1990

High-Level Microprogramming: An Optimising C Compiler for a Processing Element of a CAD Accelerator

Paul Kenyon
University of Nebraska-Lincoln

Prathima Agrawal
AT&T Bell Labratories

Sharad C. Seth
University of Nebraska-Lincoln, seth@cse.unl.edu

Proceedings of the 23rd Annual Workshop and Symposium., Workshop on Microprogramming and Microarchitecture. Micro 23.
doi: 10.1109/MICRO.1990.151431

High-Level Microprogramming: An Optimizing C Compiler for a Processing Element of a CAD Accelerator

Paul Kenyon
University Nebraska - Lincoln
Lincoln, Nebraska

Prathima Agrawal
AT&T Bell Laboratories
Murray Hill, New Jersey

Sharad Seth
University Nebraska - Lincoln
Lincoln, Nebraska

Abstract: The development of a high-level language compiler for a micro-programmable processing element (PE) in the MARS multicomputer is described. MARS, an MIMD message passing machine, was designed to speed up VLSI CAD and similar other non-numerical applications. The need for support of a high-level language at the PE level of a multicomputer is considered, and the choice of C as an appropriate programming language is justified. Special features found in VLSI processors are examined along with compiler support for them.

Conventional retargetable compiler techniques are shown to be inadequate for the highly concurrent micro-programmable PE. These techniques must be extended for microcode generation. The design of the MARS compiler is outlined. Performance data is provided to evaluate the benefit of various compiler optimizations, and to compare compiler generated microcode to hand generated microcode in terms of space and time performance

Keywords: Microcode compiler, Code Generation, Front-End DAG Compiler, Hand vs. Compiled Microcode, Performance Data, Space/Time Overhead, Hardware Accelerator, Programming Environment for CAD

1 Introduction

The problem of high-level language support for multicomputers is one of the major limitations in applying the computational power of these machines to new applications. Two major developments, which have been driven by VLSI technology, lend urgency to this task: (a) New parallel architectures have proliferated and are increasingly being used by programmers who are not intimately familiar with their low-level architectural details. (b) The design of specialized VLSI multiprocessors to accelerate computationally intensive tasks is becoming more common. The problem of software support for these machines will continue to grow.

The long-term goal of our research is to demonstrate the effectiveness of high-level language support for a microprogrammable multiprocessor designed to accelerate computer aided design tools for VLSI. This research involves the development of a C compiler for a processing element (PE), a C++ compiler for a cluster of PEs, and source-code level simulation and debugging tools.

This paper describes the first stage of that research, the development of a C language compiler for the processing element of the MARS multicomputer [5].

The first step in supporting a high-level language on an MIMD (Multiple Instruction Stream, Multiple Data Stream) message passing machine such as MARS is to be able to generate efficient, accurate code for each PE of the multicomputer. Good code generation at the PE level is needed before any work in compilation for the entire multicomputer can be attempted. The PE compiler can be used directly by a programmer who partitions a problem on to the processors, allocates the communication usage, and balances the computational load. Or, it can be used by a higher-level multicomputer compiler that performs these tasks and generates output code for compilation onto the PEs.

While our work deals with MARS, only a small, easily-identifiable part of our implementation is specific to its architecture. Thus, our method has wider applicability to other microprogrammable processors and can be the basis for development of a retargetable compiler.

There is much reported work on high-level microprogramming and retargetable compilers (see Section 4). The focus of the past work, however, is markedly different from ours. We are concerned with an environment in which a variety of applications are implemented by a programmer who is far removed from the designers of the microprogrammable PE. By contrast, earlier work has dealt with firmware development carried out in the context of a processor design.

2 MARS Background

MARS, a microprogrammable accelerator for rapid simulation, is a message-passing multicomputer with micro-programmable PEs [5]. MARS was developed to accelerate logic simulation of digital circuits, but its generality and programmability allow it to perform a wide variety of problems. The currently implemented MARS applications include logic simulation [4], fault simulation [3], and speech recognition [11]. The architectural features in MARS lend themselves to non-numeric applications such as graph search.

MARS is a message-passing multicomputer with parallelism at three levels. At the highest level MARS is proposed as a hypercube network of processor clusters. The currently implemented hardware represents one of these clusters, and is physically a plug-in VME board. The cluster level of MARS is a collection of 15 PEs and a house-keeper processor. The PEs are microprogrammable VLSI custom processors, each with its own local data memory which is accessible only by it and the house-keeper processor.

MARS Cluster Architecture

The PEs within a cluster communicate with one another via a message passing network which is connected by a 16 × 16 full crossbar. A PE can address the destination of its transmitted messages, transmit a message into the crossbar, and receive messages from the crossbar. All three of these actions are controlled dynamically by the PE's program. The crossbar handles 16-bit messages from PE to PE with only one clock cycle latency. The crossbar messages are buffered at transmission and reception with an eight-message deep total buffer size per PE. The channel control logic includes a channel hold which provides a blocking non-interruptible channel between two PEs.

The house-keeper processor performs I/O and support functions for the cluster. For example, the house-keeper may perform data transfers from one PE data memory to another, or between a PE data memory and disk storage. The house-keeper also handles cluster-wide control such as loading PE programs, starting and stopping PEs (individually or en masse), and receiving and responding to PE level interrupts. The CPU of the host Sun work-station acts as house-keeper in the VME board implementation. Figure 1 shows a block diagram of the MARS cluster architecture.

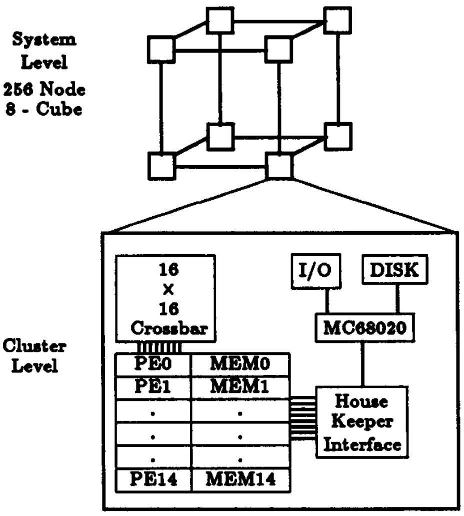


Figure 1: System and Cluster Architecture

MARS PE Architecture

The MARS PE has a parallel architecture and a horizontal (64 bit), writable microcontrol store. The parallel micro-architecture includes several special features. Among these is, the ability to perform arithmetic and logical operations quickly on variably offset bit fields of size 1, 2, 4, and 8 bits using a field operation unit (FOU) which operates independently of the 16-bit address arithmetic unit (AAU). An example bit field operation would be to take bit positions 0 and 1 from register number 16 and bits 7 and 8 from register number 17 add the two pairs as two bit integers, and place the result in bit positions 13 and 14 of register number 18. When properly configured this can be performed by the FOU in one cycle.

A second special feature of the PE is reading and writing bit fields to and from a variable aspect ratio memory. The hardware supports memory access as if the memory were 1, 2, 4, 8, or 16 bits wide. Such an access capability accelerates operations on tables of packed bit fields.

The PE also provides special hardware to transmit and receive words directly over the message passing network. The hardware makes it possible to have two control flow threads active in the PE at the same time. This is managed by hardware which generates a trap when the input buffer is not empty or when the output buffer is not full. This trap switches control to a second thread which can process the data transfer to or from the crossbar and then return control to the main control flow thread.

The writable control store was provided to make MARS a versatile machine instead of a dedicated hardware simulator. The store is 64 bits wide with only 64 entries in the current implementation and can be read and written by the house-keeper. However, writing to the control store causes a hardware reset on the PE, which clears its state and register files. Programmability was included at the microcode level to achieve high-speed processing as a pipeline stage in simulation problems. This horisontal microcode, the multiple buses, and the availability of parallel micro-architecture hardware give the PE the ability to perform up to five operations per clock cycle under the control of the programmer [5].

MARS Software

The software for the MARS project has been developed in several stages. Prior to development of the current compiler, there existed several tools to support micro-programming. First, a functional simulator was developed previous to the production of the hardware and is used for debugging programs. It simulates the 15 PE's and the message passing network in a cluster at the clock phase level. Second, a micro-assembler for translating symbolic micro-programs into microcode files was written. Micro-assembler programming is supported by a macro facility along with a library of macros for simple operations. Third, the House-keeper/PE interface has been supported by a library of system routines for accessing and controlling the MARS VME board as a device. The routines provide a means of transferring data and programs between the host work-station process and the PE memories. Together these tools provided a basic environment for developing several applications, implemented in microcode, which currently run on MARS.

The compiler work for MARS is being performed in three stages. The first stage involves the development of a compiler which inputs the C language and generates microcode. The remainder of this paper describes the design and implementation of this PE level compiler. The second stage of the research will be the design and implementation of a cluster-level compiler which maps an algorithm onto the set of 15 PEs. The third stage of the compiler work will be to explore data parallel methods of mapping algorithms onto multiple clusters at the system level. See Figure 2.

3 PE Compiler Requirements

The VLSI processors used in multicomputers often include specialised hardware features. These features are included to support operations that commonly occur in the problem domain being targeted, to enhance the processor's speed performance, and to support interaction of the PEs in the multicomputer. For example, the MARS PE provides special hardware for bit-field operations, for accessing data memory with a variable aspect ratio, and for interrupting the processor flow of control to handle message traffic.

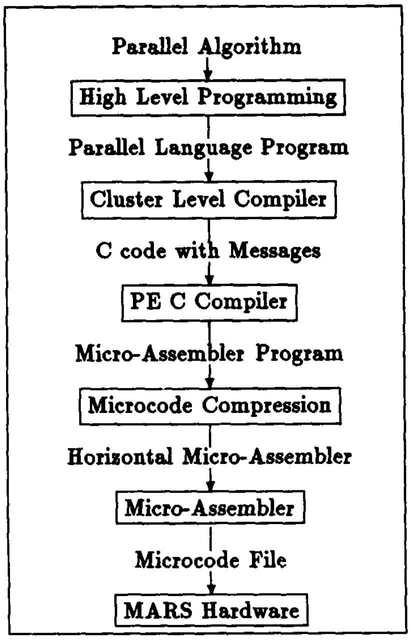


Figure 2: MARS Software Overview

The challenge of the PE level compiler is to represent the features of the hardware in an efficient and easy-to-use manner. The MARS compiler represents the specialized features of the PE hardware to the PE level programmer and/or overlaying tools in a way that is independent of the machine. The PE is supported in a machine-independent manner, in order to reduce the attention the programmer must pay to low-level architecture and to abstract hardware features, thus reducing the complexity of cluster-level compiler code generation. This abstraction is also important for portability of programs between revisions of the hardware, and to different architectures.

The MARS architecture presents three major challenges to writing or generating good microcode.

1. Object Code Compaction:
   The small program memory in the current implementation makes object code size very impor-

tant. Since reducing object code sise reduces execution time [10], object code sise compression is the primary optimisation criteria for the final microcode.

2. Register Allocation:
   MARS provides few registers. There are 8 general-purpose registers, with 24 registers in all. This, combined with the latency of memory accesses, makes good register allocation essential to efficient code generation.

3. Instruction Scheduling:
   MARS has a limited instruction set. It has no complex addressing modes; in fact, the only external memory addressing mode is register indirect (from one of two memory address registers) with an offset of 0 to 7. The compiler must schedule instructions for memory access to keep pace with data traveling through internal pipelines.

Language Choice

The input language of the MARS PE level compiler is the C programming language. The C language was chosen for several reasons. Though a single PE contains parallel hardware, it follows a single flow of control; sequential language is sufficient to represent this control flow. The C language is capable of directly expressing all of the operations of the MARS PE architecture: C bit-fields are used to represent field operation unit (FOU) bit manipulations on 1, 2, 4, and 8 bit entities; and special library functions are made available to provide access to the communication hardware and to interrupt control. C allows for direct description of bit manipulations and low-level operations. C is a good language for automatic program generation by any overlaying cluster-level compiler. Furthermore, it is already known by the community of programmers who will program MARS.

More importantly, software written for MARS can be tested on other C compilers with software simulation of the MARS library calls. The extensions of C needed for MARS take the form of special library function calls which have low-level, high-speed action on MARS. These function calls can be implemented on another system to access a software simulator of the special function, to read and to write from a file, or to perform other debugging activities. Another advantage of C is that compiler front-ends for the language are readily available. which greatly reduce implementation time. Also, using C makes it easier to test and debug the PE compiler by comparing program results against those from other machines.

Compiler Retargeting

A secondary objective of our work is to explore retargetable microcode generating compilers. While our research has not attempted to produce a compiler generator based on a machine description language, the compiler development has isolated compiler translation rules that depend upon the target micro-architecture. (See Section 5 for a complete description of compiler implementation.)

The first requirement of a retargetable compiler is a well designed compiler parser and front-end. Retargetable compiler front-ends generally reduce the input language to a machine independent intermediate code. Within the compiler back-end, intermediate optimisations are performed to customise the intermediate code to the target architecture. These optimisation rules are based on the compiler architecture and are generated by hand in the current implementation. In a retargetable implementation of a microcode generating compiler, the optimisations would be driven by an explicit rule database. This rule set could then be automatically generated from an architecture description. Several such compilers have been constructed for traditional architectures and while these compilers are not directly able to support microcode generation (see section 3) the methods of supporting retargeting should be applicable to microcode generation.

4 Previous Work

Since the construction of a compiler for the MARS PE is by definition a machine specific problem, only solutions to slightly similar problems are found in the literature. Furthermore, there exist many good bibliographies of work in the field of compilers for various parallel machines.

Inadequacy of Traditional Compilers

One approach to applying existing compiler technology to the construction of the PE level compiler was to look at currently available retargetable compilers and consider the feasibility of targeting them to MARS. Retargetable compilers have the ability to generate assembly code for many different architectures. Notable examples of these systems are *pcc*, the standard AT&T UNIX portable C compiler [6], and *gcc*, the Free Software Foundation GNU project C compiler, which have been modified to generate code for many different architectures.

More recently compilers have been developed which are retargetable to a new architecture simply by writing detailed descriptions of the architecture instruction set [14] [15]. Retargetable compilers provide a straightforward method of constructing in a relatively short time a compiler for a new architecture. However, these compilers make certain assumptions defining what the compiler writers consider to be a general purpose computer. Commonly used assumptions are a von-Neumann computer architecture, (with program and data memory and one instruction counter) and instructions of the form "operation, addresses". The address fields in instructions can at different times refer to a constant, a register, an absolute memory location, a memory location addressed by a register with optional constant offset, or a memory location addressed by another memory location. Even RISC architectures, which reduce complexity of operations and addressing modes, still provide addressing modes such as register indirect addressing (into memory) with an offset.

The MARS PE, being microprogrammable cannot be said to have addressing modes in the traditional sense but the following three operations describe the range of possibilities: loading a constant onto a bus, loading the contents of a register onto a bus, and storing the contents of a bus into a register. These operations are of a much lower level than the high-level addressing modes that retargetable code generators require traditional architectures to provide.

VLIW Architecture Compiler

Fisher [13] and Ellis [12] describe the design and production of a compiler for a Very Large Instruction Word (VLIW) Architecture. Ellis describes trace scheduling, a technique for code generation for a VLIW machine. A trace is a path through the flow of control graph of the program. In trace scheduling, instructions are reordered and scheduled within a trace rather than within a basic block because more code reordering can be found within a trace. Reordering the code within a trace involves moving instructions across basic block boundaries. This movement causes inconsistencies between the code trace and the basic blocks adjoining it in the control-flow graph. Correcting these differences requires additional code in the basic blocks that flow into and out of the trace being scheduled.

A second recent compiler technique is described by Lam [19]. Lam describes a compiler for a systolic array of microprogrammable processors which uses a technique called software pipelining. Software pipelining considers the code resulting from a source language function as a directed graph, with nodes representing operations and edges representing flow of control. Software pipelining first schedules the instructions within an inner loop of a program. The completely scheduled instructions are then collapsed into a single node and the scheduling continues. The scheduling algorithm and node collapsing are repeated until the entire graph is one node and the function is scheduled.

High-Level Microprogramming

Hopkins, Horton and Arnold [17] describe a system for high-level microprogramming which is similar to ours but with the aim of producing microcode firmware which will execute a higher-level machine language on the hardware. Their objectives are to support the development of firmware programs across evolutionary changes of the underlying hardware and to produce tools that can be retargeted to new architectures. This is in contrast to the case in MARS, where the goal of the compiler is to target application programs directly to microcode. Since the users of our compiler are not familiar with the details of its microarchitecture, it is important for the compiler to hide the low-level details of the architecture from the application program.

Hopkins et al. describe the design of a microcode producing compiler whose input is a subset of the C language. A good case is made for writing firmware programs in C instead of directly in microcode. Admittedly there is a code-size and run-time overhead in the use of a high-level language but it is pointed out that recoding just 10% of a program in efficient hand microcode often reduces a 100% run-time overhead to just 10%, (this is sometimes called the 90/10 rule). In the case of MARS we find that the 90/10 rule is doubly valid. First, within one PE's microprogram, inner loops can be identified which account for much of the processing time. Second, across a multiple-PE program – often organized as a pipeline – one or two stages may be identified that act as bottlenecks in the system. If these PE programs are rewritten in efficient microcode overall performance will be improved and further optimization of the non-critical PEs is not beneficial.

5 Implementation

Figure 3 shows a block diagram of the compiler. The PE Compiler is organized much like the compiler for a conventional architecture with a few additions. The

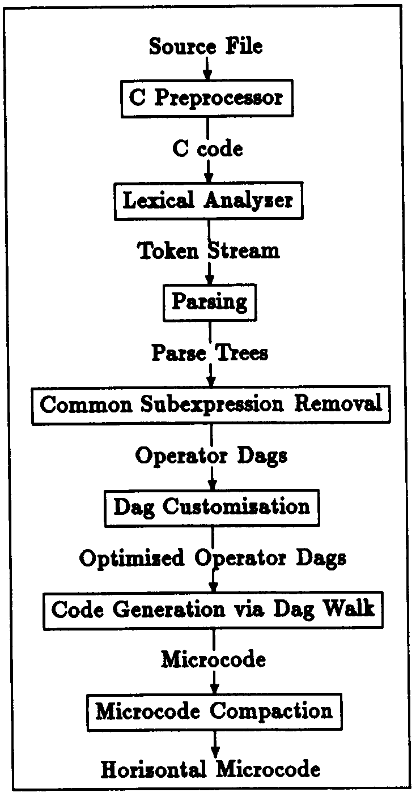


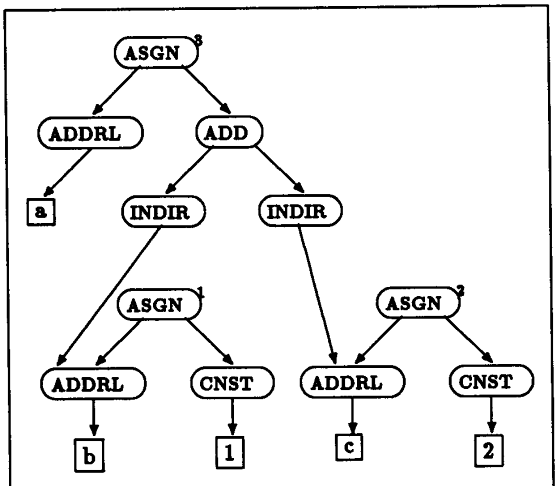
Figure 3: Compiler Block Diagram

first part of the compilation process (lexical analysis, symbol table construction, parsing, parse tree generation, and intermediate code generation) is carried out by the compiler front-end. This front-end is from a portable ANSI-standard C compiler [15]. The use of a previously developed compiler front-end greatly reduces compiler development work. It also helps to ensure an accurate implementation of the language.

After translating a section of the input program, the front-end produces a directed acyclic graph (DAG) representation of the code. The nodes of the DAG represent operations, and the edges represent data dependencies. A sequence of these DAGs is the intermediate code which the front-end passes to the compiler back-end for microcode generation. Each sequence of DAGs is a data-dependency graph for a flow-control-free basic block of code. The ordering of a DAG sequence represents normal control flow. Labels which are the targets of a branch may separate DAG sequences, and a DAG sequence may end with a conditional or unconditional branch. A function call may appear as a node in the DAG.

```
1:    f() {
2:        int a, b, c;
3:        b = 1;
4:        c = 2;
5:        a = b + c;
6:        while (a>0)
7:            a=a-1;
8:    }
```

Figure 4: Example C code



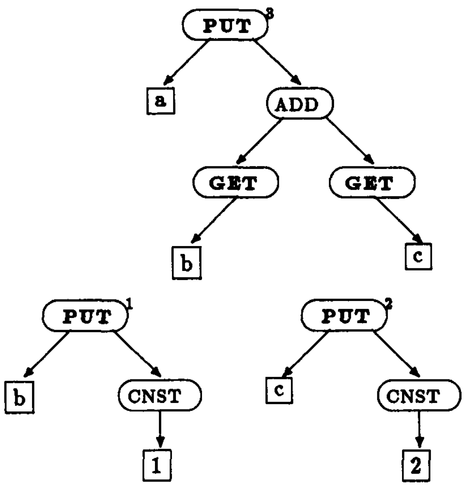
Numbers by the root nodes indicate the DAG sequence.

Figure 5: DAG from lines 3 – 5 of the example C code

For example, when the C code listed in Figure 4 is parsed by the front-end, the code in lines 1 and 2 gives rise to a series of back-end calls. These communicate a function begin, a block begin, and three local variable definitions. Lines 3, 4 and 5 translate into a DAG sequence. The while loop in lines 6 and 7 is translated to test and branch code containing another two DAGS. Line 8 generates a block end and a function end call to the back-end. Figure 5 shows the DAG sequence resulting from lines 3, 4 and 5.

DAG Customization

When an intermediate-code DAG sequence is emitted from the front-end, the operations represented by the nodes are from a relatively small set of 38 operations over 8 supported data types. These machine independent operations represent a RISC-like machine model that has 3 address instructions (e.g. A := B *op* C) and no complex addressing modes. The 8 data types supported by the intermediate code DAG sequences



Numbers by the root nodes indicate the DAG sequence.
Bold face Indicates MARS-specific DAG node operations.

Figure 6: Customized DAG from lines 3 to 5 of the example C code

are: character, short integer, integer, unsigned integer, float, double, structure, and pointer.

The first step in the compiler back-end is to customize each DAG to the PE architecture. In general terms this process involves searching for opportunities to take advantage of the special hardware features available in the PE. Figure 6 shows the DAG from Figure 5 after it has been customized for MARS. During the customization process, the DAG node operations are members of the set which is the union of the front-end generated operations and the MARS-specific DAG node operations. MARS-specific operations represent hardware features that exist in a MARS PE but not in the generalized machine model for which the front-end produced code. Example MARS-specific DAG node operations are: GET (fetch value from an address), PUT (store value to address), INC (increment), DEC (decrement), and BZ (branch if zero).

The bit-field datatypes are very well supported in MARS and must be added to the operator set produced by the front end. A bit-field may be 1, 2, 4, or 8 bits long. MARS can perform all integer arithmetic operations and memory load/stores on bit-fields as easily as on 16-bit integers by using the field operation unit and variable-aspect-ratio memory hardware.

Other alterations made during the DAG customization are performed not so much to improve the efficiency of the code directly but to ease subsequent code generation. They also reduce the improvements required from the later code optimization.

The DAG sequence is customized by traversing through each node in the graph, and attempting to match the current node (and the subtree of nodes immediately under it) against various templates. A template is a matching rule for a small section of tree. For example there is a template/substitution rule which says: "If an ADD has one argument which is a CNST whose symbol is 1, then dereference the CNST node, and change the ADD node to an INC node."

When a section of the DAG matches a template it is substituted with another section of DAG using PE operations. The substituted section is then rescanned for any other possible matches. Specifically, the DAG customization algorithm is:

- Form a list of all nodes in a DAG sequence in prefix order
- For each node, N, in this list
  - For each substitution template
    - If a template match is found, perform the corresponding substitution
  - If any match was found at node N, retest all templates

Dereferencing the CNST node means that the pointer for its use here is removed and its reference count reduced. If a node's reference count is reduced to zero then it is completely removed from the graph and all of its children are dereferenced. The time complexity of this graph search is given by:
$|\,DAGNodes\,| \times |\,Templates\,| \times |\,Transforms\,|$
In the worst case, $|\,Transforms\,|$ is bounded by:
$|\,RulesTemplates\,|^2$
Where *DAGNodes* is the number of nodes in the current DAG, *Templates* is the number of templates to be attempted, and *Transforms* is the number of successful template matches and DAG transformations.

Microcode Generation

After the DAG sequence has been customized for MARS architecture, microcode is generated from it. The code generation follows a straightforward algorithm: For each DAG in a DAG sequence, a code generation routine *gen_node* is called on the root node of the DAG. This routine takes two arguments: the node for which to generate code, and the location at which to place the result of the operation.

The *gen_node* routine performs as follows: The node is first checked to see if it has been processed by *gen_node*, and its result value left in a register. If the value was left in a register, code is produced to move the value to the current target location. At this point processing for the node is complete.

If no stored value from a previous processing step is found, the node is checked to see if it is referenced more than once. If it is multiply referenced, and the requested destination for its results is not a general DAG-node temporary register, then a general DAG-node temporary register is allocated to the node to be stored for later use. The next action varies according to the node's operation. In general, for each child of the node code generation for the child is recursively requested by calling *gen_node* and taking the child node as the node argument. The result placement argument for *gen_node* is the location from which this operation could best use the child's result value. After code has been generated for all of the children, the code to perform the current node's operation is produced. Finally, the results from this operation are moved from where this operation creates them to the node's storage location (if any) and to the location requested in the code generation call.

This code generation algorithm was designed to balance the need to use as few registers as possible for DAG temporaries with the need to reduce the amount of code generated simply to do data transfers. Many of the operations represented by DAG nodes can be performed in several different ways by the PE hardware. The choice of method is based on where the result is to be placed after the operation is finished.

Microcode Compression

As MARS assembly code is generated in the code-generation pass, peephole optimisation is performed on it [6] [10]. Peephole processing performs small localised changes on the generated code; it does not alter the operations performed, but simply changes the instruction sequencing. The optimisation identifies adjacent instructions that have no data dependencies between them and no hardware conflicts. If two instructions fitting these properties are found, they are merged together into one instruction. This post-processing increases the utilisation of the horizontal micro-instruction available in MARS.

```
      $default$ default_inst
      $origin$ 0
      const_fmt const_b(__sp_base_address)
          b_dst(SP) ;
:_f:  const_fmt const_a(d1) a_dst(R_22) ;
      const_fmt const_a(d2) a_dst(R_21) ;
      c_src(R_22) a_src(R_21) aau_add
          c_dst(R_23) ;
      const_fmt const_a(:12:) a_dst(PAR) ;
      nop ;
:11:  a_src(R_23) aau_dec c_dst(R_23) ;
:12:  const_fmt const_b(:11:) b_dst(R_16) ;
      b_src(R_23) c_src(R_16) c_dst(PAR)
          c_disable b_pos ;
      nop ;
:END: HALT
      $end$
```

Figure 7: Microcode from Example C code

Optimising the use of the horizontal control of the PE microcode is done as a post-processing step to avoid complications in the code generation algorithm. A drawback of this approach is that artificial data dependencies are introduced into the code during code generation. These data dependencies result from variables sharing registers in a sequential manner. Since the horizontal compression optimiser works on the generated microcode and not on the original operation DAG multiple uses of a register by different variables are seen as data dependencies and thus limit possible parallelisation. Figure 7 shows the microcode generated by the compiler for the C code listed in Figure 4.

6 Results

The PE compiler is currently being tested by implementing various applications under it. Table 1 gives a list of program segments that have been compiled. The table gives the lines of C code for each program file, followed by the lines of microcode generated under three different levels of optimisation. In optimisation "O0", all optimisation is turned off. Optimisation "O1" enables DAG rewrite rules that place local variables in data registers and optimise the accesses to them. Optimisation "O2" enables DAG rewrite rules that substitute increments and decrements for small additions and subtractions, and DAG rewrite rules that substitute shifts and adds for multiplication by a constant. "O2" is the default optimisation level; other values are given for comparison only.

Test file name	Lines of C source	Lines of microcode Optimization Level		
		O0	O1	O2
count1	11	102	17	16
count2	16	170	33	32
div	19	89	62	56
mod	20	107	79	79
mult	18	96	67	42
sieve	39	221	81	78
small	8	36	16	14
table	22	200	44	43
test	9	7	7	7
xbar	5	9	8	8
squares	17	101	44	41
sumtbl	19	108	42	38

Table 1: Code size using different optimizations

Of the program segments listed in Table 1 several are of special note. “mult” and “div” show the greatest gains from optimization level “O1” to “O2”. Each of these programs contains several multiplication or division operations where one argument is a constant. These operations are reduced to shorter sequences of shifts and adds.

Many different code segments in the table, particularly “count1”, “count2”, and “table”, show large improvements from optimization level “O0” to “O1”. These programs contain many accesses to local variables which can be placed in registers. Since address calculations and memory fetch must be explicitly coded in microcode, each fetch or store that can be eliminated saves several instructions.

The compiler is also being tested by comparing compiler generated microcode programs with assembler code produced by hand. Table 2 lists five programs that have been written in C and in micro-assembler. Each program was compiled using the C compiler, run on MARS Hardware and timed against a real-time clock. The microcode resulting from each compilation was then reorganized and optimized by hand, and the resulting programs were run and timed.

The test results for these and other programs indicate that the compiler generates code about twice the size of hand generated code (100 % overhead). Furthermore, the run-time data gathered for these programs show that run-time overhead is approximately the same as code size overhead.

Further optimizations are being added to the compiler. The peephole horizontal compressions are being implemented. DAG optimization templates are added to the DAG rewrite section by identifying hand-improvements which can be made in the compiler output. It is hoped that further tuning of the optimizations will reduce the microcontrol size to the goal of one and one-half times (50% overhead) hand-generated code. This may seem severe to anyone familiar with 0 – 25% code overhead of the compilers for traditional sequential architectures. The additional overhead for MARS results from targeting microcode instructions instead of a higher-level assembler.

Test file name	Lines of C source	Lines of microcode		Run Time in milli-seconds	
		cc	hand	cc	hand
copy	20	23	12	100	42
trans/rec	41	40	21	80	43
bubble	28	45	27	52420	31190
indexed	39	54	36	19940	13000
sieve	37	55	29	320	130

Table 2: Code size and run time, compiled vs. hand optimized

7 Summary

Special purpose multicomputers often use custom VLSI processing elements. The processors incorporate special architecture features to speed up targeted applications. These features represent a challenge to effective compiler design. The MARS PE compiler was designed by using retargetable compiler techniques and extending them for the special features of the MARS architecture. A reusable compiler front-end, including lexical analyzer, parser, and symbol table is used to generate intermediate code, which is stored as sequences of DAGs. These DAGs are customized to fit the MARS instruction set and then translated into microcode using a DAG walking algorithm. Peephole optimization is then performed on this microcode.

Test results indicate a 50% – 100% code size overhead in compiler vs. hand-generated microcode. Run-time overhead is also in the same range. These numbers are higher then commonly encountered in compilers for traditional RISC and CISC architectures due to the difficulty involved in microcode generation. Further optimizations, currently being implemented, are expected to bring the average code and run-time overhead to the 50% range.

Bibliography

[1] S. Abraham and K. Padmanabhan. Instruction reorganisation for a variable-length pipelined microprocessor. In *IEEE International Conference on Computer Design*, volume ICCD-88, 1988.

[2] D. P. Agrawal and J. Mauney. Structure of a parallelising compiler for the B-HIVE multicomputer. In *Microrocessing and Microprogramming*. North-Holland, 1988.

[3] P. Agrawal, V. Agrawal, and K. T. Cheng. Fault simulation in a pipelined multiprocessor system. In *Proceedings of the IEEE International Test Conference*, volume ITC-89, pages 727–734, 1989.

[4] P. Agrawal and W. J. Dally. A hardware logic simulation system. IEEE Transactions on Computer-Aided Design, 9(1):19–29, January 1990.

[5] P. Agrawal et al. MARS: A multiprocessor-based programmable accelerator. IEEE Design & Test of Computers, 4(5):28–36, October 1987.

[6] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[7] A. Aiken and A. Nicolau. A development environment for horizontal microcode programs. In *Proceedings Micro-19*, 1986.

[8] R. Allen and S. Johnson. Compiling C for vectorisation, parallelisation, and inline expansion. In *Proceedings of the Conference on Programming Language Design and Implementation*, volume SIGPLAN-88, pages 241–249, 1988.

[9] R. P. Atkin. Improved instruction formation in the exhaustive local microcode compaction algorithm. In *Proceedings Micro-17*, 1984.

[10] D. K. Banerji and J. Raymond. *Elements of Micro-Programming*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1982.

[11] S. Chatterjee and P. Agrawal. Connected speech recognition on a multiple processor pipeline. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1989.

[12] J. Ellis. *Bulldog: A compiler for VLIW Architectures*. The MIT Press, Cambridge, MA, 1986. Originally presented as the author's thesis (doctoral) – Yale University, 1985.

[13] J. Fisher. Trace scheduling: a technique for global microcode compaction. IEEE Transactions on Computers, C-30(7):478–490, July 1981.

[14] C. W. Fraser. A language for writing code generators. In *Proceedings of the Conference on Programming Language Design and Implementation*, volume SIGPLAN-89, pages 238–245, June 1989.

[15] C. W. Fraser and A. L. Wendt. Automatic generation of fast optimising code generators. In *Proceedings of the Conference on Programming Language Design and Implementation*, volume SIGPLAN-88, pages 79–84, 1988.

[16] R. Gurd. Experience developing microcode using a high level language. In *Proceedings Micro-16*, 1983.

[17] W. C. Hopkins, M. J. Horton, and C. S. Arnold. Target-independent high-level microprogramming. In *Proceedings Micro-18*, 1985.

[18] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the Conference on Programming Language Design and Implementation*, volume SIGPLAN-88, pages 318–328, 1988.

[19] M. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, Boston, MA, 1989.