Prototyping Compiler and Simulation Tools with PCCTS

Peter J. Dahl Peter E. Bergner

John C. Mejia Matthew T. O'Keefe

Department of Electrical Engineering
University of Minnesota
200 Union Street S.E.
Minneapolis, MN 55455

`dahl@ee.umn.edu`
`(612) 626-8083`

July 1, 1994

Abstract

This paper describes our experiences using PCCTS in our optimizing compiler and simulator development effort. The tools are used in our C front end, a code scheduler, a linker, an instruction level simulator, and a detailed cycle-level simulator. One of the Antlr grammars is used twice in the compiler and in two different simulators. This type of reuse and flexibility is a strength of PCCTS. A working knowledge of PCCTS made the rapid implementation of a variety of compilers and simulators possible with very little manpower.

Keywords: PCCTS, parsing, grammar, front-end, instruction level simulation.

[1]This work was supported by contract No. DAAL02-89-C-0038 between the Army Research Office and the University of Minnesota for the Army High Performance Computing Research Center, Office of Naval Research Grant No. N00014-93-1-0426, and National Science Foundation Grant NSF CCR-9110261.

1 Introduction

The advanced architectures research group at the University of Minnesota is interested in researching new compiler algorithms and machine architectures. We have developed an optimizing compiler capable of producing single processor or SPMD (single program, multiple data) code with backends for DLX [HeP90] and the DEC Alpha [DEC92, DEC93]. We have implemented instruction level simulators for both DLX and the Alpha, and a detailed cycle level simulator for DLX.

PCCTS[1] is used extensively in all of these efforts. It's ease of use, flexibility, ease of debugging, user actions, linkage with C, speed, and syntax all contribute to the success of our projects. Our research began as PCCTS was first released; the initial versions worked so well and intuitively that we soon took for granted the speed that prototypes could be constructed and put into use. Several problems with PCCTS were discovered and fixed as a direct result of our heavy usage of the tools in a variety of contexts. Extensions of the tools were added to make certain tasks easier. Sorcerer was in its very initial stages of development when we wrote our AST traversals; had it been farther along it would surely be used in our compiler now.

This paper discusses three grammars: our C front end, a DLX assembly grammar, and an Alpha assembly grammar. Our grammar that recognizes DLX assembly language is perhaps the most interesting because it is reused four distinct times in our compiler and simulator toolset.

2 C Front End

We started learning Antlr by implementing a C front end for the compiler using the newly released beta version of Antlr and DLG in the spring of 1991. Fortunately, Terence Parr had written a C grammar and packaged it with PCCTS. We modified the C grammar to generate an AST the way we wanted and added our own symbol table routines. The grammar uses automatic AST generation with a few exceptions. This section describes some of the nuances of our C front end.

2.1 Lexical Classes

We chose to use the existing C preprocessor (`cpp`) to give our compiler the macro and definition behavior of a production C compiler. The preprocessor emits lines starting with "`#`" telling what parts of headers were used. These lines are easily parsed with a separate lexical class for the line number and file name so that error messages have accurate information.

Comments, strings, and characters are recognized with a separate lexical class for each as shown in the PCCTS C grammar. We found this a very useful property of PCCTS, it lends itself very nicely to structuring modular grammars.

2.2 Token Ranges

We were intrigued by the token header files produced by Antlr. We wanted to use these same token names in all stages of the compiler from traversing the AST to code generation. Furthermore, we wanted to be able to look for ranges of tokens, *i.e.*, all the binary operators. We could have used the token class mechanism in Antlr except that at the time we wrote our front end it did not exist and we wanted token ranges that could be used outside the front end of the compiler.

[1]PCCTS (Purdue Compiler Construction Tool Set) includes an LL(k) parser generator (Antlr), a DFA-based lexical analyzer generator (DLG), and a tree walker generator (Sorcerer). [PDC91]

We took advantage of the fact that DLG assigns token numbers in consecutively increasing order. Figure 1 shows how the binary operator tokens and the associated token ranges are organized. In later stages of the compiler, the binary operators can be accessed by the expression: `BIN_END <=`

```
#token BIN_BEGIN          /* set of all binary operators          */
#token ADD      "\+"
#token SUB      "\-"
#token MULT     "\*"
#token DIV      "/"
#token IBIN_BEGIN         /* bin_ops that have int operands only  */
#token MOD      "\%"
#token OR       "\|\|"
#token AND      "&&"
#token B_OR     "\|"
#token B_AND    "&"
#token XOR      "^"
#token LL       "\<\<"
#token GG       "\>\>"
#token IBIN_END
#token COMPARE_BEGIN      /* produce ints or status bits          */
#token LT       "\<"
#token GT       "\>"
#token LTE      "\<="
#token GTE      "\>="
#token EE       "=="
#token NE       "!="
#token COMPARE_END
#token BIN_END
```

Figure 1: Binary Operator Tokens and Sets

`token && token <= BIN_BEGIN`. Two sub-ranges further delineate the binary operators that are integer only (`IBIN_BEGIN`–`IBIN_END`) and the compare operations (`COMPARE_BEGIN`–`COMPARE_END`). Note that the end points of each range are tokens themselves, but they never match any input in the front end. This increases the total number of tokens DLG has to process but does not make the DFA larger because they do not match input.

Often it is desirable to have the token names for use elsewhere in the compiler. The token header (`tokens.h`) produced by Antlr can be included for the token definitions, but the strings for each token are in a static array in the file `err.c`. The `csh` script shown in Figure 2 can be used to convert the `zztokens` array in `err.c` into a general header that can be included elsewhere in the compiler.

In C it is possible to pre-increment or post-increment a variable. The "`++`" token is the same in each case and so the context must be used to recognize the difference. We wanted two different tokens, one for each operation. In our grammar we define all four tokens (`PreInc`, `PreDec`, `PostInc`, `PostDec`), but only `PreInc` and `PreDec` recognize input. The grammar then uses the AST con-

```
#!/bin/csh
# script to strip out the zztokens array from err.c, put in zztokens.h
# 8-6-91 peter dahl
if ( $#argv != 2 ) then
        echo "usage: $0 infile outfile"
        exit
endif
rm -f $2
set s1='{ if(substr($2,0,9) == "*zztokens") {f = 1; printf("static ");}'
set s2='if (f) print $0; if ($0 == "};") f = 0;}'
awk "$s1$s2" $1 > $2
```

Figure 2: `csh` Script for Creating a Token Header

structor function (`zzcr_ast()`) to create a node with the correct token number. This node is linked it into the AST once the context is known. Unary operators are usually the textually the same as the binary operators (`&`, `*`, `+`, `-`) and so do not match input. The same procedure is followed for these once the context is known.

2.3 AST Construction

Our AST construction is very much like that distributed with PCCTS. The differences mainly stem from array references and the symbol table. Our array reference AST is much more detailed than the original, this solves problems later in the compiler when every operation should be explicit. Figure 3 shows a section of the automatic AST for the array reference `a[i][j]` and Figure 4 shows what we generate. Note that the AST in Figure 4 contains the left shift operation of the index calculation, making this explicit is key for ease of writing the live range analysis and code generator portions of the compiler. A symbol table operation is required to find the first dimension of the array and the element size of the array. The AST in Figure 3 is too compact; too much is assumed in each node and has to be looked up. This makes code motion and common subexpression elimination difficult.

The ASTs for subroutines are represented as the children of the root node for each file that the front end sees. A better alternative would be to make the AST part of a global data structure visible to the linker so that each subroutine can be found quickly along with call site information, aliasing information, and registers used. This would speed up interprocedural analysis and the linker.

There are many optimizations in our compiler that change the structure of the AST including verticalization, constant folding, and expansion of compound statements (*i.e.*, `a = b = 0;` $\rightarrow$ `a = 0; b = 0;`). Verticalization minimizes the interference of live ranges in the compiler by reorganizing the tree to minimize conflict between compiler temporaries, this results in an AST where the expressions become more vertical. Changing the AST once the compiler analysis has begun can cause problems in terms of the control flow graph and live range information if pointers point into the AST. The beginning of each basic block should be the only handle to a particular portion of the AST.

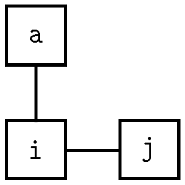

Figure 3: Automatic AST

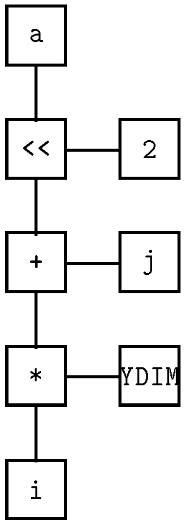

Figure 4: Better AST

2.4 Symbol Table

A coherent scheme for designating the scope of symbol table entries is probably the most important part of the symbol table. Since our AST changes due to reorganizing optimizations, we had to invent a scheme that did not involve pointers to the AST. We decided on a coordinate triplet consisting of a subroutine name, scope, and subscope number. Pointers are maintained to the beginning of each scope in a subroutine. The scope and subscope numbers can be generated in the order the code is parsed and are reset at the beginning of each subroutine. The scope describes the nesting level starting with zero for globals. The subscope then numbers scopes on the same level in the order that they are seen by the parser. The example in Figure 5 shows the various scoping levels.

A preprocessing stage forces each `IF`, `WHILE`, `FOR`, and `DO` statement to start a new scope[2]. This allows basic blocks in the subroutine to coincide with new scopes. Eventually we would like to have a coordinate system that encompasses both scoping and basic blocks into one system. This would help amortize the cost of building the symbol table because the control flow graph could use it for referring to basic blocks.

Technically the scope of formal parameters is different than the scope of local variables in that subroutine. We make them the same scope because we have not found a situation where they need to be different.

2.5 Traversing an AST Fast

Once the AST is stable, the compiler then traverses it often during the execution of the various algorithms to build the interference graph and generate code. These algorithms are often oriented around three-address code [ASU86]. We chose to perform the algorithms using the original AST. With the fast traversal data structure described in this section, the differences between these two approaches is small.

[2]We insert curly braces where they are optional.

```
int     x;               /* name: null, scope 0, subscope 0      */
sub1(int y)              /* name: sub1, scope 1, subscope 0      */
{
        int     i;       /* name: sub1, scope 1, subscope 0      */
        for (i=0; i<10; i++)
        {                /* name: sub1, scope 2, subscope 0      */
            if (i == 5)
            {            /* name: sub1: scope 3, subscope 0      */
            } else
            {            /* name: sub1: scope 3, subscope 1      */
            }
        }
        while (i > 0)
        {                /* name: sub1, scope 2, subscope 1      */
            if (i == 5)
            {            /* name: sub1: scope 3, subscope 2      */
            } else
            {            /* name: sub1: scope 3, subscope 3      */
            }
            i--;
        }
}
```

Figure 5: Scoping Coordinates

Profiling the compiler showed that a disproportionate amount of time was being spent traversing the AST in a recursive manner. To alleviate this problem, we build a simple data structure that makes any traversal very fast. There are three main AST traversals that we use in the compiler: global, local top-down, and local bottom-up. The global traversal visits every AST node ignoring basic block boundaries; a local traversal traverses only a single basic block. The traversals are done once to initialize an array of pointers to AST nodes. The traversal arrays are dynamically allocated using the number of nodes in the AST as counted by the Antlr grammar. For the local traversals, there is a separate array of pointers for each basic block and another array of pointers indexed by the basic block coordinates to the arrays of AST pointers.

An index into the array of AST pointers defines the position in the traversal. To go to the next node, simply increment the index. No recursion is required once the AST pointers arrays are built. Arbitrary traversals can be built with only the initialization of the pointer arrays and their memory space as a cost.

A stack of traversal positions (indices into the pointer arrays) is maintained. This allows a traversal to stop at some point, start another traversal (presumably looking for something), and then continue with the original traversal. We originally used this feature in our algorithm that discovers live ranges, it is not used now.

Profiling the compiler after implementing this traversal method showed that traversing the AST is now one of the cheaper routines.

3 Parsing Assembly Language

This section discusses our grammars that recognize DLX and Alpha assembly language. Examples from the DLX grammar will be used. Both grammars follow the same form but differ in their token definitions. Initially we considered extending the DLX grammar to parse both DLX and Alpha assembly, but there were enough instruction format changes to make this cumbersome.

3.1 Antlr Front End

The front end for the simulators was created with PCCTS. The grammar recognizes the appropriate assembly language instructions and assembler directives. Separate lexical classes are created for comments, assembler directives, linker directives, and general expressions. The general expressions involve address calculations for the relocating linker to solve during linking. The grammar has a main rule to recognize labels, instructions, assembler directives, linker directives, comments, and blank lines. An optional comment can follow labels and instructions on the same line.

The types of numbers represented in the assembly produced by our compiler are shown in Figure 6. Note that floating point numbers are not recognized. This was done after we noticed precision and rounding differences on different machines and between compilers. The compiler now writes out the hex bit pattern (32 or 64 bit) for each floating point number; this way the simulator can read in the exact number that was written out with no rounding or truncation errors (assuming floating point formats that conform).

Our DLX instruction level simulator has up to four register files: integer (`r`), floating point (`f`), vector (`v`), and condition code (`c`). The size of these register files is dynamically allocated up to 64k for each. The token to recognize register numbers is shown in the last line of Figure 6.

The instructions are broken down into categories: no operands, one operand, two operand, three operands, operand and label, subroutine call, and load/store. These are broken down this

```
#token DecInt   "[1-9][0-9]*|[0]"
#token HexInt   "[0][xX][0-9a-fA-F]+"
#token Immed    "#{[\-]}[0-9]+"
#token ImmedH   "#0x[0-9a-fA-F]+"
#token Reg      "[rfvc][0-9]+"
```

Figure 6: Tokens For Number Recognition

way partially because of syntax but also due to the kinds of user actions to be performed for each. There is a rule (Figure 7 shows the three operand rule) for each category describing the syntax of the instruction and a second rule (Figure 8) describing which tokens are in the category. Note

```
instr3:                 << int imm, regi; char off[50];     >>
        in3 reg ","
        reg ","         << imm=NREG; regi=NREG; off[0]='\0';>>
        ( reg           << regi = $1.val;                   >>
        | immed         << imm = $1.val;                    >>
        | "\@" Ident    << strcpy(off,$2.text);             >>
        | EXPR          << strcpy(off,$1.text);             >>
        )
        << instr(PCFG,$1.val,$2.val,$4.val,regi,imm,off);   >>
        ;
```

Figure 7: Instruction Rule

that the instruction classes defined by the rules in the grammar are not necessarily the same as the token classes. The instruction classes are based on syntax and user actions whereas the token classes are based on instruction functionality.

3.2 Intermediate Representation

The front end places the instructions it parses into the intermediate representation shown in Figure 9. There are quite a number of pointers for the code scheduler [GiM86, War90, PDE93]. Each instruction is linked with the instructions before and after it so that the code optimizer can move instructions around easily. Control flow graph structures for global analysis point to basic blocks of code. The optional portions of the intermediate representation are dynamically allocated; this saves quite a bit of space but the calls to `malloc` are a significant portion of the run time.

3.3 Back End Considerations

Since the grammar that parses DLX assembly is used in four distinct places by our research group, the calls to user actions have to be general enough to handle a variety of situations. There are four sets of user action subroutines with the same names but different functionality. The instructions

```
in3:    ( OP_ADD                << $$ = $1; >>
        | OP_ADDI               << $$ = $1; >>
        | OP_ADDU               << $$ = $1; >>
        | OP_ADDUI              << $$ = $1; >>
        | OP_AND                << $$ = $1; >>
        | OP_ANDI               << $$ = $1; >>
        | OP_DIVI               << $$ = $1; >>
        | OP_MUL                << $$ = $1; >>
        | OP_MULI               << $$ = $1; >>
        | OP_OR                 << $$ = $1; >>
        | OP_ORI                << $$ = $1; >>
        | OP_SEQ                << $$ = $1; >>
        | OP_SEQI               << $$ = $1; >>
        ...
        );
```

Figure 8: Tokens in Each Instruction Class

```
typedef struct instr_type
{       int     code;           /* token number                  */
        int     line;           /* line number                   */
        int     addr;           /* address, used by simulator    */
        char    *label1         /* label                         */
        int     dest;           /* destination register          */
        int     src1;           /* first source register         */
        union src2_type
        {       int     src2;   /* second source register        */
                int     immed;  /* immediate or constant offset */
                char    *offset;/* offset for loads/stores       */
        } s2;
        char    *label2;        /* target label in a branch      */
        char    *comment;
        struct instr_type *next, *prev; /* doubly linked list    */

        struct arc_type *d_arc, *u_arc; /* various pointers for  */
        struct instr_type *root, *leaf; /* our code scheduler    */
        int   parents, children;
        int   wgt, opt_x_time, x_time;
} INSTR;
```

Figure 9: Instruction Representation

are divided into categories by syntax and whether a different user action needs to be called. We have one user function for each of the instruction categories listed.

We have four different back ends for our DLX grammar: a code scheduler, a linker, an instruction level simulator, and a detailed timing simulator. All four define the user functions differently. The user functions cover each of the categories of instructions and are shown in Figure 10. The

```
nop(CFG *cfg, int code);                            /* no operand          */
instr(CFG *cfg, int code, int dest, int src1,
        int src2, int immed, char *offset);         /* 1, 2, or 3 operands  */
label(CFG *cfg, int code, char *label1);
branch(CFG *cfg, int code, int src1, int src2, char *label2);
load(CFG *cfg, int code, int dest, char *offset, int immed, int src1);
store(CFG *cfg, int code, char *offset, int immed, int src1, int src2);
comment(CFG *cfg, int code, char *comment);
trap(CFG *cfg, int code, int immed);
```

Figure 10: User Functions

code scheduler builds a control flow graph with doubly linked lists of instructions for each basic block. It then schedules the code and writes it out to a file in the same form. The linker builds a symbol table of labels and subroutine calls, performs data layout, and resolves addresses.

The instruction level simulator parses DLX assembly and the user actions output C function calls for each instruction. The back end of this simulator then links this file with subroutines that perform the operation for each instruction. Note that there is no instruction memory for this type of simulator since the instructions are compiled into the simulator. Furthermore, there is no dynamic instruction decoding because the link with the instruction subroutines effectively does this. This simulator is very fast; on a SPARC-10 it runs at about 700,000 instructions/second (real time to simulated time ratio is 17:1). An even better ratio may be achieved by inlining the instruction subroutines.

The pipelined simulator employs user actions to format instructions into a data structure that holds each instruction in decoded form. Both instruction memory and data memory are much larger than simulated memory (4x) due to the way instructions are maintained in decoded form. This simulator emulates the complete instruction execution pipeline with multiple functional units, instruction cache, data cache, and memory system. Operating system overhead is also modeled. It is capable of giving exact cycle counts for program timing and runs much slower, averaging 8,000 instructions/second on a SPARC-10 (real time to simulated time ratio is 1400:1).

4 Example

Our DLX instruction level simulator operates by parsing the user's DLX assembly language program and translating it into a C language program [MAF91]. The majority of DLX assembly instructions (*e.g.*, arithmetic instructions) are converted into a single C function call. These function calls emulate the operation of the assembly instructions on the simulated register file and memory. In addition, these function calls update dynamic instruction counters as well as performing some optional tasks such as address trace calculations, memory access checks (to detect accesses outside the memory space), and floating point exception checks and others.

In DLX, the branch instructions contain delay slots where the instruction following the branch is executed before the branch is completed. To simulate this behavior, we translate each branch into two C statements. The first statement tests for a condition and puts the result of the test in a local variable. It also updates the instruction and trace counters. This is immediately followed by the instruction in the delay slot and then an `IF` statement based on the condition result in the local variable and a goto statement that performs the actual jump. Subroutine calls are treated in an analogous manner.

In addition to the function calls used to simulate the instructions, we also include additional function calls to initialize our simulated machine, implement break points and to print various statistics such as instruction counts, subroutine call frequencies and basic block frequencies. Our C language simulator is then compiled and the resulting executable file is run to simulate the program. Figures 11–13 show an example C program, its associated DLX assembly program, as well as our corresponding C language simulator. Figure 14 shows a few of our simulator subroutines implementing various DLX instructions.

```
void main()
{
        for (i=0 ; i < 10 ; i++)
        {
                a = 10;
                b = 3;
                c = a % b;
                d = a % 3;
        }
        subroutine();
}

void subroutine()
{
        a = 10;
        b = 3;
        c = a / b;
}
```

Figure 11: Sample C Code

```
main:
        addui   r29,r0,10000
        addi    r5,r0,0
L001:   slti    r4,r5,10
        beqz    r4,L002
        nop
        addi    r4,r0,10
        addi    r6,r0,3
        mod     r6,r4,r6
        modi    r4,r4,3
        addi    r5,r5,1
        j       L001
        nop
L002:   jal     subroutine
        nop
        trap    0
subroutine:
        addi    r29,r29,-268
        addi    r7,r0,10
        addi    r8,r0,3
        div     r9,r7,r8
        addi    r29,r29,268
        jr      r31
        nop
```

Figure 12: Assembly Code

```
void main(void)
{       int BRANCH;
        machine_init();
        S = new_stats("main");
        addi(S,R(29),R(0),10000);
        addi(S,R(5),R(0),0);
L001:   slti(S,R(4),R(5),10);
        BRANCH = beqz(S,R(4));
        nop(S);
        if (BRANCH)  goto L002;
        addi(S,R(4),R(0),10);
        addi(S,R(6),R(0),3);
        mod(S,R(6),R(4),R(6));
        modi(S,R(4),R(4),3);
        addi(S,R(5),R(5),1);
        j(S);
        nop(S);
        goto L001;
L002:   jal(S);
        nop(S);
        subroutine();
        trap(S,0);
        print_stats(S);
}

void subroutine(void)
{       S = new_stats("subroutine");
        addi(S,R(29),R(29),-268);
        addi(S,R(7),R(0),10);
        addi(S,R(8),R(0),3);
        div(S,R(9),R(7),R(8));
        addi(S,R(29),R(29),268);
        jr(S,R(31));
        nop(S);
        print_stats(S);
}
```

Figure 13: Translated Assembly Code

```
void addi(STATS *counter, int dest,
    int src, int immed)
{
    counter->addi++;
    counter->int_op++;

    r[dest] = r[src] + immed;
}

int beqz(STATS *counter, int src)
{
    counter->beqz++;
    counter->int_op++;

    if (r[src] == ZERO)
    {   counter->branch_taken++;
        return TRUE;
    } else
    {   counter->branch_untaken++;
        return FALSE;
    }
}

void lf(STATS *counter, int dest,
    int offset, int base)
{
    counter->lf++;
    counter->int_op++;

    if (r[base]+offset < 0 ||
         r[base]+offset > (mem_size-4))
    {   printf("address error in lf\n");
        exit(-1);
    }
    f[dest] = mem[(r[base]+offset)>>2];
}
```

Figure 14: Instruction Simulation

5 Conclusions

PCCTS contains several tools for writing powerful and fast language parsers quickly. The fact that user actions can appear anywhere in the parser or lexer lends an intuitive feel to the tools. We have shown several examples of how we use the tools and how they are easily adaptable to our unique problems. Our experience with these tools has been very favorable, the flexibility and ease of debugging saved us countless hours of time.

> An apprentice carpenter may want only a hammer and saw, but a master craftsman employs many precision tools. Computer programming likewise requires sophisticated tools to cope with the complexity of real applications, and only practice with these tools will build skill in their use. *Robert L. Kruse*, Data Structures and Program Design

Acknowledgments

We would like to thank Dr. Terence Parr for his timely help with PCCTS.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[DEC92] DEC. *Alpha Architecture Handbook.* Digital Equipment Corporation, 1992.

[DEC93] DEC. *DEC OSF/1 Assembly Language Programmer's Guide.* Digital Equipment Corporation, Maynard, MA, March 1993.

[GiM86] P. Gibbons and S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN Conference on Compiler Construction*, pages 11–16. ACM, June 1986. Palo Alto, CA.

[HeP90] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., P.O. Box 50490, Palo Alto, CA 94303, 1990.

[MAF91] C. Mills, S. Ahalt, and J. Fowler. Compiled instruction set simulation. *Software Practice and Experience*, 21(8):877–889, August 1991.

[PDC91] T. Parr, H. Dietz, and W. Cohen. PCCTS reference manual. Technical report, Electrical Engineering, Purdue University, West Layfayette, IN 47909, August 1991. `pccts@ecn.purdue.edu`.

[PDE93] J. Page, P. Dahl, D. Engebretson, P. Woodward, and M. O'Keefe. Code scheduling for high performance computing. Technical report, Army High Performance Computing Research Center, University of Minnesota, 1100 Washington Avenue South, Minneapolis, MN 55415, October 1993. Preprint #93-101.

[War90] H. Warren, Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):85–92, January 1990.