sharing information to merge (i.e. assign the same stack location to) symbols. For example, if we have *a = b;* and *a* and *b* are never live at the same time, then we can put *a* and *b* in the same place, and eliminate the assignment.

Information as to when sharing two symbols is legal, or is especially valuable, is specified by the

templates in an architectural description file used by the code generator.

Sharing the same stack location for two symbols to avoid a move instruction or to reduce the number of machine instructions is called *preferred sharing*. Such symbols are merged before merging any other symbols.

After eliminating dead stores, and merging preferred symbols, all the stack symbols are divided into three separate lists. The first list contains all the symbols that have some fixed offset (outgoing parameters), the second list contains all the input parameters, and the third list contains all the remaining symbols. Symbols in the first list are arranged such that the accumulator (one word above the top of stack) is the first symbol in the list; this is because the accumulator is an implied destination for many instructions. Next, we try to merge as many symbols as possible from the third list into either the first list or into the second list (in that order). All the symbols which are still left in the third list are then merged with each other as much as is possible.

The problem of putting all the automatics in the fewest possible stack locations is similar to a bin packing problem, which is NP-complete. Therefore, our current implementation uses a simple heuristic. We arrange all the symbols by the size of the live range (symbols that are only live in a small region come first). After that we try to merge symbols in one pass. After merging, all the symbols are arranged such that most frequently used scalars are put at the low end of the stack frame (i.e. on the top of the stack) and automatic aggregates are put at the high end of the stack frame.

In benchmarks tested so far, the above heuristic has given satisfactory results; our test cases show a runtime benefit ranging from 0-30% from stack compression.

Since the stack pointer moves in quadword increments (16 bytes), we only get a benefit if we change the number of quadwords in a stack frame. Reducing a stack frame size from 30 to 20 bytes gains nothing (we still need two quadwords); reducing a stack frame size from 18 to 16 gains a lot (we reduce the actual stack frame by half).

3.2.4 Catch compression: The *catch* instruction is normally executed after a called function returns; it reads the current stack frame back into the stack cache (if it has been flushed). Sometimes we can omit this stack cache restore. For example:

```
f(){
        int a,b,c;
        ...
        g();
        return;
};
```

In this function, we do not need to do a catch after the call to *g*, since *f* is going to return without accessing its stack frame.

Similarly, if there are only one or two references to the current stack frame before the next function call or return, then we can eliminate the catch. Without a catch, we may be referencing memory instead of the stack cache, but the extra cost of the memory references will be less than the cost of executing a catch.

In CRISP the stack starts from a high address and grows downwards. If only the low end of the current stack frame is referenced before the next function call or return, then we can shrink the catch size and read only the relevant part of the stack frame into the cache. (This is a good reason for putting the most frequently used scalars at the low end of the stack frame.) For example:

```
f(){
     int big_array[100];
     int i;
     ...
     i = g();
     ... /*Use i several times*/
     f();
    /* Use big_array*/
    }
```

After the call to *g*, we can catch *i* without catching *big_array*.

For each call in the function, we look ahead and figure out which part of the stack frame is used between this call and the next call (or return). We use this to compute the size of the catch that will occur after the call.

Currently, we are using a simple catch compression algorithm. We delete catches that are followed immediately by calls or returns. For other catches, we do some lookahead (only in the same basic block) to see which parts of the stack frame are accessed. If we find a call or return in the same basic block, we only catch the part of the stack frame that was accessed since the last call.

Otherwise, we catch all of the scalars. We never catch automatic aggregates.

3.2.5 Side stack: Large automatic aggregates can take up most (if not all) of the CRISP stack cache. It probably pays to move these to a separate stack (called a *side stack*) that is not in the cache. This will make references to the large aggregate slower, but will make references to all other automatics faster. With large automatics in a side stack, function calls are also less likely to cause the cache to be flushed.

The use of a side stack is not implemented in the current prototype.

3.2.6 Text Alignment: Instructions on CRISP are 2, 6 or 10 bytes in length, and they are aligned on 2 byte boundaries. The prefetch instruction buffer on CRISP is a cache with 16-byte line size. The processor also supports quad-word accesses which are expected to be available in systems using nibble mode dynamic RAMs. When the processor has an instruction discontinuity, and instructions have to be fetched from a new address, it helps to have the new address aligned on a 16 byte boundary. This helps to get one or more instructions from the new address in the smallest possible amount of time.

We currently align functions on 16 byte boundaries.

4. Implementation Overview

The following overview is very brief and is intended to give a feeling for the overall approach of our implementation. This implementation is based on the RCC C compiler[11]. RCC generates code a statement at a time, and depends upon a peephole optimizer to clean up the generated code.

However, most optimizations require code to be generated a function at a time. We inserted a module into the compiler that accepts trees (a statement at a time) from the RCC front end, and holds them until the entire function has been parsed. Next, we perform most of the traditional optimizations on this forest of trees. We then select the templates that will eventually be used to output the assembly language instructions.

At this point, we perform stack compression, instruction scheduling, catch compression, and branch prediction. These optimizations do not affect the selection of templates, although they do affect some of the details of the actual assembly language instruction (e.g. whether a conditional branch predicts ‘‘yes’’ or ‘‘no’’.)

Finally, we emit the machine instructions for each template. The order of instructions is specified by the instruction scheduler.

Most compilers have relied on a postpass peephole optimizer to clean up the generated code. Our compiler performs all the optimizations during compilation and does not depend on any postpass phase.

5. Current Status

The prototype compiler has been running for the last several months. We have compiled a number of benchmarks and several UNIX system programs. These programs were run using the CRISP functional simulator which is cycle accurate. Except for floating point and functions returning structures, it passes our internal C quality test suite.

5.1 Benchmark results

We have measured our performance on the Dhrystone benchmark (version 1.1) and a collection of UNIX system programs. Nine UNIX system programs (cat, comm, diff, echo, mv, nroff, pr, rm, and wc) were chosen for our benchmark collection. Each program in this collection is given a relative weight. We believe that improvements in the UNIX system programs are more representative of the expected improvements in real C programs.

The prototype compiler is not finished yet, so we expect our performance to improve as we implement new optimizations. The following table shows the improvements in the Dhrystone and UNIX system programs when all the optimizations are turned on over our compiler with all optimizations off.

Program Name	Improvement
Dhrystone	52%
UNIX Programs	38%

The following table shows performance gain by individual UNIX system program:

Program Name	Improvement
cat	34%
comm	38%
diff	40%
echo	24%
mv	36%
nroff	41%
pr	26%
rm	36%
wc	17%

The following table shows the contribution made by the most important optimizations for various UNIX system programs:

Program Name	Trad. opt.	Stack comp.	Branch pred.
cat	4%	11%	15%
comm	9%	20%	16%
diff	3%	15%	4%
echo	4%	10%	21%
mv	2%	14%	11%
nroff	5%	11%	28%
pr	3%	3%	26%
rm	4%	13%	15%
wc	1%	0%	11%

The gains for the traditional optimizations are small because they do not include register allocation, which is normally the most valuable traditional optimization. The gains from stack compression come, not only from the higher frequency of accesses in the cache, but also from the use of preferred sharing which allows us to generate fewer moves.

The compile time performance of the optimizing compiler is acceptable. Our optimizing compiler with all the optimizations turned on takes about 50% longer to compile than our nonoptimizing compiler for CRISP.

6. Other implications of the architecture

In addition to the new optimizations made possible by CRISP architecture, there are some other issues of concern to the users (and developers) of the compilation system.

6.1 Branch Prediction

While our current branch prediction scheme is predicting 90% of the branches correctly, it is possible that a branch in a heavily exercised portion of a program may be predicted incorrectly. It would be useful if programmers who are especially worried about performance could explicitly specify the branch prediction bit for important conditional branches. We can imagine several ways to do this:

- Specify the branch prediction in the program. The most likely way to do this is by using the ANSI C `#pragma` directive[12]. While putting directives into the source code may seem inelegant, portability to other ANSI C compilers is not impaired, because any ANSI C compiler will silently ignore a `#pragma` it doesn't understand.
- Allow the programmer to specify the branch prediction scheme by supplying flags to the compiler at run-time.
- Add a feature to the compiler that instruments the generated code to actually count the number of times each conditional branch was or was not taken. This instrumented executable could be used to run a set of benchmarks. The data generated by these runs would then be used to actually set the branch prediction bit for each branch.
- Programmers who understand the branch prediction scheme can write their code in a manner that gives the desired prediction. This is the worst solution. It will lead to contortions in the way the code is written. Also, the implementors of the compiler are not free to change the branch prediction scheme in the future, and are restricted to simple branch prediction schemes that can be understood by a programmer. However, if one of the other solutions is not available, there will be programmers who will use this approach.

These alternatives are only possible because the branch prediction is done by the compiler.

6.2 Use of incoming arguments as temporaries

With the CRISP architecture, there can be problems if a function is passed fewer arguments than it expects, and that function assigns to one of its arguments:

```
main(){
        f(34);
}

int f(a,b,c)
int a,b,c;
{
        c = 17;
}
```

This is invalid C, but will frequently run ''okay'' on many architectures. However, with CRISP architecture, it is possible for such a function to change the automatics of the calling function. In the example above, *main* will only allocate 4 bytes for the outgoing arguments. When *f* assigns to *c*, it will be assigning to bytes 9-12 of the area in *main*'s stack frame reserved for outgoing arguments. Since this area is only four bytes long, some other value (such as an automatic variable) in *main*'s stack frame will be changed. (Many other architectures, such as the VAX,* also evidence this behavior.)

We currently use incoming arguments that are dead as space for temporary variables. While this allows us to do a better job of stack compression, this makes the error described above much more likely to occur. In particular, a function that never assigns to its arguments may get into trouble, because the compiler will generate such an assignment if it uses the space occupied by the argument as a temporary.

The decision to use incoming arguments will depend on the tradeoffs between getting the best performance out of correct code, and breaking invalid, but working, programs.

6.3 Debugging Optimized Code

As we get more aggressive in our optimizations, the difficulty of debugging the optimized code also grows. Debugging optimized code has always been problematic;[13] however, with our compiler, these problems appear much more often. In particular, the stack compression routine causes us to routinely clobber an automatic variable once it is dead; if the debugger is stopped after the last use of the automatic, and asked to print its value, it must be able to answer ''I forgot.''

* VAX is a trademark of Digital Equipment Corporation.

7. Summary and Conclusions

We have identified several non-traditional optimizations that are especially valuable on the AT&T CRISP microprocessor. We have built a prototype compiler that implements most of these optimizations. We compiled a set of actual UNIX system programs with this compiler, and they run an average of 38% faster than when compiled without optimizations.

We have also demonstrated that it is possible for the compiler to do an effective job of branch prediction. The ability to know exactly what language constructs were used to generate a conditional branch allows the compiler to correctly predict the outcome of the branch 90% of the time, which compares favorably with branch prediction strategies that are implemented in hardware.

REFERENCES

1. Berenbaum, A. D., Colbry, B. W., Ditzel, D. R., Freeman, R. D., McLellan, H. R., O'Connor, K. J., and Shoji, M., ''A Pipelined 32-bit Microprocessor with 13Kb of Cache Memory,'' *Proceedings of the International Solid State Circuits Conference*, February, 1987.

2. Ditzel, D. R. and McLellan, H. R. ''Register Allocation for Free: The C Machine Stack Cache,'' *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 48-56 (March 1982).

3. Berenbaum, A. D., Ditzel, D. R., McLellan, H. R., ''Introduction to the CRISP Instruction Set Architecture,'' *Proceedings of the IEEE Spring COMPCON* 1987.

4. Berenbaum, A. D., Ditzel, D. R. and McLellan, H. R. ''Architectural Innovations in CRISP Microprocessor,'' *Proceedings of the IEEE Spring COMPCON* 1987.

5. Aho, A., Sethi, R., Ullman, J. ''*Compilers: Principles, Techniques, and Tools*,'' pp. 585-722, Addison-Wesley 1986.

6. Clark, D. and Levy, H. ''Measurement and Analysis of Instruction Use in the VAX-11/780,'' *The 9th Annual Symposium on Computer Architecture* 10(3), pp 9-17. (April, 1982.)

7. Wiecek, C. ‘‘A Case Study of VAX-11 Instruction Set Usage for Compiler Execution,’’ *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 177-184 (March 1982).

8. Smith, J. E. ‘‘A Study of Branch Prediction Strategies,’’ *Proceedings of the 8th International Symposium on Computer Architecture"*, pp. 135-148 (June, 1981).

9. Lee, J. K. F. and Smith, A. J. ‘‘Branch Prediction Strategies and Branch Target Buffer Design,’’ *IEEE Computer* Volume 17 Number 1 (January 1984), pp 6-22

10. Gibbons, P. and Muchnick, S. ‘‘Efficient Instruction Scheduling for a Pipelined Architecture,’’ *Proceedings of the SIGPLAN ’86 Symposium on Compiler Construction, SIGPLAN Notices* Vol. 21 No. 7 (July 1986), pp 11-16.

11. Kristol, D. ‘‘Four Generations of the Portable C Compiler,’’ *USENIX Summer Conference Proceedings*, June 1986, pp 335-343.

12. *Draft Proposed American National Standard for Information Systems - Programming Language C*, Doc. #X3J11/86-151, October 1, 1986

13. Hennessy, J., ‘‘Symbolic Debugging of Optimized Code,’’ *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 3, July 1982, pp. 323-344.