A C Compiler for HPS I, a Highly Parallel Execution Engine

Michael C. Shebanow
Yale N. Patt
Wen-mei Hwu
Steve Melvin

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Historically, compilers have been able to translate high level sequential programs into a parallel form which is useful for generating optimal code, but then have subsequently thrown away that parallel information with the result that the final output is sequential target machine code. In our case this would be particularly unfortunate since our target machine HPS I can execute concurrently the operations which comprise the parallel form that the compiler started with. This paper describes our work in compiling C programs for the HPS I. The objective is to enable the HPS I to exploit the irregular parallelism available in the source program.

1. Introduction.

1.1. The Computing Engine.

The HPS I is a high performance computing engine which achieves its high performance, in part, by concurrently executing nodes from a data dependency graph. Each node consists of an operation, data dependency information pertaining to its operands, and a descriptor which is used by subsequent nodes which are dependent on this node's result. HPS I processes instructions in the form of multi-node words, where each node corresponds to a different type of operation (e.g., an integer operation, a floating point operation, and a memory access operation).

The significance of the HPS I is that it is being designed to exploit a new model of execution, which we have identified as restricted data flow (RDF).

1.2. The Model of Execution.

The principle behind RDF is as follows: Translate a "piece" of a sequential program into a data dependency graph. By "piece" we mean some fixed number of instructions. We call this piece the "active window." Execute the data dependency graph corresponding to the active window. As instructions complete execution, they are removed from the active window and new ones are brought in. These new ones are translated and merged into the data flow graph. This method is termed restricted, as at any one time, only a limited number of data flow nodes may be present in the microengine.

To be successful, the above model must allow for out of order execution of the data flow nodes. HPS I allows out of order execution by register and memory aliasing, as in the Tomasulo algorithm [17].

1.3. Why C?

C is a versatile language. Its use ranges from the UNIX operating system [15] to real-time programming applications. Corporations and educational institutions alike use it as the basis for much of their software. C's popularity stems from its flexibility and its expressive power.

All work in 'C' is expressed in terms of functions. 'C' is single level; it does not allow functions to be statically nested. It has a very simple activation record format, which makes it fast. 'C' permits variables to be initialized, which allows predefined static data to exist. This is used to advantage in software generating tools in UNIX, such as YACC [9, 10]. 'C' is very expressive; there are many ways to express the same function. These characteristics make 'C' a good implementation language.

1.4. Justification for this Project.

We are developing a 'C' compiler which will allow us to bypass the Von Neumann instruction set and go directly into data flow nodes. We want to eliminate the inefficiency involved in the translation from internal...
dependencies come in the classical forms: read-write, write-read, and write-write. It must also cope with branch points in the instruction, context switching, and interrupts and exceptions. This section shall describe briefly how the HPS is able to correctly implement the sequential model of known Von Neumann model.

An abstract view of HPS is shown in figure 2. Instructions are fetched and decoded from a dynamic instruction stream, shown at the top of the figure. The figure implies that the instruction stream is taken from a sequential control flow ISP architecture. There is no reason, however, and indeed the thrust of our work with C is that the static instruction stream can consist of single-entry/single-exit DAGs. These DAGs can be brought into the dynamic I-stream via the use of a branch predictor as multi-node words, and subsequently merged into the restricted data flow microengine. In all implementatations, the output of the decoder which is presented to the Merger for handling by HPS is a data flow graph.

The importance of the branch predictor in this scheme must be emphasized, since (unlike the piecewise data flow model of Requa and McGraw (14)) we allow out of order execution to take place across branch boundaries.

1.5. Outline of this paper.

This paper is organized in six sections. Section 2 describes HPS I in greater detail. Section 3 describes an ISP architecture which will interface C to the HPS engine. Section 4 shows an example of a C function compiled into this architecture. Section 5 discusses the various aspects of the compiler. Section 6 offers some concluding remarks.

2. HPS I.

2.1. Overview.

We are calling our engine the HPS I, which stands for High Performance Substrate. We will implement the engine using our own model of a data flow engine, which we called Restricted Data Flow (RDF). Our model of the microengine is not unlike that of Dennis [5], Arvind [3], and others, but with some very important differences. These differences will be discussed in detail in section 2.1.

The basic idea behind HPS is to gain as much parallelism from the source program as possible, while still maintaining the sequentiality of the instruction stream. HPS assumes that in order to maintain sequentiality, it must cope with data dependencies implied by a register or memory architecture. These
A very important part of the specification of HPS is the notion of the active instruction window. Unlike classical data flow machines, it is not the case that the data flow graph for the entire program is in the machine at one time. We define the active window as the set of ISP instructions whose corresponding data flow nodes are currently being worked on in the data flow microengine.

As the instruction window moves through the dynamic instruction stream, HPS executes the entire instruction stream. Parallelism which exists within the window is fully exploited by the microengine. This parallelism is limited in scope; ergo, the term "restricted data flow."

The Merger takes the data flow graph corresponding to each ISP instruction and, using a generalized Tomasulo algorithm to resolve any existing data dependencies, merges it into the entire data flow graph for the active window. Each node of the data flow graph is shipped to one of the node tables where it remains until it is ready to fire.

When all operands for a data flow node are ready, the data flow node fires by transmitting the node to the appropriate functional unit. The functional unit (an ALU, memory, or I/O device) executes the node and distributes the result, if any, to those locations where it is needed for subsequent processing: the node tables, the Merger (for resolving subsequent dependencies) and the Fetch Control Unit (for bringing new instructions into the active window). When all the data flow nodes for a particular instruction have been executed, the instruction is said to have executed. An instruction is retired from the active window when it has executed and all the instructions before it have retired. All side effects to memory are taken care of when an instruction retires from the active window. This is essential for the correct handling of precise interrupts [2].

The instruction fetching and decoding units maintain the degree of parallelism in the node tables by bringing new instructions into the active window, which results in new data flow nodes being merged into the data flow node tables.

2.2. Instruction Flow.

Figure 3 illustrates the global data path of HPS I. Instructions enter the data path as input to the Merger. This input is in the form of a data flow graph, one per instruction. The data flow graph can be the result of decoding an instruction in a classical sequential instruction stream, or it can be the output of a non-conventional compiler, as is the case here. In either case, the Merger sees a set of data flow nodes (and data dependencies), one for each operation that must be performed in the execution of that instruction. Computation and ALU functions.

The Merger, using the Register Alias Table to resolve data dependencies not explicit in the individual instruction, forms the set of data flow nodes which are necessary to execute the instruction. Nodes are then transmitted to the appropriate node tables. Node tables are content addressable memories, and thus should be kept small. The size of each node table is a function of the size of the active window and the decoding rate of the Von Neumann instruction stream. In our experiments with the VAX architecture, for example, an active window of 16 instructions, coupled with a decoding rate of eight nodes per cycle, required at most a 35 entry node table.

For each node, a slot is reserved in the global multi-port value buffer for storing the result of the operation of that node. The index of each slot is designated as a tag for the corresponding node, and is carried along with the node until it completes its execution. Value buffer slots are assigned in a circular queue, the size of the buffer being large enough to guarantee retirement of an instruction before its value buffer slot is again needed.

A node remains in its node table until all of its operands are available, at which point it is ready to fire (i.e., it is executable). A node is fired by transmitting its operator, tag, and set of operands to one of the functional units associated with that node table. When execution completes, the result and its tag are distributed to each port of the value buffer. In the case of a result destined for a general purpose register, the corresponding tag is also transmitted to the Register Alias Table to update.
information stored there. The corresponding tag is also transmitted to the node tables for the purpose of setting the ready bits in those nodes awaiting this result.

Memory read and write nodes present additional complications. Although these will be discussed in greater detail in [13], a few observations here are in order. First is the fact that at the time memory access nodes are issued by the Merger (depending of course on the addressing structure of the target architecture), the address of the memory access may be unknown, and the addresses of other memory accesses which could block the node being issued may also be unknown. A Memory Alias Table and a Read Staging Unit are provided to handle these problems. Second is the fact that writes can occur out of order coupled with our requirement that exception handling must allow the machine state to be recovered "precisely." A Write Buffer and an algorithm for retiring instructions are provided for handling this problem.

One final observation about the processing of nodes must be made. The stages that a node goes through (i.e., merging, waiting for operands, firing, executing, and distributing its results) is independent of the other nodes in the node tables. That is, for example, the number of nodes firable in a given cycle is limited by the ability to detect that multiple nodes are firable and the number of functional units available for concurrent processing of nodes. The number of results that can be distributed in a single cycle is a function of the bus structure and the organization of the node tables. The intent is that in each cycle, multiple nodes will be in each stage of the process.

3. An Architecture Tailored for C.

This section describes an architecture specifically suited for is designed to properly interface C to the HPS engine. It has a 32 bit address space, comprised of two gigabytes of virtual memory (per user) and 2 gigabytes of shared virtual system space. User virtual space contains both program and data. In addition, two other address spaces exist: a frame space and a context space. The frame space is similar to the D registers used on the B6500 [8]. It consists of a set of frame pointers, each of which is allocated to a static nesting level in the user's source program.

Context space is used for storing procedure context information after every call. This information is kept separate from the user's data, since it should be read only and accessible only to the hardware. If we included this information within the user's data stack, we would unnecessarily complicate the instruction prefetcher during the execution of a return. That is, because the return information was in the data stack, the prefetcher would attempt to resolve data dependencies in a situation where no data dependencies should occur.

3.1. Instruction Representation.

There are two classes of instructions: control and data manipulation. A control instruction specifies the location of the DAG, if any, to be executed, and a pointer to the successor control instruction. A data manipulation instruction specifies the evaluation of an expression.

Data manipulation instructions may not change the flow of control in the machine. In C, all expressions are evaluated with data manipulation instructions, except for [ ], &&, ?, and function calls which also require control instructions because they involve a change in the instruction flow.

3.2. Format of Control Instructions.

The control block of a function consists of a function header, followed by one or more control nodes. The function header creates the context for the control nodes. It contains the nesting level of the function, its local frame size, and a register save mask. Each control node specifies two things: an action to be taken and the next step after that action. The format of a control node and a function header are shown in Figure 4. Note that each consists of exactly 64 bits. (Consequently, the minimum number of bits required to express a function is 128 bits, or four longwords.)

<table>
<thead>
<tr>
<th>Function Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNL</td>
</tr>
<tr>
<td>R_SAVE</td>
</tr>
<tr>
<td>I_SAVE</td>
</tr>
<tr>
<td>FRAME_SIZE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPCODE</td>
</tr>
<tr>
<td>R_CNTL</td>
</tr>
<tr>
<td>OFFSET</td>
</tr>
<tr>
<td>BLK_POINTER</td>
</tr>
</tbody>
</table>

Fig. 4  Control Node Format

The field SNL is the static nesting level of the function. R_SAVE and I_SAVE specify the ranges of real and integer registers to be saved. FRAME_SIZE is used to determine how much stack space will be allocated to this function for local variables. In a control node, the opcode determines the control operation. R_CNTL specifies an integer register which may be used as part of the control operation. OFFSET is a signed longword displacement (i.e., it is scaled by 2 bits). It may be used in a call or a branch. The BLK_POINTER has two uses. If the short offset is insufficient, the BLK_POINTER can be used as the offset instead. Certain opcodes are provided for that purpose. In other cases, the BLK_POINTER points to the DAG or SYS block to be executed.

There are two general types of control nodes, DAG and STS. A DAG specifies that a DAG is to be executed. It references a DAG block. A DAG block contains a count of the number of longwords used to represent the DAG, followed by the nodes which comprise the DAG (explained in the next section). Like other control nodes, the DAG control node also provides a pointer to the control node to be executed after the DAG, which may be (for example) a conditional branch or a cell. A SYS
control node specifies a system action to be performed. The action is a pointer to a system instruction.

Two control node classes are of interest, one related to branching, the other related to function calls. Two opcodes are provided to reference a call frame. This is used to implement the setjmp/long jmp functions defined in the standard C library. The classes are shown below:

**Call Class**
call $<\text{dst}>$
call $(<\text{Rn}>)$
ret $<\text{Rn}>$
ret $<\text{blk}>$

**Branch Class**
br $<\text{dst}>$
br $(<\text{Rn}>)$
brt $<\text{Rn}>, <\text{dst}>$
brf $<\text{Rn}>, <\text{dst}>$

In the call class, two types of calls are provided; one uses relative addressing, the other uses indirect addressing via a register. Three types of returns are provided. One returns a real or integer value via a corresponding real or integer register. This is accomplished by the return mechanism on the stack. A second allows a block of data to be copied as a return value. The third, VRET, returns nothing (it is a void return).

In the branch class, there are only two forms of conditional branches: branch true (brt) and branch false (brf). Both use the contents of an integer register to determine the branch condition, and absolute addressing to specify the target address. A register is considered false if it contains 0 and true otherwise. Two unconditional branches are included in the set of control nodes; one uses absolute addressing and one uses register indirect addressing. A cont instruction allows a control node to just continue on to the next sequential control node.

### 3.3. Format of Data Manipulation Nodes

Data manipulation instructions (or DAG instructions) specify data operations. These are the only instructions which actually modify the state of main memory. The machine supports both monadic and dyadic operations. We have chosen an instruction encoding which allows us to design a fast parallel decoder. All DAG nodes, with one exception, are the same width (32 bits). The single exception is the case where a long offset is needed by an instruction. In that case, it immediately follows the instruction. A simplified sample decoder is shown in figure 5.

In this diagram, "R" is a register delay and "D" is a decoder. At worst, a node may need to reference a longword following it. These are represented by the diagonal paths to the left. If an instruction needs the longword to the right, the corresponding decoder inhibits the decoder to the right from issuing a node. This scheme allows a pipeline to be formed which will process all nodes plus constant offsets in time proportional to the number of nodes plus constants with a latency of one cycle.

Instruction formats are shown in figure 6; field formats are shown in figure 7.

All types can be distinguished by OPCODE. Monadic, dyadic, and literal nodes have RDEST fields. This field specifies which register (if any) should store the result of an operation. The source fields of the monadic (SRC) and dyadic (SRC1, SRC2) nodes specify the source operands. Sources can be either register contents or results of other nodes. This representation allows up to 128 nodes to be defined in a DAG. All DAG node addresses are relative to the base address of the DAG.

Literal nodes allow small constants to be easily entered into the machine with no indirection. The literal value can be signed, unsigned, or converted into a floating point value. In addition, a SRC field may specify a short literal, which is very useful for small constants which can be expressed with no more than six bits. If the N/R bit is set, then the source is taken to be the 7 bit number including the LIT bit. This allows up to 128 nodes to be referenced. If it is reset, then the LIT bit determines if the 6 bit field is a register number or a literal value.
Fig. 7. Data Node Field Formats

Memory nodes reference memory locations by means of a short or long offset to either the DB (the DAG Base address) or FP (the frame pointer of the current static nesting level CSNL). The OPSPEC field provides this information.

3.3.1. Dyadic Operations.

Dyadic operations operate on either nodes or registers. As there are two types of registers, real and integer, in general two types of nodes are provided. The type of the source and destination registers are implicit in the opcode. There are four types of arithmetic dyadic nodes: real compare, real arithmetic, integer compare, and integer arithmetic. The Compare operators always produce the integer result zero or one (which represent false and true), independent of the sources. (real or integer). Unsigned compare for less than and less than or equal is also provided for the unsigned data types. Real and integer arithmetic operators include add, subtract, multiply, and divide, plus an integer remainder operator. In addition, the logical operators AND, OR, XOR, and signed and unsigned leftshift and rightshift are provided. Finally, three bit field operators are provided: insert, signed extract, and unsigned extract.

3.3.2. Monadic Operations.

Monadic operators take one operand and produce one result. These operators include negate, square root, unpacking of floating point numbers, log and antilog (for priority encoding and decoding for operating system support), coercion operators (trunc8, trunc16, ext16, ext32, toreal, toint, to allow changing between machine data formats), and logical negation (NOT produces a 1 if its source is a zero, and a zero otherwise).

3.3.3. Literal operations.

Literals are provided to efficiently introduce constants into the machine. The format is value = literal. If a destination register is selected, then the operation is value = R_DEST = literal. Three types of literals are provided: signed, unsigned, and real. In the first and last case, the value field is treated as a short signed quantity. In the case of unsigned, it is not sign extended. We have not supplied a long literal node because we do not feel its frequency of occurrence justified it. For those occasions where we need a long literal, an address node in combination with a read memory node can produce it.

3.3.4. Memory Operations.

There are two types of memory nodes: address nodes and access nodes. An address node produces an address which can then be used by an access node to access the memory of the machine. An access node specifies the access (read or write) and which of the eight standard data types is to be accessed. The cost of not unifying address nodes with memory nodes is not as great as it may seem, since addresses within the DAG are generally reusable; they need to be calculated only once in the DAG. If they are needed between DAGs, they can be placed in registers.

An access node can read or write a single value, or can combine to execute a block move. Single access reads and writes are done by the monadic and dyadic operators. The block move is subdivided into two separate operations: one which reads a block (creates an internal reference number for it) and another which writes the block. Both operations are dyadic.

3.4. Data Representation.

The architecture supports eight external data types, and three internal data types. The external data types are s8, s16, s32, u8, u16, u32, f32, f64. The letter in front of the type stands for signed, unsigned, and float, respectively. The number gives the number of bits used to store the value. The three internal data types are s32, u32, and f30. When data is transferred from memory to the data path of the microengine (and vice versa), it is converted from external form to internal form (again, and vice versa). This significantly reduces the number of operations which need to be supported, while increasing the range and precision available to numeric algorithms. Programs are not allowed to store f30 data in memory, as this would unnecessarily complicate the data path and normal stack format. However, during a context switch, the full 80 bit values are saved.

4. An Example.

We are now ready to show how a simple piece of C code can be represented by the architecture of the previous section.

This example takes an array as an argument and sorts it into ascending order. Figure 8 shows the control nodes and figure 9 shows the data manipulation nodes which make up the bubble sort function:

In figure 9, IX refers to integer register X. The C compiler has removed (optimised) the variable temp from the code. Variables array, size, i, and changes are allocated to 10 through 13. Registers 14 to 18 are used as temporaries by the compiler.
bubble_sort(array, size)
/* BUBBLE_SORT */
register short *array;
/* array of items to sort */
register int size;
/* size of array */
{
    register int i;
    register short temp;
    register int changes;

do {
    /* sort array */
    changes = 0;
    for (i = 1; i < size; i++) {
        if (array[i - 1] > array[i]) {
            /* in order? */
            temp = array[i];
            array[i] = array[i - 1];
            array[i - 1] = temp;
            /* elements */
            changes = 1;
        }
    }
} /* end for */
/* until no change */
}

The program starts with DAG1, which transfers the parameters for the bubble_sort from the stack to the registers. Addresses required by memory read nodes are generated by integer operators. DAG2 initializes the for loop. DAG3 tests the looping condition for the for loop. DAG4 evaluates the expression for the if statement. DAG5 swaps the two array elements and then sets the register allocated to changes. In DAG3 and DAG4, lit_s stands for the less than relational operator. In DAG2 and DAG5, lit_s stands for a literal node. In DAG4, shl_s stands for shift left (by the first quantity).

| SNL=0, R_SAVE=none, I_SAVE=9 |
| FRAME SIZE=0                 |
| c0:                           | cont    | dag1 |   |
| c1:                           | cont    | dag2 |   |
| c2:                           | brf i4, c5 | dag3 |   |
| c3:                           | brf i4, c2 | dag4 |   |
| c4:                           | br c2   | dag5 |   |
| c5:                           | brt i3, c1 | NULL |   |
| c6:                           | vret    | NULL |   |

**Fig. 8. Control Nodes for the Example of Section 4**

**Fig. 9. Data Nodes for the Example of Section 4**

5. The Compiler.

5.1. Overview.

Our compiler implementation is fairly standard. It consists of a front end, which does the parsing and semantics, followed by a backend which outputs code for HPS I. The front end utilizes some very advanced error recovery techniques based upon work done by Corbett [4], and the Graham-Haley-Joy [7]. We use a new panic mode algorithm, which Corbett derived from Sippu and Soisalon-Soininen [16], which depends on the feasibility of a recovery goal. In addition this method allows symbols to be inserted in order to allow a panic recovery. We will not concentrate on the error recovery aspects of the compiler, except where it impacts the back end.

The back end of the compiler is responsible for converting the internal form into HPS I code. The internal form consists of lists which control trees. Control structures are represented by lists. Expressions are represented by trees. A simplified diagram of these two structures is shown in figure 10. Features such as labels are ignored. Leaf nodes of expression trees are also more complex than shown. For example, fields such as links to symbol table entries, relative offsets, etc. are ignored for simplicity.

Figure 10a represents a control node. Its exact contents depends on the type field. For instance, the if statement in 'C' would use the Control Expr field to point to a Data node, such as the one shown in Figure 10b, the Cond Statement to point to another control node to be executed if the Control Expr field evaluates to true, and the Next Statement to point to the control node to be executed if the Control Expr field evaluates to false. Figure 10b represents a data node. It contains an operator which
functions are called or the `?` operator is used. These two cases imply a change in control. However, it is more convenient in the actual parsing phase to leave them as part of an expression.

3. Allocate and initialize all arrays. Generate control and expression nodes for both the prologue and epilogue of the function.

4. For each control node in a function, in sequence, starting at the head, scan the control expression and fill in the appropriate entries in the tables shown in figure 11. Note that table space is allocated as if the arrays were stacks. After the control expression is generated, scan the conditional statement list if (one exists) and generate its corresponding code. Note that this is recursive, but a temporary stack is used to mark what lists have been scanned or still need to be scanned.

5.2. Code Generation.

After creating the array based format, the next step is code generation. An estimate is made for the number of control instructions and the sizes of the DAGs. Space for the largest DAG is allocated. The control nodes are then generated. DAG addresses are labeled. In control nodes, only the labels are used as addresses. It is assumed that the linker will later resolve references to the DAGs. The DAGs themselves are then generated. This involves scanning the array format sequentially and entering actual DAG nodes into the DAG node buffer. If the size of the array format exceeds that allowed by the hardware (this is accounted for above when the scan is done), then the excess nodes are rolled over into the next DAG. A cont (continue) control node is used to merge the DAGs.

This process involves selecting algorithms for node scheduling, node selection, and node compression/expansion. Node scheduling is the task of selecting which nodes are presented to the target architecture first. Within a level, the longer running nodes are chosen over fast nodes. Node selection is a problem when there is more than one way to execute the same DAG. This is a direct result of ambiguity in the target ISP. The fast, node compression/expansion, occurs if the machine supports a node which performs more or less work than a node in the 'C' compiler. Expansion is required if a compiler's basic node is not represented in the target machine. Node compression is required if there exists a target machine node which can be used for a subgraph of the DAG.


As this paper goes to press, the compiler is undergoing final debugging. Although it is dangerous to predict performance benefits in the absence of a running system, our earlier work with HPS I (18) suggests that the compiler/machine will provide performance improvements over other machines made from the same technology. Our encouragement is based on the new architecture's ability to execute nodes out of order and the compiler's ability to not throw away the parallelism inherent in the source program. This allows an asynchronous model of execution, which should make its
speed compatible with the potential of data flow machines. However, the architecture visible to the programmer is the sequential Von Neumann model, so it does not require departure from prior programming practice.

A statement about our immediate future plans for this project is in order. After the compiler is completed, we will measure the effective parallelism it provides. We expect to obtain statistics on the average mix of the various nodes, which will help determine how many of each class of execution units are needed. Further, we hope to understand better the limits of parallelism inherent in 'C'. Finally, we plan to benchmark our compiler/computer pair against other models of execution.

Acknowledgement

This work was supported in part by the Defense Advanced Research Projects Agency (DOD), Arpa Order No. 4871, and monitored by Naval Electronic Systems Command under contract No. N0003984PCP00089. We also acknowledge the Advanced Development team of Digital Equipment Corporation's High Performance Systems/Clusters Group for providing first rate technical interaction on a lot of the tough problems, as well as funding for part of the project. Finally, we acknowledge the contributions of the other members of the HPS research team, Chien Chen and Jia-juin Wei.

References