INSTRUCTION PREFETCH STRATEGIES IN A PIPELINED PROCESSOR

by

Hubert Rae M^cLellan, Jr.

Submitted to
The Department Of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science,
January 13, 1983

Certified by [illegible]
Stephen A. Ward
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Departmental Committee on Theses

Instruction Prefetch Strategies in a Pipelined Processor

by

Hubert Rae M^{c}Lellan, Jr.

Submitted to
The Department Of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

Abstract

Instruction prefetching is an important aspect of contemporary high performance computer architectures. The C Machine, a pipelined processor currently under design at Bell Laboratories Computing Science Research Center, incorporates a microinstruction cache. This cache permits a fully autonomous prefetch unit to incorporate a variety of intelligent prefetch strategies. Measuring the performance of real programs run on an an architectural simulator enables us to evaluate the utility of branch prediction, intelligent prefetching, and instruction caching. Several prefetch procedures were analyzed in an attempt to quantify the efficacy of each method and identify the limiting architectural parameters. Experimental results showed that highly optimized prefetch strategies did not produce significant performance improvements.

Keywords: computer architecture, instruction prefetch, pipeline computers, hazard, cache, computer aided design

Thesis Supervisor: Stephen A. Ward
Title: Associate Professor of Computer Science

Acknowledgements

There are many people whose assistance on the C Machine design project contributed substantially to the completion of this thesis. S. C. Johnson constructed the compilers and participated in many discussions of the architectural design. The dynamic measurements were based on work done by P. J. Weinberger. The C Machine would not have been possible were it not for the enthusiastic research of Dave Ditzel, whose seminal design methodology was central to the architectural development. A. G. Fraser, G. L. Chesson, and R. F. Cmelik are thanked for their helpful comments and support.

I would like to thank my thesis advisor, Steve Ward, for his support, and giving me the freedom to conduct this research. I am especially grateful to my compatriots in the RTS group of the Laboratory for Computer Science for their friendship and support during work on this thesis. I would like to thank:

Bert Halstead for patiently assisting me in focusing the thesis direction.

Oded Feingold, who miraculously converted my draft into readable English.

Jon Sieber, for his helpful suggestions of important parameters to measure during C Machine simulations and his proofreading pass over the final report.

Jessica Hodgins, who proofread and assisted in preparation of the final copy.

Lisa Chabot, who bakes fantastic cinnamon bread.

Jeff Arnold, Dave Goddeau, Al Mok, Jon Powell, Kem Stewart, Chris Terman, Eva Tervo, Jack Test, and Rich Zippel all whom made the environment a stimulating and productive place work.

And finally, I'd like to thank my parents for their love and support throughout my academic career.

Table of Contents

Figures

§1 Introduction

Instruction prefetching is an important aspect of contemporary high performance computer architectures. The C Machine, a pipelined processor currently under design at Bell Laboratories Computing Science Research Center, incorporates a microinstruction cache. This cache permits many possible prefetch strategies. Measuring the performance of real programs run on an an architectural simulator enables us to evaluate the utility of branch prediction, intelligent prefetching, and instruction caching. In this manuscript, several prefetch procedures will be analyzed in an attempt to quantify the efficacy of each method and identify the limiting architectural parameters.

The first two sections are a presentation of background information. Section 1 contains the definition of terms required to describe the problems associated with pipeline processor design. A discussion of the effects of control transfer instructions follows and culminates with a description of the tradeoffs which govern realizable pipeline performance. Section 2 presents the two major segments of a processor, the IFU and the EU, and discusses their possible interconnection methods. The randomly addressable cache is introduced as a new IFU-EU interconnection technique.

The architecture of the C Machine is then presented as a paradigm for examining prefetch strategies. Section 4 describes the decisions surrounding the design of a streamlined instruction set. Aspects of the C Machine instruction set are presented as well as the design tradeoffs which affect its rapid decode. Section 5 describes the details of the C Machine IFU and how instructions are decoded. Demand instruction fetching is contrasted with instruction prefetching. Section 6 defines the two types of prefetch, blind prefetch and intelligent prefetch, and suggests possible intelligent prefetch strategies that might improve pipeline performance.

To understand the problems associated with an architectural design, empirical performance measurements are required. Section 7 describes the software tools written to meter performance and justify the design decisions. Section 8 contains statistical data gathered from the C Machine simulator analyzing the advantages of several actual prefetch strategies. The results of these measurements and their implications are then discussed. Section 9 summarizes the findings and draws some conclusions on the design process of the C Machine architecture.

§1.1 Pipelining

Pipelined processors partition instruction execution into separate stages[1,2] and the parallel execution of multiple pipeline stages on sequential instructions achieves a high execution rate. With pipelining, the performance is limited only by slowest stage's execution delay and the time to complete any single instruction is the pipe latency, or the sum of the delays for all the stages. As the computation is partitioned into smaller, simpler sections, the portion of the execution performed at each stage decreases. These simpler operations are performed faster and the maximum execution rate is increased.

Two constraints prevent the partitioning of a pipeline into ever finer stages. The first is inter-instruction data dependencies or *hazards*. The second and more stringent restriction is caused by the execution of *control transfer instructions*. Both introduce delays that interrupt the smooth flow of instructions in the pipeline and reduce performance. As the number of stages is increased, more instructions are concurrently executing in the pipeline and the detrimental effects of hazards and control transfer instructions degrade pipeline performance ever more severely.

In a synchronous pipeline, as shown in figure 1, instructions proceed uniformly from one stage to the next each clock cycle. Instruction i is followed by instruction $i+1$, $i+2$, ... If the data dependencies of instructions $i+1$, $i+2$, ..., require results which are not yet available from instruction i, a hazard[3] arises and the data dependent instructions are made to wait until the hazard is resolved. The more stages there are in the pipeline, the longer the delay may be before the hazard is resolved. This waiting prevents the pipeline from flowing at the maximum execution rate.

Three distinguishable points in a pipeline are relevant to control transfer instructions: the *head* of the pipe, the *transfer recognition point*, and the *control point*. The relative position of these three points determine how severely control transfer instructions affect pipeline performance.

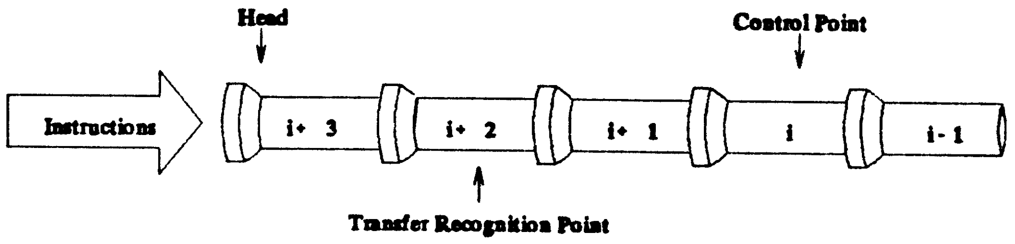

Figure 1: Pipeline Overview

The *head* is the first and most crucial stage in the pipe. To maximize the execution rate, the head must initiate a new instruction down the pipe every cycle. In a highly pipelined processor, the instruction may not even be decoded at this early stage. Therefore, the true successor instruction may be indeterminable. he best the head can do is assume the succeeding instruction will reside in the next sequential word of memory.

The *transfer recognition point* is the stage in which control transfer instructions are discovered. From the head to this stage in the pipe, the instruction stream is constrained to be purely sequential. At the transfer recognition point, a control transfer instruction has been sufficiently decoded to generate the target address of a non-sequential instruction. This is the earliest point in the processor pipeline where non-sequential instruction fetching can be specified.

The *control point*[4,5] specifies the stage in the pipe beyond which instructions are guaranteed to complete their execution, even in the event of an interrupt. Instructions in prior stages of the pipeline prior can be aborted at any time without changing the processor's state. The control point[†] is also the stage where results of instructions which affect the processor's condition code are first available.

Generally, contiguously stored instructions are executed sequentially; after executing the instruction at location *l*, the processor will execute the instruction at location $l + 1$. Control transfer instructions potentially interrupt the sequential nature of instructions flowing through the pipe by explicitly specifying the target address of the next instruction to be executed.

There are two classes of control transfer instructions: unconditional and conditional. Unconditional control transfer instructions always interrupt sequential instruction execution; conditional control transfer instructions may generate a nonsequential target address depending on the state of the processor.

When an unconditional control transfer instruction is detected at the transfer recognition point, the head is redirected to commence fetching instructions at the specified target address. Instructions in the pipe's initial stages, from the head to the transfer recognition point, are aborted or flushed. The number of computation cycles lost may be reduced by designing the transfer recognition point as close to the head of the pipeline as possible.

However, conditional transfer instructions are a more complicated issue. When a conditional transfer instruction reaches the transfer recognition point, the deciding condition may be not yet be known. If it is known and matches the transfer condition, the conditional control transfer will be followed exactly like an unconditional control transfer instruction. If the deciding condition is known but does not match the transfer condition, the transfer is not

[†]Holgate's late control point.

followed, preceding stages of the pipe continue to execute sequential instructions; zero pipe stages are flushed, and no cycles are lost. Only if the conditional transfer is followed will there be any lost cycles. When a conditional transfer instruction depends on the results of a preceding instruction which has not yet reached the control point, the deciding condition is indeterminate (unknown). Unfortunately, there may be many stages between the transfer recognition point and the control point. These intervening stages result in a *gulf of ignorance* between instructions entering the pipeline and those completing execution. This dependency is similar to the hazards already mentioned. However in this case, the hazard is in the control stream, rather than the data stream.

There are two strategies to follow upon encountering an indeterminate conditional transfer instruction. The simplest strategy would have the control transfer instruction wait until the instruction which specifies the deciding condition arrives at the control point and the deciding condition is determined. As the conditional control transfer is held at the transfer recognition point, a gap of inactive stages will develop in the pipeline. The maximum length of this processing gap is the number of stages between the transfer recognition point and the control point, or the gulf of ignorance. These stages are lost cycles.

A second, more productive strategy is to guess the outcome of the indeterminate condition. The predicted path is fetched and processing continues. The conditional transfer instruction propagates through the pipe with enough information to correct the transfer should the prediction be inaccurate. Once the condition is determined, if the condition was incorrectly predicted, subsequent instructions are aborted.

In summary, control transfer instructions degrade execution performance by introducing gaps in the flow of instructions reaching the control point. Unconditional control transfer instructions impose a known fixed gap of execution; the number of instruction cycles lost is equal to the number of stages between the head of the pipe and the transfer recognition point. But, the performance degradation caused by conditional transfer instructions is variable. If the control transfer is not followed, performance degradation is nearly zero -- the only cost is the single stage occupied by the control transfer instruction itself. The subsequent sequential instructions are not flushed and no cycles are lost. However, the worst case degradation caused by a conditional control transfer instruction can be quite severe. This worst case arises when a conditional control transfer instruction must reach the control point before the deciding condition is determined. If the predicted path was incorrect, the complete length of the pipeline from the head to the control point must be flushed.

Conditional transfer instructions are often a limiting factor in the execution rate of pipeline processors. In architectures where most instructions are executed in one clock cycle, pipe flushing caused by control transfer instructions is the major expense. Long pipes tend to be damaged more by conditional control transfers; the longer the gulf of ignorance, the more stages lost to pipe flushing. The pipe stages lost to handling control transfer instructions reduce the effective execution rate. As the number of stages in the pipeline increases, performance is degraded by the percentage of incorrectly predicted conditional transfers.

Pipeline considerations in a SISD[6] (Single Instruction Single Data) architecture introduce two conflicting goals. First, for maximum speed, the pipeline should have many stages. Each stage executes a simple segment of an instruction's complete computation. The execution delay of these individual steps is minimal and the peak execution rate of the architecture is maximized. Second, to minimize the performance degradation due to both data hazards and control transfer instructions, the pipeline should be kept as short as possible. These conflicting desires force an engineering tradeoff. Performance measurements indicate that a reasonable pipeline length is three or four stages. The performance potential of longer pipelines is more readily obtainable in SIMD[6] (Single Instruction Multiple Data) processors, which will not be discussed in this paper.

§1.2 IFU -- EU Interconnection

Most contemporary high performance architectures achieve parallelism by overlapping instruction fetch with instruction execution. While the processor is executing instruction *i*, instruction *i + 1* is fetched and possibly even decoded. The CPU hardware associated with retrieving instructions from main memory is the ***instruction fetch unit***, (IFU) and the remainder is the ***execution unit*** (EU). Since the operations performed in each of these units are distinct and separable, it is convenient to split the processor into two such sections. The overlapped processing of these two units is a form of pipelining. Often, both the IFU and the EU are internally pipelined. Such an architecture can be described as a hierarchical pipeline.

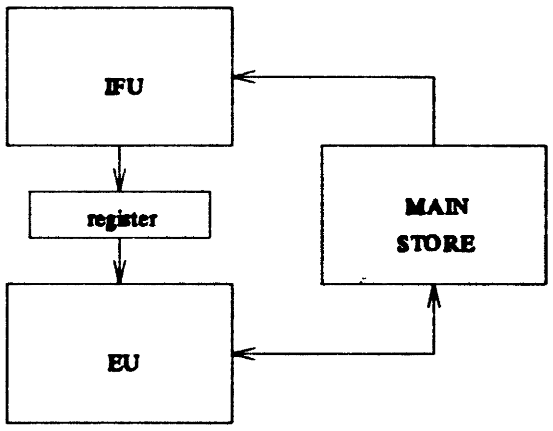

Figure 2: Processor Organization: IFU-EU Connected By Register

The organizational advantage of separating the processor functions is not without its associated penalties. Reconnecting these two units is not necessarily easy and the technique used affects the overall processor performance. The IFU and EU can be connected in several ways[2,7,8,9,10] The simplest method rigidly connects the two units with a single register[11]

However, if the EU is busy executing the instruction held in this register, the IFU activity must eventually come to a stop. Parallel activity is achieved only when both units are processing.

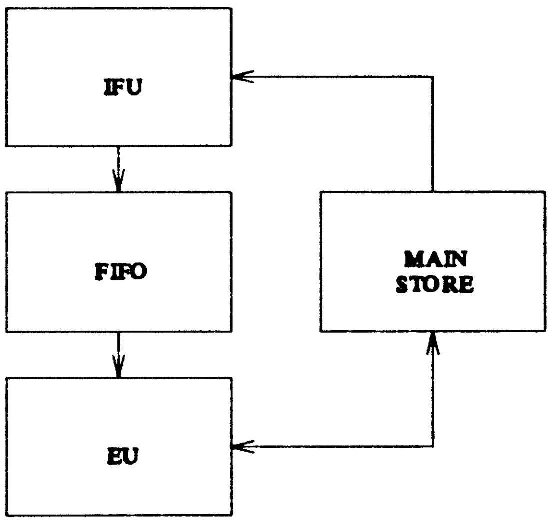

Figure 3: IFU-EU Connected By FIFO

A more commonly used interconnection replaces the single register with a first-in-first-out (FIFO) queue[12,13] The FIFO decouples the two units and allows each to proceed at a more or less independent rate. The instruction queue also smoothes out the extremes in processing times -- EU time for an add versus a multiply, or IFU delay due to control transfers -- so each unit is more fully utilized. The main problem with the FIFO interconnection is the linear nature of its contents. Since the IFU can only guess the correct path beyond execution-dependent control transfer instructions, the queue's contents are the IFU's prediction of the correct sequential instruction stream. If the EU specifies a different path, the queue is flushed and the IFU is redirected to follow the corrected instruction sequence.

A recently proposed interconnection method is a randomly addressable cache[14] One advantage of a cache over a queue is the ability to capture program loops. Ad hoc mechanisms for minimizing control transfer overhead, such as target instruction buffers[15,16] jump traces[17] and loop buffers[16,18] are eliminated. The cache interconnection provides these capabilities with improved performance. The IFU doesn't have to refill the instruction queue repeatedly and system memory traffic due to instruction fetch is reduced.

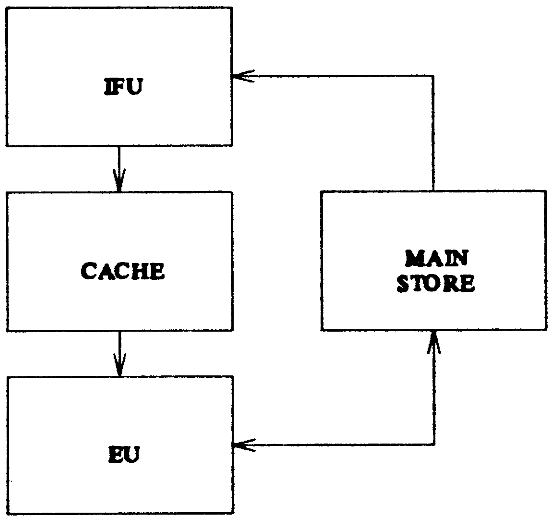

Figure 4: IFU-EU Connected By Cache

In comparison with registers and FIFOs, the cache is a more general IFU - EU interconnection technique, and more effectively decouples the IFU and EU. Since they are less tightly connected, each unit processes instructions unhindered by the hazards and pipeline flushing of the other. Hence the cache interconnection splits a long pipeline into two independent shorter segments.

2 C Machine Architecture

The current C Machine design effort is a continuation of previous work at Bell Laboratories Computing Science Research Center. In 1975, A. G. Fraser and D. M. Ritchie[19] suggested a C Machine instruction set oriented for a microprogrammed implementation on the HP21-MX. Fraser later (1978) constructed a prototype 16-bit C Machine that unfortunately was never made fully operational. Interest in 32-bit architectures grew as many C programs began feeling the limits of a 16-bit address space. S. C. Johnson initiated a design cycle of generating an instruction set and encoding format, writing a compiler for that machine, and then using performance measurements of newly compiled programs, to propose a new instruction set. This process yielded a 32-bit CPU instruction set with smaller static code size than the PDP-11's[20] In 1980, D. R. Ditzel[21] investigated an relatively simple machine architecture and implemented the register set and ALU in NMOS VLSI. S. R. Bourne designed a 32-bit C Machine to be implemented in TTL, but this machine was never constructed.

More recently, Ditzel and I have designed a high performance pipelined version of the C Machine. This SISD processor was intended to exhibit a significantly better cost/performance ratio than existing computers. Its design was heavily influenced by extensive performance measurements of C programs that guided hardware/software tradeoffs between compiler technology, common architectural techniques, and circuit technology limitations.

2.1 Instruction Set

It is often assumed that a good instruction set minimizes the *semantic gap*[22] between the concepts in high-level languages and their realization in computer architecture. Recent instruction set designs have attempted to reduce this gap by introducing complex functions and addressing modes. This trend is based on the belief that processing power increases as the operations performed by a computer's instruction set approach the semantics defined by the high-level language[23,24,25] There is current debate[26,27] on the validity of such an approach. There are several reasons why a simpler instruction set[28,29,30] can produce a higher performance architecture. Two examples are discussed below.

Complex, high-function instructions often execute significantly faster when implemented with dedicated hardware than when programmed as a sequence of primitive instructions. However, additional hardware is likely to spread existing logic further apart, increasing both propagation delays[31] and the basic machine cycle time. Since primitive instructions account for the majority of instructions executed, the performance improvement of complex instructions must be weighed against the reduction in speed of the primitive instructions. This comparison must also take into account the relative frequency with which proposed complex instructions are executed. Because a simpler instruction set reduces the basic machine cycle time, any "improvements" must justify themselves by decreasing the aggregate timing of actual computations.

Compilers may have a difficult time producing code for complex instruction sets if they attempt to match the operations specified in the high level language program to the functions provided by the machine's instruction set. Finding a good match with complex instruction sets may be an arduous task – in the DEC VAX11/780, which has a very rich set of instructions and addressing modes, there approximately two dozen ways to perform an addition. Since there are so many possible mappings, often the one selected is suboptimal and generates additional, gratuitous processing.

An instruction set's power is determined by the amount of work performed by each instruction as well as the instruction execution rate. It is not the time required to complete an individual instruction, but rather the total time to complete a computation that is important. An architecture which performs many impotent instructions in a short time is no more powerful than a machine which executes a single high level "solve-it" instruction in a very long time. Since instruction potency and execution rate typically don't have an exact inverse relationship, a reasonable balance between aggregate computational power and execution delay is an important aspect of a good instruction set.

The C Machine achieves a balanced instruction set by designing the simple, primitive capabilities of the hardware to match the frequently required operations of the high level language. For example, the C Machine has a full 32-bit architecture: Byte addresses are 32-bits long, arithmetic is 32-bit two's complement, and logic functions are 32-bit operations. The C Machine is a two address, memory to memory architecture, so registers are noticeably absent. Instead, several varieties of caches[32] are used to speed data access. The instruction set supports only a two-address form of dyadic operations since the extra power of three-address instructions was insufficient when compared to the increased hardware complexity required to support them. Each binary operation contains two full 32-bit operand specifiers. The *signed* and *unsigned* C data types, *char*, *short*, and *long*, are fully supported.

As closely as possible, the addressing modes reflect the high level language storage class referencing methods. The C Machine supports a minimal set[26] of four addressing modes:

- Immediate constant
- Absolute address
- Stack pointer relative
- Stack pointer relative indirect

The absolute addressing mode is used to reach static or global variables whose address is known at compile time. Stack pointer relative mode is used to access local variables in the stack frame. Complex addressing modes were intentionally avoided to smooth the flow of pipelined instructions as this reduced set of addressing modes simplifies the hardware associated with effective address generation.

The incidence of inter-instruction data hazards can be minimized by proper instruction set design. For example, *push* and *pop* operations, which modify the stack pointer by side effect, create hazards for subsequent instructions with stack-pointer-relative operands. With a modification of the calling sequence and stack frame conventions[32] these operations can be converted to fairly innocuous *move* instructions. The stack pointer (SP) is only modified when necessary to allocate or deallocate storage on procedure entry and return. A relatively constant SP smoothes the flow of pipelined instructions.

Another important consideration in a pipelined instruction set is how to deal with condition codes, which provide an implicit communication mechanism between two otherwise disjoint instructions. Two problems associated with their use are aggravated by pipelined architecture: First, not all instructions update the condition codes and this irregular[33] use complicates the condition code controlling hardware. Second, since condition codes may be updated by side-effect, predicting their value in a pipelined system is extremely difficult. Typically, conditional control transfer instructions modify the direction of the instruction stream depending on the condition code's value.

Several pipelined architectures[11,34,35,36] have eliminated condition codes completely by use of an atomic "compare and branch" instruction. Initial evaluation of such an instruction would indicate little difficulty -- two apparently orthogonal, frequently paired operations have been combined into a single instruction. However, since the branch part is executed at the transfer recognition point, and the compare part of the instruction is executed possibly many stages later at the control point, the *full* gulf of ignorance must pass before the validity of the branch prediction is determined.

The use of separate instructions for conditional control transfer and condition code setting[37,38] allows the interposition of other instructions which do not affect the condition code's state. If enough of these intervening instructions are present, the condition code setting instruction will reach the control point before the conditional control transfer instruction arrives at the transfer recognition point. Then the condition codes will already be set, the conditional control transfer is immediately determinable, and there is no extra pipe flushing penalty associated with the gulf of ignorance. All of the advantages and none of the drawbacks of combined "compare and branch" instructions are available via *branch folding*. The C Machine instruction set defines a single True/False condition code flag, thereby reducing the difficulties encountered in a pipelined environment. In addition, only a single instruction, *compare*, may modify the condition flag. Restricting the condition code operations to a few instructions minimizes the control hardware and permits the smooth flow of pipelined instructions.

Once the pipeline considerations of the instruction set semantics have been settled, the issues of instruction encoding arise. Instruction encodings can be either fixed or variable in length. In a fixed size instruction set, all instructions are encoded in the same number of bits. A variable length instruction set defines operations in several instruction sizes. A particular instruction's size depends on how many bits are required to specify its operations and how frequently that instruction appears. Uncommon instructions need not be densely encoded, since the code space penalty associated with their larger size is mitigated by their infrequent occurrence.

A fixed size instruction set is better suited for a pipelined architecture, since it facilitates the early decode of control transfer instructions. The transfer recognition point may be closer to the head of the pipe and therefore the number of pipestages flushed due to followed control transfer instructions is reduced. A fixed size instruction set simplifies decoding and generally allows a more continuous flow of pipelined instructions. On the other hand, variable length instruction sets permit a more compact encoding. A program can be described in fewer bits, and the instruction fetching bandwidth required to support the execution of such an instruction set is reduced. However, unlike the smooth flow of a fixed size instruction set, the bandwidth requirements of a variable length instruction encoding is sporadic and execution gaps may form in the pipe.

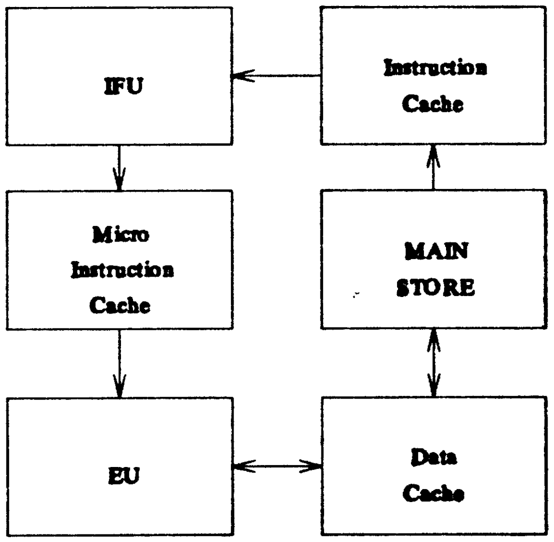

Figure 5: C Machine Block Diagram

By means of the microinstruction cache, the C Machine supports both a fixed and a variable length instruction set. The microinstruction cache stores fully expanded, decoded instructions, permitting the performance advantages of a fixed length instruction set on the EU side of the microinstruction cache. In fact, the microinstruction cache can be viewed as a most recently used set of horizontal microinstructions. But the length of these *canonical* instructions is prohibitive for describing entire programs. Hence, it is the IFU's function to translate the space-saving variable length encoded macro instruction format to the wide, readily executable canonical microinstruction cache entries.

Given a variable length instruction set, code size is partially a function of the minimum "granule" size. In the C machine design process, 8-bit and 16-bit resolution encoding formats were investigated, and the 16-bit granularity was found superior. For example, PC-relative offsets need not specify a byte boundary, hence in control transfer instructions a 16-bit granularity allowed a fixed-size bit field to specify a wider range of targets. Also, 16-bit resolution required fewer instruction formats, reducing the overhead associated with format disambiguation and improving code density. And with simpler formats, the amount of IFU decode hardware is decreased.

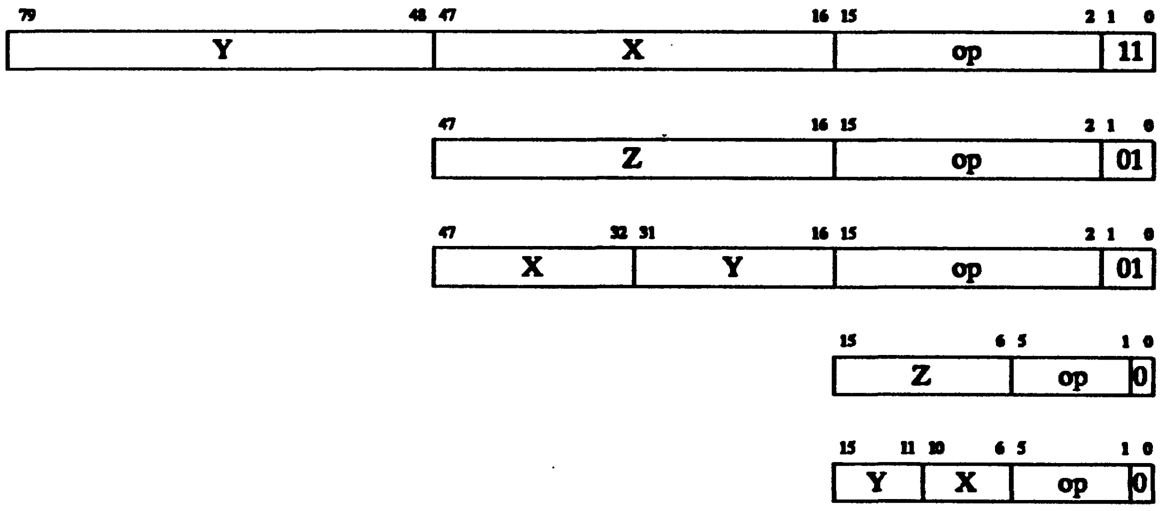

Figure 6. C Machine Instruction Formats.

The C Machine instruction set is composed of 16-bit *parcels*. Instructions are defined in either one-, three-, or five-parcel formats. Each instruction is translated by the IFU into canonical microinstructions. At the maximum instruction size, a five-parcel instruction encoding is necessary to specify fully the two operands of a binary operation. The operation and operand addressing modes are defined by the first parcel. The next two parcels provide 32 bits to specify the source operand and the final two parcels provide 32 bits to specify the destination operand.

Five-parcel instructions are needed to specify the most general form of a two address binary operation. However, it is often the case that both operand specifiers can be encoded in fewer than 32 bits. So all five-parcel dyadic instructions have three-parcel analogues, wherein each operand must be expressible in 16 bits. In fact, a final optimization has been made to compress the encoding yet further; statically measured instruction counts were analyzed to determine the most frequently used operation/operand combinations. The 32 most common pairs whose dyadic operands could be specified in only 5 bits or whose monadic operand could be specified in 10 bits were redundantly defined in the compact one-parcel form.

One of the major difficulties with a variable length instruction set is determining the address of the next contiguous instruction. Also, to facilitate rapid parallel decoding of a variable length instruction set, the format and length information must be readily discernible. Some architectures[39] compound the problem by allowing not only instructions to be of variable size, but also each operand. The C machine instruction set provides all this information in the first parcel of each instruction. With the format information in a single, fixed place, the contiguous successor instruction is fetched immediately, and the IFU can fetch and decode a new instruction every clock cycle.

The C Machine instruction set design was guided by measurements[40] of dynamic instruction frequencies. The high percentage of control transfer instructions was the most disturbing. They account for approximately one quarter of all instructions executed! To minimize the extent to which these instructions slow down execution, a static prediction bit is included in the encoding of conditional control transfer instructions to assist the IFU in guessing the correct instruction path. This bit is a static measure, generated at compile time, predicting which path should be taken following a conditional control transfer instruction. Use of the static prediction bit as a prefetch heuristic will be discussed in a subsequent section.

In summary, the performance limiting effects of inter-instruction data hazards and control transfer instructions can be reduced by careful design of the pipelined processor's instruction set. Use of complex addressing modes, especially those with side-effects, should be discouraged. Instructions are required to flow through the pipeline with minimal hindrance since maximum performance can be obtained only when there are no interruptions of the smooth flow of completed instructions exiting the processor pipeline. The combination of variable length macroinstructions with fixed length microinstructions permits high code density and rapid decode in performing the computations required by high level languages.

§2.2 Instruction Fetch Unit

The C Machine IFU fetches and decodes macroinstructions from the instruction cache, effectively translating the densely encoded macroinstructions into very wide horizontal microinstructions. After each macroinstruction has passed through the IFU's three stages, it has been expanded to the canonical microinstruction format. These microinstructions are placed in the microinstruction cache. There is a one to one correspondence between macroinstructions and microinstructions. Whereas the macroinstructions are compactly encoded, the microinstructions are readily executed by the EU.

The ability to decode instructions in rapid succession, one per clock cycle, is an important design goal of the C Machine IFU. The IFU's processing bandwidth must match the EU instruction bandwidth for optimal results. During peak demand, the EU is capable of executing one instruction per clock cycle. Peak performance is attained by an equivalent IFU processing rate.

The IFU's efforts are directed towards reducing microinstruction cache misses. Such cache misses can produce considerable delays while the errant instruction is fetched and decoded. With proper prefetch strategies, the IFU can maintain the appropriate working set of instructions in the microinstruction cache.

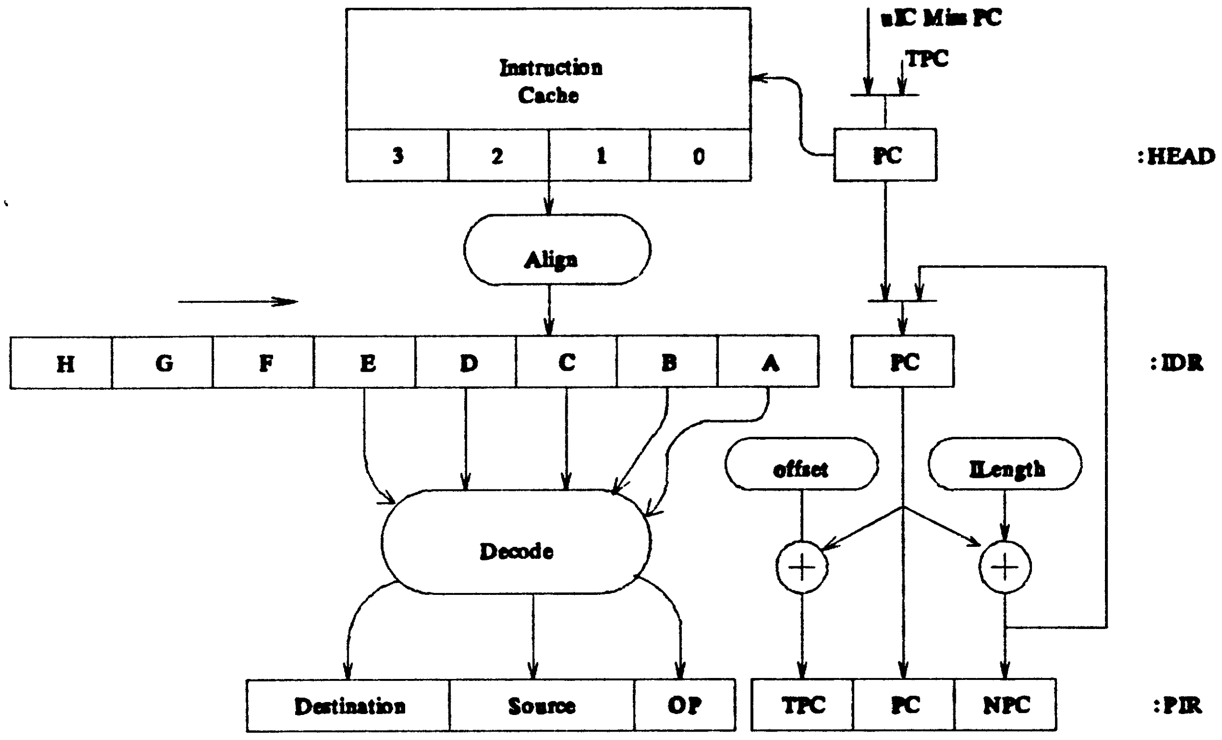

Figure 7: Three Stages of the C Machine IFU

Instruction fetching begins at the IFU's first stage, or head of the pipe. At this early stage, the instructions are nothing more than an uninterpreted bit stream. The head stage *program counter*, or PC, addresses a 64-bit quad parcel in the instruction cache. Since it is too early to determine instruction boundaries, during each clock cycle the head stage merely transfers as many of the four parcels as possible to the *instruction decode register*, the IFU's second stage. When the last parcel is transferred, the head's PC is incremented and the next sequential quad parcel is retrieved from the instruction cache.

The instruction decode register (IDR) can be considered as an 8 parcel shift register. As instructions are decoded, parcel are shifted from locations H to A (figure 7). Several operations are performed on the IDR every clock cycle. At the end of each cycle an instruction, which is either one, three or five parcels long, is decoded in parallel and removed from the IDR. The remaining valid unused IDR parcels are shifted into position and any free slots (up to four) are filled from the next quad parcel delivered by the head of the pipe. The IDR PC is incremented by the amount appropriate to reflect the next contiguous instruction's address.

Since the instruction length controls the number of parcel locations the IDR contents are shifted, it must be easily decoded Additionally, when fetching sequential instructions, the next IDR PC is determined by adding the instruction length to the current value. Distributing the length information throughout the instruction, or requiring serial decode of each operand by poor instruction set design [39] can complicate the decode hardware and introduce unacceptable delays. In the C Machine the first parcel of every instruction contains all the information required to control subsequent decode operations. Signals derived from the IDR's parcel A determine the formation of each canonical instruction.

Canonical instructions are prepared in the *prefetch instruction register* (PIR.) The operand specifiers, as indicated by the instruction format, are expanded to a full 32 bits and inserted in the PIR source and destination fields. The PIR *next program counter* (NPC) indicates the address of the next contiguous instruction. The PIR *target program counter* (TPC) is used if the decoded instruction is a control transfer instruction. It is generated by adding the selected PC-relative offset to the IDR's PC.

Both the instruction and the microinstruction caches are direct map caches of the instruction virtual address space. The microinstruction cache stores one instruction per entry, whereas the instruction cache stores one quad parcel per entry. The fixed sized microinstruction cache entries are designed to facilitate instruction retrieval in the EU. Each canonical microinstruction the IFU produces is written in the microinstruction cache entry selected by the PIR PC. Because of this addressing scheme, and the variable length nature of the instruction set, the microinstruction cache is sparsely filled. Conversely, the instruction cache is densely filled.

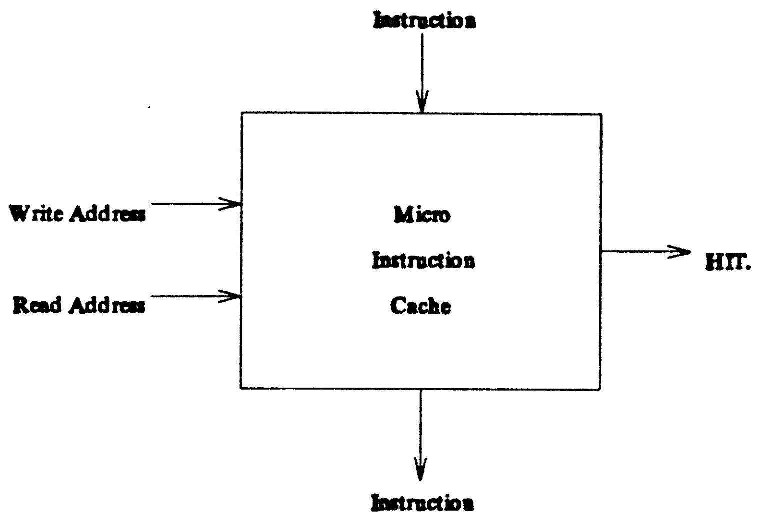

Figure 8: Microinstruction Cache Schematic

The TPC and NPC in the canonical instruction are full 32-bit addresses. Therefore, the EU can follow control transfers without the penalty of lost stages or branch delay. When the EU executes an instruction, the following instruction is specified as soon as possible. Depending whether the next instruction will be sequential or the (nonsequential) target of a control transfer instruction, either the NPC or TPC entry of the current instruction is used to index into the microinstruction cache. There is no extra delay for altering the sequential instruction flow, since the mechanism is the same. Sequential instruction execution in the EU is not impeded by control transfer instructions while the targets are resident in the microinstruction cache.

There are two possible consequences when the EU requests a non-resident instruction from the microinstruction cache. If the missing instruction is already proceeding through the IFU, the EU idles until it is completely decoded. Otherwise, if the instruction is not in the IFU pipeline, the IFU is directed to fetch and decode the offending instruction. This is known as *demand fetching*.

Demand fetching is the main communication link between the EU and IFU. During a demand fetch the IDR is cleared of all parcels in preparation for receiving the new quad parcel from the head stage. The head PC is initialized to the instruction address of the microinstruction cache miss. It then used to retrieve the 64-bit quad parcel which contains the missing instruction's initial parcel. During the following clock cycle, the appropriate parts of this quad parcel are aligned and placed in the IDR. The low order two bits of the head PC determine the alignment of the quad parcel in the IDR. As the parcels are loaded, the head PC's contents are copied to the IDR PC. The IDR now holds the address and at least the initial parcel of the desired instruction. The remaining decode operations are performed and the instruction is sent on to the EU.

The EU will have to wait a variable number of cycles for the *repair* of a microinstruction cache miss. It waits from one to three cycles if the instruction is already in the IFU pipeline. Otherwise, if the instruction cache contains the necessary quad parcel, the EU waits either four or five cycles depending on whether the instruction crosses a quad parcel boundary. Finally, the worst case delay following a microinstruction cache miss occurs when the instruction cache misses too. If the instruction does not reside in the instruction cache, it must be fetched from main store, whose access latency may be many cycles. In order to avoid these delays, it is important to maintain a high microinstruction cache hit rate. Processing performance is severely degraded by a poor hit rate.

The IFU's transfer recognition point resides in the IDR. At this stage, control transfer instructions are sufficiently decoded to affect the fetching of subsequent instructions. Unconditional control transfer instructions leaving the IDR may redirect the sequence of instructions entering the head of the pipe – a followed control transfer instruction is handled like a demand fetch. The remaining IDR parcels are flushed, the TPC is passed to the head, and the new instruction stream is fetched. The specific choice of instruction path following a conditional control transfer instruction depends on the details of the prefetch strategy.

§2.3 Branch Folding

Branch folding was serendipitously discovered while designing the IFU and results in considerable performance improvement in the EU. With branch folding, the machine cycle spent executing a control transfer instruction in the EU is eliminated entirely. While not all control transfer instructions are "foldable", the majority may be combined with previous *execution class* instructions and processed as a single step in the EU.

Execution class instructions are those which do not use the branch control fields of the canonical microinstructions. Examples are *add*, *subtract*, *multiply*, and *compare*. After executing any of these instructions, the next instruction executed is, by default, the next contiguous one. The microinstruction NPC field contains the address of the next sequential instruction, and since the IFU is a three stage pipeline, it may contain several instructions in various stages of decode. Often an execution class instruction is held in the last stage of the IFU, (the PIR), while a control transfer instruction is in the IDR. Before the PIR's contents are written into the microinstruction cache, the branch control fields are modified to produce the effects of the logically subsequent control transfer. In this way, the two macroinstructions are combined in a single microinstruction entry. Branch folding provides up to a 30% performance improvement by eliminating separate control transfer instructions in the EU.

§3 Prefetch Strategies

Instruction prefetch contributes considerably to overall processor performance. An effective execution rate depends on the IFU being able to supply the EU an uninterrupted stream of instructions. When not servicing a demand instruction fetch, the IFU is *prefetching*. The prefetching IFU attempts to process instructions without an explicit EU request. While prefetching, the IFU generates the instruction stream by autonomously predicting which instruction will be executed next. Ideally, the IFU would provide the EU decoded instructions coincident with their need. Unfortunately, since the IFU cannot infallibly predict the EU's instruction requirements, an optimal prefetching strategy is non-causal. All realizable prefetch strategies are engineering approximations of this unimplementable ideal.

There are two instruction prefetch strategies of interest. The first and simpler one is *blind* prefetch where there is no interpretation of the items to be cached. Blind prefetch has long been used to enhance cache[41,42,43,44,45] performance. When cache line l is referenced, cache line $l + 1$ is also retrieved. Instruction cache prefetching[46] is particularly effective because of the highly sequential nature of instruction access patterns. The simple multiword fetch of the PDP11/70 cache[47] is an example of blind instruction prefetch. The C Machine instruction cache hardware implements blind prefetch to reduce the number of misses and thereby the effective main memory latency seen by the IFU requests.

Blind prefetch also describes instruction fetching in the early stages of the C Machine IFU. The sequential flow of instructions in the IFU from the head stage to the IDR is constrained to be contiguous parcels. Control information which would modify the instruction stream is not available until the parcels have reached the IDR, or transfer recognition point. Since instructions in the head stage are uninterpreted, blind prefetch or sequential fetching from contiguous locations is the only possible option.

The C Machine microinstruction cache effectively separates the IFU from the EU and provides the possibility of a more informed prefetch. While the EU is processing a loop resident in the microinstruction cache, the IFU is free to prefetch instructions beyond the iteration. Most architectures are only capable of blind prefetch, but the C machine, with the aid of the the microinstruction cache, permits intelligent prefetch as well as blind prefetch. The microinstruction cache permits the complete power of the IFU to be directed to intelligent instruction prefetching. When instructions reach the transfer recognition point in the IFU, they can affect the future instruction stream. If a control transfer instruction is decoded, the IFU pipeline can be redirected to fetch the target path.

But the IFU cannot follow all control transfer instructions. The ability to specify a dynamically calculated target is provided by indirect control transfer instructions of which the most frequently occurring example is the subroutine *return* instruction. Indirect control transfer instructions specify the memory location where the address of the next instruction is stored. Because the microinstruction cache decouples the IFU and EU, resolving the data hazard on this location is impossible. Hence, indirect control transfer instructions are processed by the EU. Prefetching ceases when an indirect control transfer instruction is decoded. The IFU must wait for the EU to execute it and provide the address of the next instruction. This is unfortunate because statistically, approximately 2.5% of all instructions executed are *returns*.

Upon decoding, the prefetching IFU immediately follows unconditional control transfer instructions. The difficulty arises in the inherent delay of pipeline flushing associated with specifying a new instruction stream. Because of this penalty, some designs[48,49,50,51] have ignored entirely prefetching the targets of conditional control transfer instructions.

Branch prediction techniques[52] can aid the IFU in dealing with conditional control transfers. The static prediction bit represents the most expeditious path to follow. It must be kept in mind that the real purpose of this prediction is to speed the flow of instructions in the EU and a dynamic copy is maintained in the microinstruction cache. This bit is initialized to the value of the static prediction bit each time the associated control transfer instruction is decoded and loaded into the microinstruction cache. Subsequent execution will update the dynamic bit with the sense of the path just taken, maintaining a record of the most recent choice.

To offset the penalty of following branches, the IFU can temper its choice by checking the transfer distance. If the target is only a few instructions *beyond* the current one, it may be more profitable for the IFU to ignore the control transfer and continue decoding sequential instructions. The cycles that would have been lost in following the conditional control transfer are used to decode the intervening instructions. Otherwise, if the target is only a few instructions *previous* to the current one, it can be reasonably assumed that the IFU has already decoded them. If the specified target is within a threshold distance from the current instruction, the IFU may disregard the branch prediction.

Rapid instruction decode is only one aspect of the IFU's prefetching heuristic. The IFU also attempts to maintain a high microinstruction cache hit rate for the EU. Since the microinstruction cache is direct mapped, new instructions issued by the IFU will write over older ones. As in all cache designs, a pathological accessing pattern can ruin any performance improvements. Uncontrolled prefetching might actually hinder overall performance.

To minimize microinstruction cache conflicts, the IFU's headlong prefetching activity should be controlled. After a demand fetch, the IFU can be limited to prefetch a fixed number of instructions, then enter an idle state. IFU activity is reinstated only after the EU again demands an instruction fetch. This technique also limits the increased main store traffic that prefetching causes, even though most of the this traffic is masked by the instruction cache.

The problems associated with prefetching straight-line code are different than those encountered in prefetching iteration code. As long as the new instructions do not conflict with older ones in the microinstruction cache, straight-line, sequential instructions are processed as rapidly as the IFU can decode them. More control is required, however, in handling iteration code.

For example, consider the two possible meta-assembly language templates for the **for** statement, a C iteration construct. In the following figures, the time sequence indicates the order in which the meta-assembly language lines will be prefetched. Each time the IFU pipeline is broken, the number is incremented. Each template breaks the IFU pipe twice. Both code segments are entered from the top.

```
                                                 Time →

      < initialization code >                    0
L1:   < test termination condition >              0   1
      < conditional branch to L2 >                 0   2
      < body of loop >                              0
      < unconditional branch to L1 >                 1
L2:                                                     2
```

Figure 9a: Loop Template Optimized For Prefetch

```
      < initialization code >                    0
      < unconditional branch to L2 >              1
L1:   < body of loop >                               2
L2:   < test termination condition >              1  2
      < conditional branch to L1 >                 2  2
                                                      2
```

Figure 9b. Loop Template Optimized for Execution

In each case, if the prefetch heuristic follows the conditional control transfer instruction's branch prediction, the IFU will continuously decode the loop cycle. To prevent this, the microinstruction cache can be interrogated each time a instruction is loaded to see if it is already resident. When a duplicate instruction is found, the IFU will enter an idle state and wait for an EU demand fetch. This avoids unnecessary main store requests.

Higher performance is possible if the IFU can continue beyond the iteration code. While the EU is executing the loop, the IFU can prefetch the subsequent instructions. As each conditional control transfer instruction is decoded, the appropriate microinstruction cache entry is checked. In contrast with the previous strategy, when the instruction is already resident, the IFU follows the path opposite that suggested by the static branch prediction bit. Hence on first encounter with a conditional control transfer instruction, the most likely path is followed. On subsequent decodings, recognized by their residency in the microinstruction cache, the alternate path is followed. If this strategy is adhered to, the bodies of both templates are decoded only once.

The first template is more suitable for prefetching instructions. Since the IFU pipeline is broken only after the instructions in the bulk of the loop are decoded and placed in the microinstruction cache, the EU can be looping while the the IFU is following the control transfers. The second template is more suitable for execution in the EU. The unconditional control transfer is executed only once, whereas in the first template, it is executed each time through the loop. Therefore, the second template may conceivably execute fewer instructions. Also, the second template lends itself more readily to code motion. If the <*body of loop* > contains instructions which do not affect the determination of the loop exit condition, they can be interposed between the <*test termination condition* > and the <*conditional branch to* L1 >. This separation provides for a determinate deciding condition when the conditional control transfer instruction is issued in the EU. Such code motion is not possible in the first template.

§4 Architectural Design Tools

Over the past two years, several computer programs have been developed to guide design of the C Machine's architecture. These software tools were written in C and run on VAX11/750's under the Unix† operating system. Iterative performance measurements of the interactions between the C compiler, instruction set, and processor directed the architectural design. Trade-offs were made in each of these domains in order to produce a higher performance system architecture.

The *Portable C Compiler*[53,54] was modified to produce C Machine assembly language. There are several interesting aspects of the architecture which affect the compiler: First, the C machine has a memory-to-memory architecture and thereby obviates the difficulties of register allocation. Second, since iteration constructs were expected to loop, the compiler itself could generate the static branch prediction bit for some conditional control transfer instructions. Finally, a novel calling convention and statically sized stackframe[32] were used to reduce pipeline hazards.

Usually an assembler produces object code for the target machine. Since the object code's format changed as different instructions and addressing modes were tested and various encoding formats analyzed, the C Machine assembler generated a relatively constant intermediate instruction format. Each instruction in the intermediate format was represented as a fixed size, 16 byte horizontal C structure. Although these structures are large, they do not require decoding and are readily manipulated by programs.

†Unix is a trademark of Bell Laboratories

An interpreter was written to execute these intermediate instructions. Its chief goal was to provide statistics for an objective basis on which to judge the various proposed instruction sets. With the intermediate instruction format, the interpreter could execute approximately 1500 instructions per second. This rapid interpretation rate allowed statistics from a substantial body of programs to be gathered and analyzed. Several iterations of the instruction set, compiler, interpreter design cycle produced a fairly powerful instruction set well matched with current compiler capabilities.

While the interpreter cannot provide a measure of the internal interactions of the architecture, it is very useful for producing branch statistics. These statistics can be used to generate the static prediction bit in conditional control transfer instructions. This method is equivalent to the best the compiler could do to describe statically the most probable instruction stream.

The final object code is generated by the the instruction formatter, which produces the optimized one- and three-parcel macroinstructions for those intermediate instructions which meet the proper constraints. This program translates the intermediate format to the final encoding interpreted by the C Machine simulator.

An architectural simulator for the C Machine has been written to measure the effectiveness of various engineering design tradeoffs. This simulator has allowed the architecture to be tested long before any hardware is built. With a system simulator, the design process is much more objective. Performance variations introduced by architectural modifications can be empirically measured by simulating the execution of benchmark programs, and analysis of these collected statistics provide concrete justification for architectural decisions. This design process has promoted a simpler architecture. "Creeping featurism" has been avoided since all additional hardware had to be justified by a measurable improvement in system performance.

The simulator emulates the pipeline architecture down to the stage level. It is an interactive program which graphically displays the state of each pipeline stage. The simulator may be stepped on a cycle by cycle basis, so the detailed flow of pipeline instructions is dynamically analyzable. Each pipeline register may be examined and its contents modified. Breakpoints are provided to let the simulated program be halted and studied after a particular instruction is fetched. The simulator consists of approximately 5000 lines of C code and can execute about 200 basic machine clock cycles per second.

Although the detailed workings of the IFU and EU will not be described in this report, their internal pipeline delays (caused by inter-stage data dependencies, hazards, and bypassing) have all been carefully simulated. The aggregate effect of these pipeline irregularities during the execution of typical sequences of compiled C code, is sufficient information for evaluating different prefetch strategies.

An architectural simulator is useful in studying the efficacy of possible prefetch strategies. To describe such strategies, only a high level description of instruction flow is necessary. As shown previously in the top-level view of the architecture (figure 5), the EU retrieves and processes instructions from the microinstruction cache. If a miss occurs, the EU directs the IFU to fetch the missing instruction. The IFU typically requests the appropriate memory words from the instruction cache, decodes them and places the required instruction in the microinstruction cache. Since the IFU is a three stage pipeline, this process takes three cycles.

Instruction and data caches are standard architectural constructs used to reduce the memory access latency. Due to the sequential nature of instruction requests, the instruction cache need be no more than one-way set associative. As a rudimentary form of blind prefetch, four entries are retrieved from the main store each time the instruction cache misses. In the simulator, the main store is modeled as providing a fixed latency of six cycles after a request is made. With the measured hit rates, the instruction cache serves well to reduce the apparent main store latency.

A benchmark is needed that is representative[55] of typical C programs. The benchmark should be fairly flattened and not *inner loop bound* so that repeated executions of a small piece of code do not dominate the runtime statistics. The extensive use of caches in the C Machine's architecture requires the benchmark to be a fairly large program, otherwise the working sets of repeated loops and short programs might be entirely cache resident. Then, the statistics would no longer be representative of the complete system, but merely reflect the raw speed of the cache memories. Also the effectiveness of various prefetch strategies can only be measured with a large benchmark program.

The Portable C Compiler (PCC) is a good benchmark program because it is a large program representative of many existing C programs. Besides that, it has been widely distributed and extensively profiled. Also, it produces a tangible and testable output. Another one of its benefits is the small number of system calls required: the system interface requires only the *fopen, getc, putc,* and *fprintf* system calls. Since the simulator only measures the effects of user level programs, system functions must be handled by explicit simulator code.

The final measure of performance is the total number of cycles required to execute the benchmark. Another important statistic, signifying the power of the instruction set, is the total number of instructions executed. The ratio of these two measures, or cycles per instruction, indicates how smoothly the instructions flow through the pipeline. One instruction per cycle would be ideal. Any larger ratio indicates pipeline limitations, such as hazards, cache misses, and the effects of control-transfer-instigated pipe flushing.

Other statistics provide useful insight in recognizing performance limitations. An example is the microinstruction cache hit rate. Many cycles can be wasted waiting for a cache miss repair. The distribution of such idle cycles also conveys significant information. The prefetch algorithm's effectiveness can be diagnosed by analyzing the microinstruction cache's miss-repair idle cycle distribution.

Since the data cache competes with the IFU for access to the main store, uncontrolled instruction prefetch may also increase main store data requests to the detriment of processor performance. While the main store is processing IFU requests it can not service requests emanating from the data cache, and while the data cache is waiting for main store, the EU will be idle. This idle time increases the number of cycles per instruction and reduces overall performance. Because of this competition for main store, prefetch strategies can be ranked by the memory traffic they generate. Strategies which consume less of the main store's bandwidth are preferred.

§5 Measurements

In order to evaluate the comparative impacts of these various prefetch strategies, empirical measurements were gathered.

Seven different prefetch strategies were defined and implemented in the C Machine architectural simulator program. The other details of the simulation, such as cache sizes, memory latencies, and the benchmark program were held constant. For purposes of analyzing prefetch strategies, the instruction cache was set at 256 quad parcel entries, and the microinstruction cache size was also set to 256 entries. These cache sizes were specifically chosen to exaggerate the differences among the prefetch strategies' performance statistics. As mentioned previously, the benchmark used was PCC compiling a typical 100 line C program. This program running on the C Machine simulator produces approximately 5000 bytes of assembly code in about three hours. To verify the accuracy of this simulation, the output was compared with the assembly language produced by PCC running directly on the VAX11/750.

The prefetch strategies tested were:

TEST 1, No microinstruction cache: In order to judge the efficacy of instruction prefetching, base measurement needed to be established. This control was made by removing the microinstruction cache and combining the PIR, or final stage in the IFU, with the head of the EU. The PIR acted as the IFU-EU connecting register. The total pipeline became one stage shorter, but the advantages of the microinstruction cache were lost. Without the microinstruction cache, intelligent instruction prefetching was no longer possible. All other tested strategies included the microinstruction cache.

TEST 2, Conditionals ignored: The IFU was directed to ignore conditional control transfer instructions and continue fetching sequentially. In this strategy, the IFU ignored the branch prediction bit, while the EU continued to use it. Since only unconditional control transfer instructions were followed, pipe flushing in the IFU was reduced. This particular prefetch strategy did not perform well with the previously described meta-assembly language iteration templates. The results of this test were compared with other strategies to show the utility of following conditional control transfers.

TEST 3, Static prediction: The IFU followed conditional as well as unconditional, control transfer instructions. The prefetch path following a conditional control transfer instruction was selected based on the value of the static branch prediction bit.

TEST 4, Branch threshold: This test was the same as test 3 except that conditional control transfer instructions closer than a minimum threshold distance were ignored. Pipe flushing in the IFU was reduced and closer targets entered the microinstruction cache in other ways. The measured thresholds were 16, 32, and 64 bytes. These thresholds translate to roughly 4, 8, and 16 instructions since on the average, instructions are encountered every four bytes.

TEST 5, Limited prefetch: This test was also the same as test 3 except that the IFU was forced to enter an idle state after decoding a specified number of instructions. These limits were imposed in order to measure the effects of uncontrolled instruction prefetching. The limits measured were 1, 2, and 4 instructions.

TEST 6, Idle on duplicate instructions: This test was similar to test 3 except that the IFU will entered an idle state when the instruction it was decoding was a duplicate of one already resident in the microinstruction cache. While not necessarily a good prefetch strategy, this illustrated the degree of excess main store accessing in other strategies.

TEST 7, Alternate path selection: This test was the same as test 3 except that when a conditional control transfer instruction was encountered, the IFU checked whether it was already resident in the microinstruction cache. If so, the path opposite that indicated by the static prediction bit was followed. Unlike the strategies in test 2 and 3, this heuristic had no difficulty in continuing to prefetch beyond loop constructs.

Test	Clocks	Clks/Inst	Mreads	IC%hit	IFU%idle	uICmiss	uIC%hit
1	2,937,430	2.611	214,947	94.3	9.52	NA	NA
2	2,691,714	2.392	225,175	92.1	7.04	182,057	84.025
3	2,541,609	2.259	210,993	92.7	2.50	180,749	84.192
4_{16}	2,571,122	2.285	211,632	92.3	7.62	177,847	84.422
4_{32}	2,613,794	2.323	216,937	92.4	7.42	179,195	84.292
4_{64}	2,647,727	2.353	217,632	92.4	7.20	181,414	84.074
5_1	2,541,671	2.259	210,985	92.7	2.50	180,761	84.191
5_2	2,541,754	2.259	211,012	92.7	2.72	180,778	84.190
5_4	2,541,754	2.259	211,012	92.7	2.72	180,778	84.190
6	2,773,869	2.465	206,362	91.1	27.84	209,938	81.596
7	2,575,594	2.289	215,027	91.1	12.00	181,137	84.117

Figure 10: Prefetch Strategy Performance Measurements

●*Clocks*: The total number of basic machine cycles required to execute the benchmark program.

●*Clks/inst*: The ratio of basic machine cycles to the number of instructions executed. (This particular benchmark executed 1,125,115 instructions.)

●*Mreads*: Main store requests. *IC%hit*: Instruction cache hit rate -- the percent of instruction requests which were already cache resident.

●*IFU%idle*: Percent of the time the IFU was idle, either because it could not prefetch beyond an indirect control transfer, or because the prefetch strategy specified it.

●*uICmiss*: Number of microinstruction misses produced by EU requests.

•*uIC%hit*: Microinstruction hit rate -- the percent of microinstruction requests which were already cache resident.

These statistics provide a measure of the various prefetch strategies' relative merits. The close similarity in the total number of executed clock cycles for each of the intelligent prefetch strategies, tests 2 through 7, indicates that one heuristic is not significantly better than another. The largest variation in clock cycles, between tests 3 and 6, is only 9%. These statistics indicate that significant performance improvements are not provided by the individual prefetch strategies. Performance improvements must outweigh the additional hardware costs engendered in supporting these prefetch strategies.

For an architecture whose design goals included streamlined execution of an instruction each clock cycle, the Clks/inst statistics were not very encouraging. These numbers, however, were taken from a simulation run with only 256 microinstruction cache entries. The microinstruction cache size had been deliberately reduced to accentuate statistical differences among the prefetch strategies. With a more realistic cache size, the average number of clocks per instruction would be expected to decrease.

Figure 11 depicts the distribution of idle cycles that the EU waited for an instruction following a microinstruction cache miss. Whereas the overall pattern reflects attributes of the architecture, the local variations are a function of the measured prefetch strategies. Ignoring data cache contention, the maximum wait for each instruction cache miss is the sum of main store and IFU latencies. The simulated main store latency is six cycles, which are counted from the time the IFU detects an instruction cache miss until the first word from main store becomes available to the IFU. Since the IFU itself is a three stage pipeline, the earliest an instruction is available for loading into the instruction cache is three cycles after the word arrives from main store. However, since the instruction set has variable length instructions it is possible for the instruction to reside in two contiguous quad parcels. Such instructions accrue an additional cycle of latency. Therefore the microinstruction cache service latency will be either nine or ten cycles.

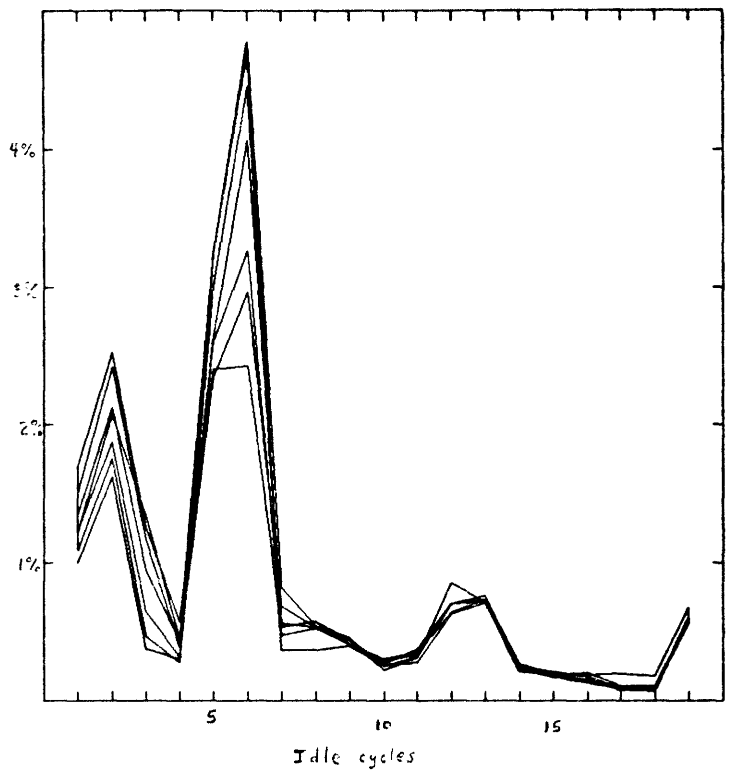

Figure 11: EU's Idle Cycle Distributions For Microinstruction Cache Miss

§6 Conclusions

It is easy to lose perspective in dealing with all of the complex issues and trade-offs in engineering a computer architecture. In the C Machine's initial design phase there was much excitement surrounding the possible prefetch strategies. Their relative unimportance was only realized after actual performance statistics were derived by simulation. The performance advantage was only a third that of simple branch folding.

The microinstruction cache of the C Machine is far more significant architectural feature and provides many performance improvements. For example, the simulator executes F. Baskett's "puzzle" benchmark[56] in about 16 million cycles with only 1.36 clocks per instruction.

The design methodology revealed that the performance advantage of one architectural feature was significantly less than initially anticipated. This same evaluation process can be applied to justify other design decisions. Empirical measurements identify which features can be synergistically combined to create a successful high performance system and which are not worth implementing. Most significantly, the described design process allows an architecture to be evaluated as a complete system rather than individual, isolated pieces.

Several other questions concerning prefetch strategies remain to be studied. For example, should the IFU follow procedure calls? The different domain of instruction addresses thus prefetched may conflict with instructions already resident in the microinstruction cache. Another area, is the interaction of prefetch strategies and cache sizes. Since a large microinstruction cache may contain a program's entire working set, once the instruction is resident, the IFU's prefetching activity is useless. Conversely, the high number of EU demand fetches caused by a small microinstruction cache's reduced hit rate interferes

with the IFU by preventing prefetching activity.

References

1. C. V. Ramamoorthy, "Pipeline Architecture," *Computing Surveys* 9(1) pp. 61-102 (March 1977).
2. Peter M. Kogge, *The Architecture of Pipelined Computers,* McGraw-Hill, New York (1981). IBM Federal Systems Division
3. Robert M. Keller, "Look-Ahead Processors," *Computing Surveys* 7(4) pp. 177-195 (December, 1975).
4. R. W. Holgate and R. N. Ibbett, "An Analysis of Instruction-Fetching Strategies in Pipelined Computers," *Transactions on Computers* C-29(4) pp. 325-329 (April 1980).
5. R. W. Holgate, *Analysis of Instruction Fetching Strategies in High Performance Computers,* Ph.D. dissertation, University of Manchester, Manchester, England (1977).
6. M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers* C-21(9) pp. 948-960 (Sept 1972).
7. R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development* 11(8) p. 25 (January 1967).
8. W. D. Connors, "The IBM 3033: An Inside Look.," *Datamation,* pp. 198-218 (May 1979).
9. Butler W. Lampson and Kenneth A. Pier, "A Processor for a High-Performance Personal Computer," *Proc. 7th IEEE/ACM Symp. on Computer Architecture,* pp. 146-160 (May 1980). (also in Xerox PARC technical report CSL-81-1)
10. Butler W. Lampson, Gene A. McDaniel, and Severo M. Ornstein, "An Instruction Fetch Unit for a High-Performance Personal Computer," Xerox PARC technical report CSL-81-1 (January 1981).
11. Cray Research Inc., *Cray-1 Computer System, Hardware Reference Manual,* Cray Research Inc., Mendota Heights, Minnesota (May 1980).
12. Bernard T. Murphy, Roger Edwards, Lee C. Thomas, and John J. Molinelli, "A CMOS 32b Single Chip Microprossor," *IEEE ISSCC 81, Proceedings of the 1981 Solid State Circuits Conference* 24 p. 230 (February 1981).
13. INTEL Corp., *The 8086 Family User's Manual,* INTEL Corp. (1979).
14. David R. Ditzel and Rae McLellan, *The C Machine: An Architectural Description,* (Bell Labs internal document) August, 1982.
15. B. Ramakrishna Rau and George E. Rossmann, "The Effect of Instruction Fetch Strategies Upon the Performance of Pipelined Instruction Units," *Proc. 4th IEEE/ACM Symp. on Computer Architecture,* pp. 80-89 (March 1977).
16. S. F. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journal of Research and Development* 11(8) pp. 8-24 (January 1967).

17. D. Morris and R. N. Ibbett, *The MU5 Computer System*, Springer-Verlag, New York (1979).

18. J. E. Thornton, *The Control Data 6600*, Scott, Foresman & Co., New York (1970).

19. A. G. Fraser and D. M. Ritchie, *C-Language Oriented Microprogram for the HP-21MX*, Bell Laboratories internal memorandum (1975).

20. S. C. Johnson, *A 32-Bit Processor Design*, Bell Laboratories internal memorandum (April 1979).

21. D. R. Ditzel, *A Simple 32-Bit Processor for VLSI Implementation*, Bell Laboratories internal memorandum (May 1981).

22. U. O. Gagliardi, "Report of Workshop 4 -- Software-Related Advances in Computer Hardware," *Proceedings of a Symposium on the High Cost of Software*, pp. 99-120 Stanford Research Institute, (1973).

23. S. Colley, G. Cox, K. Li, J. Rattner, and R. Swanson, "The Object-Based Architecture of the Intel 432," *COMPCON*, (February 1981).

24. B. G. Utley and al et, "IBM System/38 Technical Developments," IBM GS80-0237 (1978).

25. William D. Strecker, "VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family," *Proceedings National Computer Conference*, pp. 967-980 (June 1978).

26. D. A. Patterson and D. R. Ditzel, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News* 8(6)(October 1980).

27. Douglas W. Clark and William D. Strecker, "Comments on the Case for the Reduced Instruction Set Computer," *Computer Architecture News* 8(6) p. 34 (October 1980).

28. George Radin, "The 801 Minicomputer," *Proceedings Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47 ACM, (March 1982).

29. David R. Ditzel and David A. Patterson, "Retrospective on High-Level Language Computer Architecture," *Proc. 7th IEEE/ACM Symposium on Computer Architecture*, p. 97 (May 1980).

30. David A. Patterson and Carlo H. Sequin, "Design Considerations for Single-Chip Computers of the Future," *IEEE Transactions on Computers* C-29(2) pp. 108-115 (February 1980).

31. H.L. Caswell and K.L. Leiner, "The Key to High Performance Processors -- Packaging," *ICCC 82*, pp. 2-5 (Sept 28, 1982).

32. David R. Ditzel and H. R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," *Proceedings Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 48-56 ACM, (March 1982).

33. Robert D. Russell, "The PDP-11: A Case Study of How Not to Design Condition Codes," *Proceedings Fifth Annual Symposium Computer Architecture*, pp. 190-194 (1978).

34. L. C. Widdoes, "The S-1 Project: Developing High Performance Digital Computers.," *Proc. IEEE Compcon*, pp. 282-291 (February 1980).

35. DEC, *PDP10 Processor Handbook*, Digital Equipment Corporation, Maynard Massachusetts (1972).

36. J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI Processor Architecture," *Proceedings CMU conference on VLSI Systems and Computations*, (October 1981).

37. James W. Rymarczyk, "Coding Guidelines for Pipelined Processors," *Proceedings Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 12-19 ACM, (March 1982).

38. E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers* C-21(12) pp. 1405-1411 (December 1972).

39. DEC, *VAX11 Architecture Handbook*, Digital Equipment Corporation, Maynard Massachusetts (1979).

40. P. J. Weinberger, *Dynamic Statement Counting: A Manual for Users and Owners*, (BTL internal Memorandum) February 1982.

41. F. J. Aichelmann, "Memory Prefetch," *IBM Technical Disclosure Bulletin* 18(11) pp. 3707-3708 (April 1976).

42. David Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *Proc. 8th IEEE/ACM Symp. on Computer Architecture*, pp. 81-87 (1981). Control Data Corporation

43. Alan Jay Smith, "Cache Memories," *Computing Surveys* 14(3) pp. 473-530 ACM, (September 1982).

44. B. T. Bennett and P. A. Franaczek, "Cache Memory with Prefetching of Data by Priority," *IBM Technical Disclosure Bulletin* 18(12) pp. 4231-4232 (May 1976).

45. A. L. Bergey Jr., "Increased Computer Throughput by Conditioned Memory Data Prefetching," *IBM Tenchnical Disclosure Bulletin* 20(10) p. 4103 (March 1978).

46. Alan Jay Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer* 11(12)(December 1978).

47. William D. Strecker, "Cache Memories for PDP-11 Family Computers," *Proceedings Third Annual Symposium Computer Architecture*, pp. 155-158 (January 1976).

48. R. T. Blosk, "The Instruction Unit of the Stretch Computer," *Proceedings Eastern Joint Computer Conference*, (18) pp. 299-324 (December 1960).

49. F. P. Brooks, "Instruction Sequencing," in *Planning a Computer System: Project Stretch*, ed. Werner Buchholz,McGraw-Hill (1962).

50. E. Bloch, "The Central Processing Unit," in *Planning a Computer System: Project Stretch*, ed. Werner Buchholz,McGraw-Hill (1962).

51. R. S. Ballance, John Cocke, and H. G. Kolsky, "The Look Ahead Unit," in *Planning a Computer System: Project Stretch*, ed. Werner Buchholz,McGraw-Hill (1962).

52. James E. Smith, "A Study of Branch Prediction Strategies," *Proceedings 8TH IEEE/ACM Symposium on Computer Architecture*, pp. 135-142 (1981). CDC

53. S. C. Johnson, "A Tour Through the Portable C Compiler," *Unix Programmer's Manual* 2BBell Laboratories, (1977).

54. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proceedings 5th ACM Symposium on Principles of Programming Languages*, pp. 97-104 (January 1978).

55. J. C. Gibson, "The Gibson Mix," (Report TR 00.2043), IBM Systems Development Division, Poughkeepsie, New York (1970).

56. Forrest Baskett, *puzzle benchmark*, (widely circulated)