# Using GCC as an Efficient, Portable Back-End*

Derek Chiou


## Abstract

In order to generate efficient object code for modern microprocessors, aggressive compilers customize their code generators (the back-end of the compiler) for each target processor. Customized back-ends, however, are major undertakings that most experimental compilers writers cannot afford. We describe a method to easily generate efficient object code for virtually any UNIX platform. The method relies on Gnu's GCC compiler and its extensions to the C language. These techniques have been applied to pHluid[1], a pH/Id compiler. The techniques, their application to pHluid and some performance results are reported in this paper.


## 1 Introduction

Making the best use of modern microprocessors requires very sophisticated instruction scheduling due to their multiple instruction issuing capabilities and longer pipelines. To generate the best possible code, aggressive compilers customize back-ends for each supported architecture. Such back-end customization requires substantial time as well as significant expertise in the target architecture, assets that small compiler efforts generally do not have.

An attractive solution to an experimental compiler writer is to compile to a standard language such as C and let the C compiler do the final code generation. Such a technique is desirable because generating the code becomes trivial, the generated code is extremely portable and it can use a large body of existing support libraries and tools. In fact, many language implementations, such as Scheme2C, F2C (Fortran), P2C (Pascal), and Kyoto Common Lisp are all compiled to C rather than directly to object code. Though there is some loss of efficiency, the loss is relatively small since the *execution model* of these languages is essentially the same as C's. Procedures in the source languages correspond to procedures in C, and the programmers using those languages are used to making the same decisions as programmers writing native C.

For programming languages that naturally generate very short threads of control which we will call multithreaded languages, however, the overhead of the general C procedure call mechanism is too high. Short threads do not need to build the C procedure linkage nor allocate a C stack frame, and generally pass very few, if any, variables. Thus, a simple jump is a sufficient mechanism to transfer control for multithreaded code and register passing conventions are sufficient for passing values. Support for such operations is found on virtually all modern microprocessors in the form of fast branches which look-ahead in the instruction stream and large register sets. A back-end writer wants full access to abstract hardware mechanisms while having the conveniences and code generation of a high-level language. Conventional languages such as C, however, do not expose such hardware mechanisms to the user for good reasons.

The solution explored in this paper provides the experimental compiler writer with what needs: the ability to alter control flow and use registers *and* the ability to use a high-level language and its conveniences in a seamless fashion. The scheme requires GCC's extensions to the C language. The scheme produces extremely efficient code, taking advantage of GCC's optimizations, register allocation and code scheduling while avoiding C overheads. Since GCC has been ported to most machines, the code is extremely portable and is generally efficient. The code obeys standard C conventions, allowing it to call C functions and library routines. Using GCC in this fashion allows the compiler writer to use GCC as a flexible, portable back-end.

## 2 GCC: A Portable Back-end

In addition to implementing ANSI C, gcc also defines two basic extensions which are extremely useful to code generation. The first extension is the ability to perform an indirect jump. That extension consists of two features: the ability to store label values into a variable and the ability to jump indirectly to a label through a variable. The second extension allows GCC code to allocate registers explicitly.

An interesting question is how can one jump from a thread in one procedure to a thread in another procedure? Jumps within a single procedure can (and should) be done with the ANSI C jump, but those cannot jump to labels outside of the jumping procedure.

The indirect jump can be used to jump from one cord (a collection of related threads compiled into single C procedure) to another. Two problems exist when doing so: getting the instruction pointer to jump to and avoiding the standard C procedure calling code. The second problem is easy to deal with: you never execute the head of the function or the tail of the function and thus never execute the C procedure calling code. Getting the instruction pointer can only be done from within the procedure, so each procedure initializes a set of global variables with all of the possible labels that threads from other procedures should be able to access. The called procedures are traversed during the start of the program.

To pass arguments between two cords, one can either use registers or specified memory locations. To pass arguments between threads in the same cord, C variables local to the cord's procedure can be used. The GCC will insure that procedure variables will be live between threads. If a thread needs temporary registers, a scope can be used to provide those temporary registers. Registers can be allocated globally to keep frequently accessed data quick to access. If all registers used are in callee saved regions of the register set, native C functions can be called indiscriminately, without having to save registers. Of course, if more registers are needed, C function calls can be surrounded by save/restore register code.

Such code can be ported with absolutely no effort other than selecting the correct registers to use on a particular

*This work is supervised by Prof. Arvind and is supported in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

architecture.

## 3 Applying the Techniques to pHluid

pHluid[1] is an Id[2]/pH[3] compiler written in the Scheme2C dialect of Scheme and C. Id and pH are parallel multi-threaded functional languages. The compiler is meant to be extremely portable and a complete implementation of both languages and has not been optimized or even well partitioned (the process of creating threads). The code it originally generated was standard C, where each thread was a single procedure. It originally ran between 20 and 40 times slower on a Sparc 10 than native C code running the same algorithm.

We first describe the execution model of pHluid, then describe how we used the techniques described in the previous section to speed up the executable code. We then discuss performance.

### 3.1 Execution model of pHluid

Since Id and pH are parallel languages, they are compiled to a potentially parallel execution model. A tree of frames, rather than a stack of frames, is used for procedure frames. A global, synchronizing heap is assumed.

In addition to the tree of frames and heap, pHluid uses a "message stack" which is essentially a priority queue containing work to do in the form of continuations. The stack is not necessarily a simple stack. When a thread wishes to create more work which it will not exploit immediately, it puts it into the message stack. Incoming messages from the network are also placed into the message stack. When a thread completes and it does know what work to schedule next, it takes the highest priority continuation from the message stack and executes it.

### 3.2 Implementation

We take the abstract machine language output of the pHluid compiler, called P-RISC, and translate it into stylized GCC code. Each cord contains a single codeblock, which is roughly a procedure in the source language. When direct jumps to the same cord can be used to alter the control flow of the program, they are used. If a thread terminates and knows it should schedule a thread from another cord, it will jump to that thread directly. Otherwise, the terminating thread will jump to the next thread on the priority queue. The priority queue is implemented as a hierarchy of stacks, currently a single stack. The terminating thread simply jumps to the instruction pointer on the top of the stack to schedule the next thread. If more complex scheduling is desired, more code can be added.

To pass information between threads in a single cord, procedure variables are used. The instruction pointers on the message stack are accompanied by their data. Currently, no registers are reserved to pass information between threads; thus, when going from one cord to another, all values are pushed onto the message stack. We plan to pass values through reserved registers in the future.

Threads which can be entered from other cords have preliminary code to read values from the message stack into its local variables. An additional entry point is provided after the local variables are loaded to allow a thread from the same cord to assign the local variables directly.

Frame allocation is done on a modified stack which can tolerate holes. Stack allocation of frames works most of the time, and is very quick. When more heap-like frame allocation needs to be done, a penalty is paid, but this happens very rarely.

| Program | C time | pHluid time | Ratio | Ratio to original |
|---|---|---|---|---|
| nfib 30 | 1.1 | 1.6 | 1.5 | 5.9 |
| matmult 200 | 5.4 | 31.1 | 5.8 | 2.7 |
| paraffins 20 | 1.4 | 6.4 | 4.6 | 2.3 |
| nqueens 11 | 8 | 19.3 | 2.4 | 3.3 |

Table 1: Performance Results for Sparc 10

### 3.3 Results

The back-end runs on Sparc, Alpha and RS6000 architecture families. The results are summarized in Table 1 for a Sparc 10. The "Ratio" column is the ratio of pHluid to C and the "Ratio to Original" column is the ratio of the original pHluid implementation to the new one. The ports to the various architectures was trivial. Overall the run-times of code generated with the new back-end are between 2 and 6 times faster than those generated by the original back-end and is between a factor of 1.5 and 6 times slower than native C code. Since the back-end is not yet complete, explicit tagging for garbage collection is being performed, and there are very few middle-end optimizations yet implemented, we believe that significant performance gains over the current ones are achievable.

## 4 Suggested C Compiler Modification

Though GCC extensions enable us to generate efficient code, it would be desirable if the C compiler will allow functions to be compiled without the standard procedure call code which is found at the head and tail of the executable code. Then, if jumps can be made directly to procedures, each procedure can now be a single thread. Procedures that encompass the threads of a cord can be used for cord-level variables.

## 5 Conclusions

This paper describes a novel way to use GCC to act as a portable back-end which will generate efficient code. Using such techniques, a compiler writer can generate extremely efficient code with little effort by leveraging the back-ends in the GCC compiler. We produce promising results and suggest trivial modifications to C compilers which will make this technique even more useful.

## References


[1] R. S. Nikhil. A Multithreaded Implementation of Id using P-RISC Graphs. July 1991.

[2] R. S. Nikhil and Arvind. *Programming in Id: A Parallel Programming Language.* Book in preparation, 1990.

[3] R. S. Nikhil, Arvind, J. E. Hicks, S. Aditya, L. Augustsson, J.-W. Maessen, and Y. Zhou. pH Language Reference Manual, Version 1.0. Computation Structures Group Memo 369, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, January 1995.