Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

SIMULATION AND ANALYSIS OF A PIPELINE PROCESSOR

P. G. Emma
J. W. Knight
J. H. Pomerene
T. R. Puzak
R. N. Rechtschaffen
Processor Organization Project
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT

In this paper we describe a software simulator (a timer) that is used to model a wide range of pipeline processors. A set of performance equations is developed that allow a user to separate the performance of a processor into its infinite-cache and finite-cache performance values. We then use the timer to study the performance of two different machine organizations. Performance curves are presented that help a user compare the performance of each organization (in terms of MIPS and cycles per instruction) to the cycle time chosen to implement the design.

1. INTRODUCTION

Due to the increasing complexity of a modern computer's design, it is essential that software models be available during the early design stages of a processor. Designers must consider the effects of the many possible combinations of design parameters to produce a 'good' design. Technology constraints must be considered along with computer architecture requirements, pipeline behavior, instruction characteristics, cache effects, and many more processor features before actual manufacturing of the processor can begin. Once manufacturing has begun, redesign or manufacturing changes may be cost prohibitive to meet any unforeseen performance concerns over the initial design. In many cases, even hardware prototypes are no longer a design option due to increasing design costs and hardware complexity. For these reasons, software models that can give an accurate indication of the processor's performance, before the manufacturing process begins, become an essential ingredient in helping designers weigh the many variables that make up a good design.

In general, there are two techniques available for analyzing the performance of a processor's design. The first approach is analytical in nature and involves mainly the construction of queueing models. This approach typically requires many input parameter assumptions to characterize the design. Additional assumptions may be required to characterize any instruction interdependencies and cache behavior. Unfortunately, many of the initial input parameters may be unknown when the design phase of the computer begins. For these reasons, these models are typically limited to a simple description of the processor being designed.

The second approach involves discrete simulation. This approach requires the construction of a software model that represents the internal organization of the computer hardware. These hardware simulators are also called *timers* and these two terms will be used interchangeably throughout this paper. The level of detail specified in a timer can vary greatly. Some timers may require specifications down to the internal logic of the machine while other models may try to represent the processor's hardware on a functional level. In this paper, we concentrate on timers that only require a functional description of a machine they are to model because, we assume, that the timer is used primarily in the early design stages and that a detailed logical description of the machine does not exist. A trace tape is used as input to these simulation models, where a trace tape contains the actual sequence of instructions and addressing information collected from executing a known benchmark on an existing machine.

The actual design of the software simulator can be just as challenging as the hardware it is supposed to model. In order to investigate the total range of design trade-offs, the timer must be able to simulate a wide range of processors since not all design variations can be foreseen during the early stages of the processor's design. Design decisions may include whether to have a common or split cache (instruction and data cache), instruction set characteristics (a RISC or CISC instruction set), pipeline length, and bus sizes. The timer must be modular and easy to change since not all of the design variations can be investigated through parameters supplied by the timer. Actual software modifications should be able to be accomplished by the hardware designer and not require someone with a software expertise. And finally, the timer must be able to give detailed and accurate cycle-by-cycle performance information as well as be efficient.

In this paper we describe the structure and use of a software simulator that is capable of modeling a wide range of machine organizations. In Section 2 we discuss pipelining and areas of related work. Section 3 describes the performance methodology used to analyze a processor. Section 4 describes the structure of the timer. In Section 5 we present two experiments that use the timer to investigate two possible design trade-offs that a designer can be faced with. Finally, in Section 6 we present our conclusions.

2. PIPELINING AND RELATED MATERIAL

In this section we discuss pipelining and related publications. Few papers specifically deal with timers and pipelining but there are many papers that analyze the performance of the individual components that make up a pipeline processor.

2.1 Pipelining

Computer designers have used pipelining as a means of improving the performance of a modern computer. Pipelining allows the designer to partition the computer into separate components where each component is responsible for completing a different phase of an instruction's execution. For example, Figure 1 shows six instructions I_1, I_2, I_3, I_4, I_5, and I_6 passing through a pipeline consisting of the following phases: Decode, Address Generation (AGEN), Cache Access (FETCH), Execute, and Putaway (store the results).

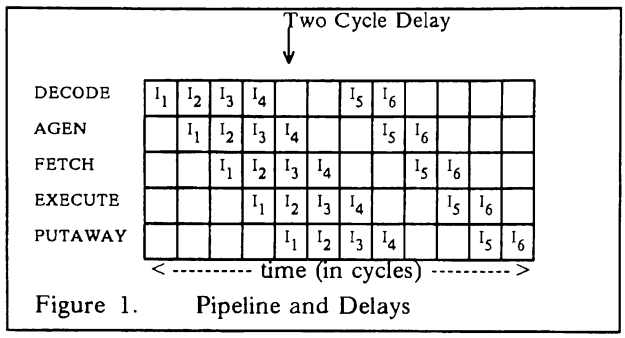

Figure 1. Pipeline and Delays

Each instruction enters the pipeline at the decoder and spends one cycle in each of the pipeline stages. Individually, each instruction takes five pipeline cycles to complete. However, if the pipeline can be kept full then each component of the processor (decoder, execution unit, cache) can be kept actively working on a different instruction, each at a different pipeline stage, and one instruction will complete on every cycle. Unfortunately, keeping the pipeline full is not an easy task. Instruction interdependencies often disrupt the pipeline and cause delays that reduce the overall processing potential of a pipeline. The figure shows a pipeline disruption occurring after I_4 and lasting for two cycles. This is represented by a two cycle gap between decoding instructions I_4 and I_5. In this example, the delay is introduced into the pipeline at the decoder and passes through the pipeline delaying each stage by two cycles. Each delay then contributes to the average number of cycles needed to process an instruction. A computer designer must try to reduce the frequency and duration of each pipeline delay in order to achieve the full processing potential of the pipeline.

2.2 Related Material

Using a trace tape to analyze the performance of a processor has become a widely adopted practice. The literature is full of papers that separately measure the performance of the individual components that make up a processor. Many of these papers use trace tapes as input to obtain their results. The cache serves as a good example. There are many papers that study the performance of the cache [*STON*87], and [*SMIT*82]. Replacement algorithms and cache behavior are studied in [*BELA*66], and [*MATT*66]. All are trace tape driven.

The effect of branch behavior on performance has also been studied through trace tapes. Branch prediction mechanisms are described in [*POME*87], and [*SUSS*71]. Each describes a different mechanism that can be used to improve branch prediction. Instruction unit (decoder) and execution unit performance have also been studied. Algorithms to control multiple execution units are given in [*TOMA*67]. The instruction unit and pipeline disruptions are studied in [*ANDE*67]. Still other papers incorporate trace tapes to analyze the relationships of different components of a processor. Pipeline performance and instruction behavior are studied in [*RYMA*82] and [*EMMA*86]. Emma and Davidison in [*EMMA*87] characterize branch behavior along with instruction and data dependencies while investigating pipeline performance.

Once a processor is built its performance can be analyzed through hardware monitors. Instruction use and processor performance for the VAX-11/780 are measured and analyzed in [*CLAR*82]. Instruction set usage in the System/370 is analyzed in [*LAMA*86].

A software simulator must be capable of simultaneously analyzing the performance of all of the components in a processor as well as collecting any information that is consistent with having a hardware monitor attached to a running processor. Only by gathering information on the cache, instruction unit, execution unit, and pipeline can a researcher perform the necessary performance analysis to accurately weigh any design trade-offs. The number of papers on processor simulators is limited. Many simulators represent the internal organization for a proprietary machine design and are rarely published. MacDougall in [*MACD*84] hints at the use of a cycle-by-cycle machine simulator while discussing pipeline performance and processor modeling.

Once the timer is written, the system designer will use trace tapes as input to the timer to evaluate the performance of the planned processor. The designer may want to investigate the performance of a machine under different workloads, (interactive, batch, scientific), each represented by a different trace tape, to be assured that the design is robust and not customized to a specific environment. In the next section we describe a performance methodology and define all the terms needed to evaluate the performance of a pipeline processor.

3. METHODOLOGY

Processor designers are constantly faced with evaluating the performance of a pipeline processor. This includes the organization of the instruction and execution units as well as the performance of the memory hierarchy. Described below are a set of equations that relate the performance of individual components to the overall performance of the processor. In developing these equations, we assume that the processor has a single

processing element that can only execute one instruction per cycle and that there only exists one level of intermediate storage, a cache, between the processor and memory. By analyzing systems that can decode and execute multiple instructions per cycle and have multiple levels in their storage hierarchy (a first and second level cache) we would only introduce a level of complexity that is unnecessary for the topics discussed in this paper.

The performance of a computer system is a function of the machine's organization coupled to the technology used to implement the organization. An equation for evaluating the overall performance of a computer system is:

$$\left(\frac{\text{nanoseconds}}{\text{cycle}}\right)\left(\frac{\text{cycles}}{\text{instructions}}\right)\left(\frac{1}{1000}\right)=\frac{1}{\text{MIPS}} \quad (1)$$

where the term

$$\left(\frac{\text{cycles}}{\text{instructions}}\right)$$

represents the average number of cycles needed to execute an instruction, an overall performance measure of the machine's organization. The designer will try to include any and all design features in order to reduce this term. Techniques such as pipelining, cache and bus management, and parallel processing are all used to reduce the average number of cycles needed to execute an instruction.

The term

$$\left(\frac{\text{nanoseconds}}{\text{cycle}}\right)$$

defines the cycle time used in the processor and represents any underlying constraints imposed by the technology on the organization. Clearly, the cycle time along with the cycles per instruction term defined above interact with each other. A designer is always faced with making organizational trade-offs to conform to any technology limitations. For example, certain processor enhancements that can reduce the overall cycles per instruction term defined in (1) may have to be abandoned due to a disproportionate increase in the cycle time needed to implement these enhancements. However, the product of these two terms, when properly scaled, computes inverse MIPS, *millions of instructions per second* executed.

The performance of the memory hierarchy as well as the performance of the instruction and execution unit contribute to the overall performance of the computer system. The cycles per instruction term defined in (1) can be expressed as:

$$\left(\frac{\text{cycles}}{\text{instruction}}\right)=\left(\frac{\text{cycles}}{\text{instruction}}\right)_{\infty C}+\left(\frac{\text{cycles}}{\text{instruction}}\right)_{FC} \quad (2)$$

where

$$\left(\frac{\text{cycles}}{\text{instruction}}\right)_{\infty C}$$

represents the average number of cycles required to execute an instruction if all memory references are satisfied at cache speeds. This term, in effect, assumes that the cache is *infinite* in size and thus removes the memory hierarchy from the processor's design. This term relates the performance of the processing elements of the machine (decoder and execution unit) to the pipeline structure.

The infinite cache performance of a processor can be divided into still further components that define an *intrinsic* amount of work that is required by the underlying organization to execute an instruction. These components are:

$$\left(\frac{\text{cycles}}{\text{instruction}}\right)_{\infty C}=\left(\frac{\text{cycles}}{\text{instruction}}\right)_{EBUSY}+\left(\frac{\text{cycles}}{\text{instruction}}\right)_{ENBUSY} \quad (3)$$

Here the term

$$\left(\frac{\text{cycles}}{\text{instruction}}\right)_{EBUSY}$$

describes the average number of cycles needed to execute an instruction provided the execution unit can be kept *busy* all of the time. This term represents the *peak rate* that a processor can execute instructions for a given workload. In a machine with a single processing element and a RISC-like instruction set this ratio will be close to one. In computer architectures with a more complex instruction set and multi-cycle instructions this ratio will typically be much greater than one. Each instruction executed by the processor contributes to this term and it is possible to express this ratio as the sum of the individual EBUSY components attributed to each instruction. A designer would pursue this line of analysis in order to understand the contributions that each instruction, or group of instructions, have on the overall performance of the processor. For example, it may be determined that the floating point, decimal, or string manipulating instructions require some additional hardware to reduce their average EBUSY time in order to meet certain performance goals that were set at the start of the project.

The term

$$\left(\frac{\text{cycles}}{\text{instruction}}\right)_{ENBUSY}$$

represents the difference between the infinite cache performance of a processor and the EBUSY term and is usually the result of pipeline disruptions. It describes the average number of cycles that the execution unit is *not busy* (idle) while executing the instructions found on a trace tape. The ENBUSY term is the result of pipeline delays and disruptions caused by instruction interactions, cache bandwidth limitations, and pipeline length and it is again possible to define this term as the sum of the delays introduced by each individual instruction as was the EBUSY term defined above. Examples of these delays can be found in [*RYMA*82], [*MACD*84], and [*EMMA*86], and include interlocks between instructions (both address and execution), taken-branch

delays, storing into the instruction stream, and decoding delays caused by the instruction buffer being empty.

The term

$$\left(\frac{\text{cycles}}{\text{instruction}}\right)_{FC}$$

in (2) represents the delay incurred because the cache size is, in reality, *finite*. Since the cache is of finite size, a certain portion of the references made by the processor miss in the cache and must be satisfied from main memory. These requests take several cycles longer to complete since the access time to memory is much longer than the access time to the cache. This term accounts for the performance of the memory subsystem.

The finite-cache delay can be expressed as the product of the following three terms:

$$\left(\frac{\text{cycles}}{\text{instruction}}\right)_{FC} = \left(\frac{\text{cycles}}{\text{miss}}\right)\left(\frac{\text{misses}}{\text{reference}}\right)\left(\frac{\text{references}}{\text{instruction}}\right) \quad (4)$$

where

$$\left(\frac{\text{cycles}}{\text{miss}}\right)$$

represents the average number of cycles of delay introduced by a cache miss. This term can be reduced by using faster main memory, increasing the bandwidth between the cache and main memory, or increasing the amount of overlap between instruction execution and miss processing.

The term

$$\left(\frac{\text{misses}}{\text{reference}}\right)$$

represents the average rate at which a reference misses to the cache. This rate is also referred to as the miss ratio and along with the hit ratio, which is (1 - miss ratio), is the metric that is commonly used to evaluate the performance of the cache.

The term

$$\left(\frac{\text{references}}{\text{instruction}}\right)$$

represents the average number of cache references that must be made to execute an instruction. This number includes all instruction fetches, data fetches and stores. Cache bandwidth limitations and bus widths are the primary factors that control this value. However, other organizational features that affect this value are the instruction buffer size, pipeline strategies that affect branch prediction, and pipeline concurrency.

Substituting (2), (3), and (4) into (1) yields:

$$\left[\left(\frac{\text{cycles}}{\text{instrn}}\right)_{EBUSY} + \left(\frac{\text{cycles}}{\text{instrn}}\right)_{ENBUSY} + \left(\frac{\text{cycles}}{\text{miss}}\right)\left(\frac{\text{misses}}{\text{refs}}\right)\left(\frac{\text{refs}}{\text{instrn}}\right)\right]\left(\frac{\text{ns}}{\text{cycle}}\right) = \frac{1000}{\text{MIPS}} \quad (5)$$

From (5) we see the separation of the processor's internal organization (the pipeline length, instruction unit and execution unit) from the performance of the cache and memory. Overall performance improvements can be obtained by making organizational changes that improve any individual component. For example, improving the performance of the memory hierarchy will usually not affect the infinite cache performance of the processor. Similarly, making organizational changes that improve the EBUSY or ENBUSY components of a processor will generally not affect the performance of the memory hierarchy. For this reason large organizational performance improvements are usually the result of several individual design changes and not just the result of a single design change.

4. TIMER STRUCTURE

In this section we describe the structure of the timer and its components. An engineer will use the timer to analyze many different machines by varying the input parameters. In order to provide this function, the timer is organized into separate components where each component allows the user to identify and isolate any programming errors, design changes, and analyze any performance limitations associated with the machine being modeled. The timer consists of the following components:

- Machine Definition
- Instruction Plot
- Summary Statistics (reports)
- Modeling Loop (Timer structure)

A brief explanation of each component is given below:

The **machine definition** component of the timer gives the user the ability to define and model a wide variety of machines. The definition of the machine can be logically separated into two components:

- machine architecture
- machine organization

Each part is usually defined as a separate input file to the timer that is read at the beginning of each simulation run.

The *architectural* description of the machine consists of a table that describes the format and pipeline characteristics of each instruction that is defined in the architecture of the processor. This information includes:

- The Instruction Format: The opcode, instruction length, mnemonic.
- Instruction Type: A branch, serializer, load, store.
- Operand Information: Base, index, and displacement locations, length parameters, fetch and store algorithms
- Decoding Information: Decoding cycles, interlock information.
- Execution Unit Information: Execution cycles, execution algorithms, execution rate (in bytes, words).

The timer will then use this information to determine the proper cycle to decode, and execute each instruction as well as control the pipeline behavior of all fetches and stores. For example, the timer can maintain a list of registers that will be changed for each instruction that is currently in the pipeline. Register interlocks can then be detected by comparing this list against the set of registers needed to decode the next instruction. If a match occurs then a register interlock is present.

The *organizational* description of the machine defines the basic size and structure of the components that make up the processor (decoder, cache, execution unit). Some of the parameters defined in the organization file are:

- Pipeline Stages; Decode, address generation, fetch, execute, putaway.
- Cache Parameters: Cache size, replacement algorithm, number of ports and widths.
- Queue Sizes: Instruction buffers, execution buffers, operand buffers.
- Branch Prediction Strategies: Guess taken/fall thru strategies, branch error delays, restart delays.
- Decoder/Execution unit capabilities: Amount of concurrency, overlap.

The amount of detail contained in these two files depends on the amount of information known about the machine that is being modeled. During the early design stages of a processor, many of the design parameters may be unknown and default values may have to be supplied. However, as the design nears completion, new parameters may be needed to describe new hardware that is integrated into the design of the machine.

The **summary statistics** must give a detailed report analyzing the performance of the various components defined in the machine's design. Contained in the output are the total number of cycles needed to simulate the execution of the trace tape, instruction frequencies and totals, and EBUSY and ENBUSY analysis. Also, each component of the proposed design (the decoder, execution unit, and cache) is analyzed in detail.

The summary report provides a user with a detailed analysis of all events that delayed the decoder. The most common delays are:

- Instruction interlocks: This delay occurs when one instruction needs the result from another instruction in the pipeline and the value is not yet determined.
- Decode Wrong Stream: This delay occurs when the outcome of a conditional branch (taken, or not taken) is guessed incorrectly. The decoder is delayed until the branch can be resolved and the correct decoding direction can resume.
- Instruction buffer empty: This delay occurs when the instruction buffer is empty and the decoder cannot decode an instruction. This is usually the result of cache bandwidth limitations.
- Serialization: Certain instructions are architected to drain the pipeline and be executed alone (i.e. Compare and Swap in IBM System/370 architecture). These instructions disable the decoder until they finish execution.
- All operand buffers full: Typically machines have a limit on the total number of operands that can be accessed in any one cycle. If this limit is currently reached and the instruction that is being decoded requires an operand then the decoder must wait until an operand buffer becomes available.

The summary report for the execution unit will list the performance of the machine in terms of the EBUSY and ENBUSY components defined above. On every cycle of the simulation run, the timer must monitor the execution unit and determine whether it is busy or idle. Each cycle is then identified as either contributing to the EBUSY or ENBUSY term, respectively. Usually, each instruction spends only one cycle in the execution unit before completing. However, in computer architectures with multi-cycle opcodes, an instruction may require several thousand cycles of execution before completing. For example, the *MVCL* instruction in the IBM System/370 architecture can fetch and store up to 16 megabytes of memory. The summary output will then list the contribution that each instruction makes to the total EBUSY term. The total EBUSY term (in cycles per instruction) is then the weighted sum of the individual EBUSY terms where the weights are the relative averages that each instruction contributes to the overall instruction total.

Identifying the cause of each ENBUSY delay is more difficult than was collecting the EBUSY cycles. Many of the delays that appear in the execution unit are introduced into the pipeline by the decoder (usually as a decoder delay) and are passed through the pipeline to the execution unit. Also, many of the decoder delays may be the result of multiple causes and identifying a single reason may be difficult. For example, an instruction can follow a serializer, be interlocked on an address calculation, and have all of the operand buffers full. In this example, any one of the three delays listed above will individually delay the decoder. Accounting for this delay in the execution unit (if it appears) involves remembering the sequence of events that led to the decoder delay and recognizing these causes in the execution unit. The ENBUSY component can then be examined in terms of each individual delay or taken in context with multiple delays that happened earlier. The mechanisms and structures needed to accurately account for these events will be discussed in the modeling-loop component of the timer described below.

The performance of the memory hierarchy is usually derived from the performance statistics supplied from infinite-cache and finite-cache simulation runs. An infinite-cache simulation run assumes that every refer-

ence is found in the cache and can be satisfied at cache speeds. In a finite-cache simulation run, the timer must first maintain a cache directory to identify which lines of memory are currently in the cache. Any reference not found in the directory (a miss) is delayed and satisfied from memory. The timer must then represent this delay by simulating the logic of the processor to process the miss. The performance of the memory hierarchy (in cycles per instruction) is then the difference between a finite-cache simulation run and an infinite-cache simulation run. The average miss delay (in cycles per miss) is then the product of the miss rate (in terms of instructions per miss) with the performance of the memory hierarchy (in cycles per instruction).

The **instruction plot** provided by the timer is the designer's main interface between the software description of the machine and its physical design. The plot must provide the user with the information needed to understand the performance of the machine on a cycle-by-cycle basis. The plot can be used by many people and in different ways.

The designer uses the plot to verify that the machine being modeled is the actual machine that the designer has in mind. This usually involves examining several instruction sequences on an instruction-by-instruction and a cycle-by-cycle basis to verify that each instruction is decoded and executed in the proper cycle, and that any pipeline delays (in the decoder or execution unit of the machine) are properly recorded.

The programmer uses the plot as a debugging aid. If the design of the timer is modular then its components can reflect the components and structure of the actual machine. The programmer can then display, thru the plot, the subroutines used by the timer to simulate the pipeline just as the designer uses the plot to identify the internal state of the processor. This feature allows the programmer to identify and isolate quickly any areas of the program that need to be modified whenever program enhancements or debugging changes are necessary.

Figure 2 gives an example of an abbreviated instruction plot provided by the timer. The plot shows a program that computes the value of $(A^2 + B^2) / C^2$. (Typically, several pages of output are required to display all of the information associated with simulating this program). Because of the underlying architecture of the machine, the program is written in a language that requires all operands to be first placed in a register before any calculation can occur. We assume that all values are integers, that multiply overflow cannot occur and that the divide instruction only returns the value of the quotient. The remainder, if any, is ignored. In order to highlight pipeline disruptions, we assume that each operand address is interlocked. That is, the address of each operand (A, B and C) must be loaded into a register before that value of the operand can be established. The actual program is given as part of the plot output. We assume that the instruction formats are similar to those given in the IBM System/370 Principles of Operation manual [*IBM*81]. The timer parameters were set to model a machine with three pipeline stages: Decode/AGEN, Fetch, and Execute.

The plot identifies on each cycle all instructions, fetches, and stores that occur over a specified interval (cycles). The plot lists each instruction that is read from the trace tape and identifies each instruction fetch that is made to execute these instructions. The plot also identifies the cycle that each instruction was decoded, lists the contents of the execution queue and instruction buffer, identifies each EBUSY cycle, and lists each reference (instruction or datum) that is made to the cache and when it returns. Each instruction that is read from the trace tape is assigned a *tag* where a tag is a letter identifier that is used to represent the instruction as it passes through the pipeline. For example, the first four instructions read from the trace tape (L, L, LR, and MR) are assigned the tags 'A', 'B', 'C' and 'D', respectively. The plot shows that instruction 'A' (the load) was decoded on cycle 2, fetched its operand on cycle 3, and executed on cycle 4. Similarly, instruction 'D' (the multiply register, MR) was decoded on cycle 6 and required six cycles in the execution unit to complete, cycles 8 through 13 of the EBUSY line.

The plot allows the designer to follow each instruction through the pipeline and identify each pipeline disruption. The signals listed at the end of the plot identify each delay that occurs in the decoder and execution unit. For example, the decoder was idle on the first cycle of the simulation run because the instruction buffer was empty. In this case the decoder will remain idle until an instruction can be fetched from the cache and placed in the instruction buffer. The decoder was also idle on cycles 3, 8, and 17 of the simulation run. These delays are the result of the three operand-address interlocks described above. Each delay lasts for one cycle and is removed when the address of the operand (A, B, or C) is fetched from the cache. This allows the address generation phase of the following instruction to complete. The remaining amount of time (cycles) that the decoder is idle is the results of the execution unit being full.

Not all of the decoder delays appear as delays in the execution unit. The plot shows that only the instruction-buffer-empty delay and the first instruction-interlock delay appear as delays in the execution unit. (These delays appear as idle cycles on the EBUSY line of the plot and are identified by the execution-unit signals listed at the end of the plot.) The remaining decoder delays are overlapped in the execution unit with useful EBUSY cycles because of the multi-cycle instructions (the MR and DR) that appear in the program.

The structure of the pipeline can usually be seen through the **modeling loop.** Contained in the loop are the sub-

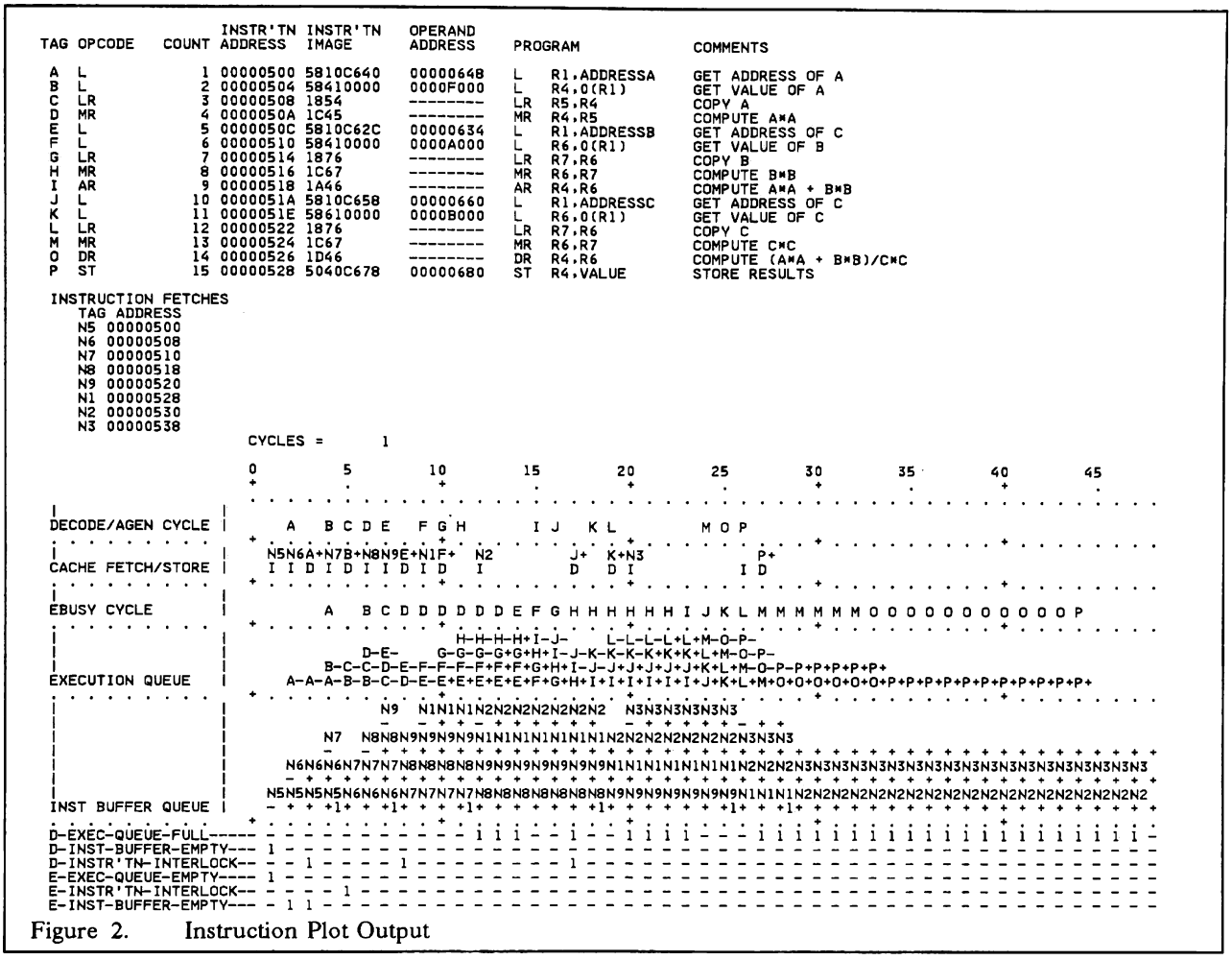

Figure 2. Instruction Plot Output

routines used to model each machine. Modeling a pipeline is then easy. Each subroutine in the loop represent a physical components of the machine. (decoder, cache, execution unit). Time, in cycles, is then represented by the order that the subroutines are arranged and executed. For example, consider a pipeline with the following stages: Decode, Address Generation (AGEN), Cache Access (Fetch), Execute, and Putaway. A separate subroutine is written for each pipeline stage. These subroutines, along with housekeeping, instruction fetching (IFETCH), statistics and tracing subroutines, are embedded in a loop to represent one machine cycle (Figure 3).

```
DO until end-of-trace
  CALL HOUSEKEEPING
  CALL IFETCH

    CALL DECODE
    CALL AGEN
    CALL CACHE
    CALL EXECUTE
    CALL PUTAWAY

  CALL TRACE
  CALL STATISTICS
END
```

Figure 3. Timer Loop

Each subroutine will have its own queue to receive instruction and operand information from the other subroutines in the calling loop. This information is presented to each subroutine by placing a token on its queue, where a token represents a pointer to a data structure that describes the instruction or operand. Each token is used to represent *work* (in the form of an instruction or operand) that a subroutine must process. The concept of instructions passing through the pipeline is accomplished by having each subroutine pass tokens (representing instructions and operands) from one queue to another until each instruction is executed and the results are processed from the PUTAWAY queue. For example, each instruction found on the trace tape will initially be placed on the DECODER's queue. The DECODE subroutine will then move the instruction (token) to the AGEN queues when decoding is complete. The AGEN subroutine will check for all interlocks and at the appreciate time (cycle) place the operands on the CACHE queue and pass the instruction onto the EXECUTION queue. The instruction will then wait in the EXECUTION queue

until the operands are available from the cache. The EXECUTION subroutine will then 'execute' the instruction. Upon completion, the EXECUTE subroutine will move the instruction to the PUTAWAY queue where all stores are competed and the instruction is finished.

In the above example, we see tokens being used to pass instruction and operand information through the pipeline. However, the tokens can also be used to carry delay-information through the pipeline. Consider the case where the decoder is idle due to an instruction interlock. We would like to be able to understand and record the consequences of this event as it passes through the pipeline, especially if it produces an idle cycle (ENBUSY) in the execution unit. This can be accomplished by using tokens to pass delay information through the pipeline just as they were used to pass instruction and operand information. Pipeline events can then be monitored by having each subroutine pass a token on every cycle. The token will describe the action of the subroutine for that cycle. If the decoder was idle then the token information will describe the events that produced the delay (ie. instruction interlock, serializer). If the decoder was busy doing useful work (decoding instructions) then the token information will identify the instruction that was decoded. Now, whenever the execution unit is idle, a token can be found in its queue describing the events that took place in the decoder (either busy or idle) several cycles earlier. This mechanism allows the timer to accurately identify and record each delay that appears in the pipeline.

5. ANALYSIS

In this section we will use the timer to evaluate the performance of two different machine organizations. The machines differ in the number of stages that are used to describe their pipelines. The first machine (Model 1) has four pipeline stages consisting of the following steps: decode/AGEN, operand fetch, execute, and putaway. The second machine (Model 2) has six pipeline stages: decode, AGEN, operand-fetch-1, operand-fetch-2, execute, and putaway. Figure 4 shows four instruction passing through these two pipelines. The figure shows that each instruction in Model 1 spends at least four cycle in the pipeline while an instruction in Model 2 requires six cycles.

The timer will be used to simulate each machine organization and produce infinite and finite-cache performance measurements. Each measurement will be given in terms of the performance equations defined in Section 3. A designer would then use this information to evaluate any design decision trade-offs.

Studies of this type are common. A designer must always consider the effects of the machine's cycle time on the performance of the organization, in terms of cycles per instructions. For example, a designer may wish to evaluate performance trade-offs between a machine with a long cycle time and few pipeline stages (Model 1) versus a machine with a very fast (short) cycle time and a much longer pipeline (Model 2). We assume that the long cycle time in Model 1 allows a designer to place more function (logic) into each pipeline cycle whereas the short cycle time of Model 2 limits the amount of logic that can be designed into any single cycle. In Model 1, the cache can be accessed in a single cycle and the decode and address generation phases of an instruction can be combined into a single pipeline stage. In Model 2, the short cycle time requires that the decode and address generation phase of an instruction be split into their own pipeline stages and that two machine cycles are needed to complete a cache access.

Except for the pipeline the remaining features of the two machines are similar. Each machine represents a RISC-like architecture that has the processing potential to decode and execute one instruction per cycle. The cache can be accessed once per cycle and the data path between the decoder, cache and execution unit is four bytes wide. The instruction buffer can hold up to four entries and an instruction fetch is attempted whenever the instruction buffer is not full. However, all operand fetches and stores are honored before any instruction fetch. All conditional branches are assumed to be not taken and branch prediction errors (conditional branches that are taken) are discovered when the branch is executed.

The timer parameters were set to reflect the machines described above. A trace tape was generated for a RISC-like processor running a scientific benchmark and run through the timer. Figure 5 shows the infinite-cache performance measurements from the simulation runs.

The results show that Model 2 required approximately 27 percent more cycles to execute an instruction than did Model 1 (1.9/1.5 = 1.266). Each machine experienced the same EBUSY performance component of 1.0 cycles per instruction due to the RISC architecture, while pipeline disruptions (ENBUSY) accounted for an average delay of .5 cycles per instruction in Model 1 and .9 cycles per instruction in Model 2. The major contributors to the ENBUSY delay were instruction interlocks, instruction buffer empty, and branch delays.

Typically, the pipeline delays in Model 2 are longer than the pipeline delays of Model 1. This is primarily the result of the increased length of pipeline in Model 2. Consider an interlock between two instructions. Figure 6 shows instructions passing through two pipelines represented by Model 1 and Model 2.

Assume an interlock is detected during the address generation phase of instruction I_3 and cannot be resolved until instruction I_2 finishes execution. We assume that the decode phase of each instruction is not interlocked. The figure shows that the interlock causes a two cycle delay in Model 1 but a three cycle delay in Model 2. The extra cycle of delay in Model 2 is due to the extra pipeline stage that exists between the address generation phase and execution phase of the pipeline, in this case the sec-

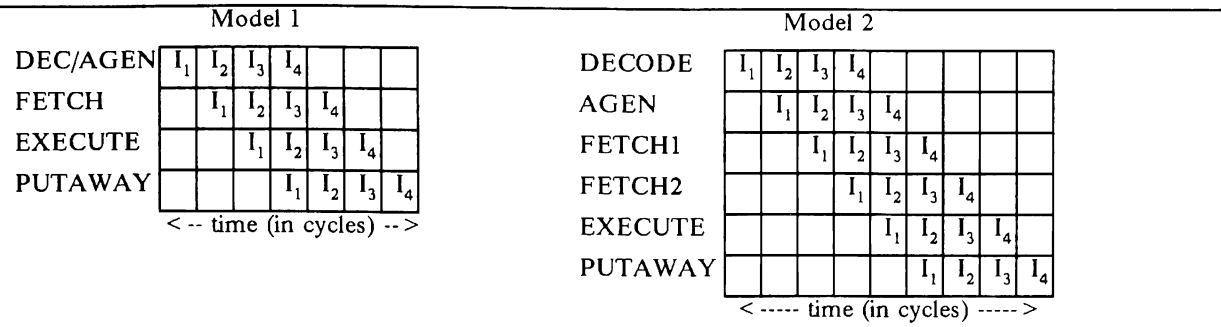

Figure 4. Pipelines for Model 1 and Model 2

	Model 1	Model 2
Infinite-Cache Cyc/Inst	1.5	1.9
EBUSY (cyc/inst)	1.0	1.0
ENBUSY (cyc/inst)	.5	.9

Figure 5. Infinite-Cache Performance

ond cache access cycle. Similar results would be produced by modeling the other pipeline delays (serializer, instruction buffer empty, etc.).

The performance measurements of Figure 5 can be converted into MIPS by choosing an appropriate cycle time for the organization. It is still possible for Model 2 to outperform Model 1 provided it can be implemented in a technology that can provide a faster cycle time than the technology used to implement Model 1. For example, a bipolar technology typically offers devices that are faster than the devices found in a CMOS technology. Also, certain organizations may prove to be too complex to implement in certain high-performance technologies when considering packaging delays, and cooling requirements and a less aggressive implementation may have to be pursued. Figure 7 plots MIPS as a function of the cycle time (in nanoseconds) for machine organizations that average 1.5 and 1.9 cycles per instruction, Models 1 and 2.

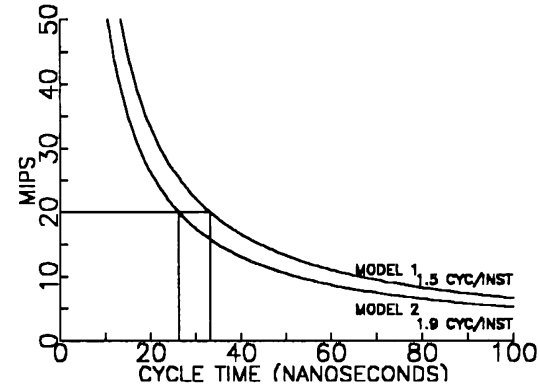

Figure 7. MIPS versus Cycle Time

The figure shows that if a 20 MIPS processor is desired then the cycle times for Model 1 and Model 2 must be approximately 33 and 26 nanoseconds, respectively. Implementing Model 2 in a technology that can produce a cycle time faster than 26 nanoseconds will produce more than 20 MIPS. For example, Model 2 would achieve 30 MIPS if a 17.5 nanoseconds cycle time could be achieved. A designer would continue to explore the organization described in Model 2 in the hope that the simpler pipeline structure can produce a faster cycle time,

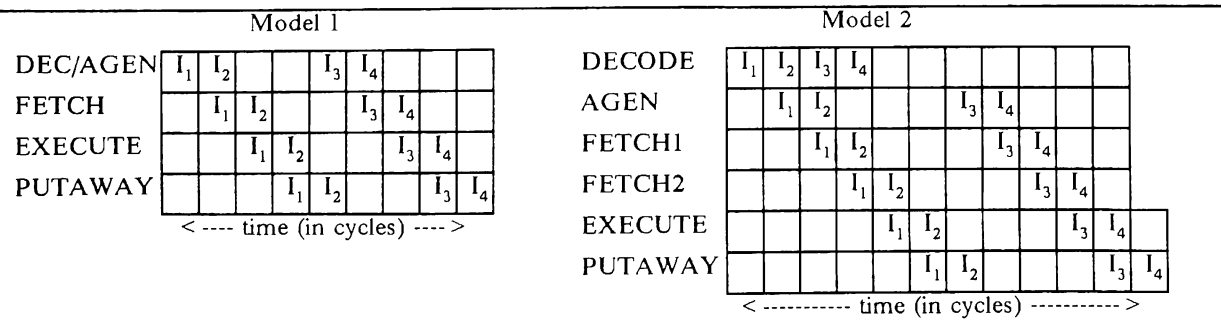

Figure 6. Pipelines with delays for Model 1 and Model 2

and thus more MIPS, than the organization described in Model 1.

Once the designer is satisfied with the infinite-cache performance of a processor the finite-cache performance can be determined. The timer parameters are again set to simulate the machines defined in Models 1 and 2. However, the cache parameters are also defined to reflect an 8K cache with a 16 byte linesize. All stores are sent simultaneously to the cache and memory. This eliminates the need to castout any modified line during the miss process. When a miss occurs, we assume that the memory access time will delay the original access by 5 cycles and that a line of memory is transferred to the cache at the rate of four bytes per cycle, for four cycles. Figure 8 shows the results of the simulation runs.

The overall performance of Model 1 and Model 2 is now 2.0 and 2.4 cycles per instruction, respectively. The finite cache delay is the difference between the finite-cache performance and the infinite-cache performance and is .5 cycles per instruction for both models. The average miss delay (in terms of cycles per miss) can now be determined. From (4) in Sections 3 we see that the average miss delay can be expressed as

$$\left(\frac{\text{Cycles}}{\text{Miss}}\right) = \left(\frac{\text{Cycles}}{\text{Instruction}}\right)_{FC}\left(\frac{\text{Instructions}}{\text{Miss}}\right)$$

Substituting the values shown in Figure 8, we see that the average miss delay for Models 1 and 2 was approximately seven cycles. Two factors contribute to the average miss delay: the speed of the memory and the line putaway time (amount of time needed to copy a line of memory into the cache). The memory speed affects the average miss delay by delaying the original reference that caused the cache miss. This reference must now be satisfied from the memory which is several cycles slower than the cache. The line putaway time is a function of the linesize of the cache and bus width used to store lines into the cache. Any benefit derived from increasing the linesize to reduce the overall number of cache misses may be offset by increasing the number of cycles used to store the line into the cache.

The organization of the cache can also affect the processor's cycle time. It is commonly known that the miss ratio of the cache can be reduced by increasing the size of the cache. However, the extra time needed to access a larger cache may not be feasible considering the present cycle time and technology limitations. For example, an existing design may integrate the cache onto a single module and technology limitations may demand that a larger cache be spread across two modules. The extra path delays needed to access the cache across a second module may exceed the current cycle time. Now any benefits derived from reducing the miss ratio by increasing the cache size must be weighed against the performance loss from increasing the processor's cycle time. A designer would continue to pursue this design change only if it would lead to a more viable and powerful machine organization.

	Model 1	Model 2
Finite-Cache (Cyc/Inst)	2.0	2.4
$\left(\frac{\text{Cycles}}{\text{Instruction}}\right)_{FC}$	.5	.5
$\left(\frac{\text{Instructions}}{\text{Miss}}\right)$	14	14

Figure 8. Finite-Cache Performance

6. CONCLUSIONS

Timers have proven to be a very useful tool in analyzing the performance of a processor. Indeed, since design and engineering cost have risen so dramatically, the timer has become the most flexible and cost effective working prototype that allows a designer to study the performance of a new machine. We have divided the performance of a machine into three components: EBUSY, ENBUSY and finite cache effects. These components are used to describe the infinite-cache and finite-cache performance of a processor. By using a common set of trace tapes, a designer is able to compare the performance of different machine organizations within the scope of a single architecture.

The usefulness of a timer goes beyond that of a design tool. A timer serves as an excellent tool for researchers to explore the processing limits of a computer and its design. A timer allows a user to remove a delay entirely from an organization to investigate its performance implications. For example, a user can configure the timer to ignore all instruction interlocks or correctly predict the outcome of each branch. These runs establish a limit on the performance benefit that can be derived from implementing a mechanism that is intended to remove one of these delays.

Usually the performance gain derived from removing a pipeline delay is less than expected. As one delay is removed from the pipeline certain events and instruction sequences occur closer together. This can cause a new delay to appear that was not present in the original instruction sequence. A timer gives the researcher the ability to analyze these effects and accurately evaluate a mechanism that is intended to remove the delay.

REFERENCES

[*ANDE*67] Anderson D. W., F. J. Sparacio, and F. M. Tomasulo, 'The IBM System/360 Model 91 Machine Philosophy and Instruction-Handling', IBM Journal, Vol 11. Jan. 1967, pp. 8-24.

[*BELA*66] Belady, L., 'A Study of Replacement Algorithms for a Virtual-storage Computer', IBM Systems Journal, Vol. 5, No. 2, 1966, pp. 78 101.

[*CLAR*82] Clark D. W., and H. M. Levy, 'Measurements and Analysis of instruction use in the VAX 11/780', Proc. 9th Int. Symp. Comp. Arch., Apr. 1982, pp. 9-17.

[*EMMA*86] Emma P. G., J. W. Knight, J H. Pomerene, R. N. Rechtschaffen, and F. J. Sparacio, 'Components of Uniprocessor Performance', IBM Tech Report, RC 12203 (54836), Oct. 1986.

[*EMMA*87] Emma P. G., and E. S. Davidson, 'Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance', IEEE Trans. on Comp., Vol. C-36, No. 7, July 1987, pp. 859-875.

[*IBM*81] IBM System/370 Principles of Operation, GA22-7000-7, File No. S370-01.

[*LAMA*86] Lamaire O. R., and W. W. White, 'The Contribution to Performance of Instruction Set Usage in System/370', Fall Joint Computer Conference, Nov. 1986, pp. 665-674.

[*MACD*84] Macdougall M. H., 'Instruction-Level Program and Processor Modeling', Computer, July 1984, pp. 14-24.

[*MATT*70] Mattson, R., J., Gecsei, D. Slutz, and I. Traiger, 'Evaluation techniques for storage hierarchies', IBM Systems Journal, Vol. 9, No. 2, 1970, pp. 78-117.

[*POME*87] Pomerene J. H., T. R. Puzak, R. N. Rechtschaffen, F. J. Sparacio, Pageable Branch History Table, U. S. Patent 4,679,141, July 1987.

[*RYMA*82] Rymarczyk J. W., 'Coding Guidelines for Pipelined Processors', Proc. Symp. Architectural Support for Programming Languages and Operating Systems, 1982.

[*SMIT*82] Smith A. J., 'Cache Memories', Computing Surveys, Vol. 14, No. 3, Sept. 1982, pp. 473-530.

[*SUSS*71] Sussenguth E. H., 'Instruction Sequence Connection', U. S. Patent 3,559,183, 1971.

[*STON*87] Stone H. S., and D. F. Thiebaut, 'Footprints in the cache', ACM Trans. on Computer Systems, Vol. 5, No. 4, Nov. 1987, pp. 305-329.

[*TOMA*67] Tomasulo R. M., 'An efficient Algorithm for Exploiting Multiple Arithmetic Units', IBM Journal, Vol. 11, Jan 1967, pp. 25-33.

AUTHORS' BIBLIOGRAPHIES

All of the authors can be contacted at the following address:

IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598

PHILIP G. EMMA received his BS, MS, and Ph.D. degrees in Electrical Engineering from the University of Illinois. He currently works for the IBM Research Division in Yorktown Heights, N.Y. His area of specialization is high-end processor organization. He holds numerous patents in this area.

JOSHUA W. KNIGHT joined IBM in 1981 as a research staff member at the Thomas J. Watson Research Center. Since joining IBM, he has studied high performance uniprocessor machine organization, system support for CAD applications and software cache interactions. From 1978 to 1981 he was a Lecturer and Research Associate at Stanford University. He received a B.S. in Engineering Physics from Cornell University in 1968 and a Ph.D. in Applied Physics from Stanford University in 1978. Dr. Knight is a member of the ACM and IEEE.

JAMES H. POMERENE received his BS in Electrical Engineering from Northwestern University in 1942. He worked on radar at the Hazeltine Corporation until 1946 when he joined the Electronic Computer Project at the Institute for Advanced Study in Princeton at the invitation of John von Neumann and Herman Goldstine. He was made Chief Engineer of the project in 1951. In 1956 he came to IBM in Poughkeepsie and was responsible for the design of the Harvest System. In 1966 he led a small group investigating a highly available system that resulted in the cache memory in the IBM Model 85 processor. In 1967 he became a Senior Staff Member of the Corporate Technical Committee and was appointed an IBM Fellow in 1976. He has since pursued his Fellow program in Research, concentrating on improvements in high performance 370 processors. Mr. Pomerene is a member of the National Academy of Engineering and a Fellow of the Institute of Electrical and Electronic Engineers.

THOMAS R. PUZAK received a BS and MS in Mathematics and Computer Engineering from the University of Pittsburgh and a Ph.D. in Electrical and Computer Engineering from the University of Massachusetts. Since joining IBM in 1970 he has spent nearly the last ten years working at IBM Research as a Research Staff Member. His areas of interest include processor design concentrating in cache and pipeline performance. While at IBM he has received an Invention Achievement Award and holds several patents in these areas. Dr. Puzak is a member of the ACM and IEEE.

RUDOLPH N. RECHTSCHAFFEN received the B.A. degree (Summa Cum Laude) from Columbia College in 1958, and the M.S. and Ph.D. degrees in Mathematics from New York University, New York, N.Y. in 1968 and 1970, respectively. Prior to joining IBM in 1963, he held positions with Brookhaven National Laboratories, CEIR Inc., and MITRE Corporation. At IBM he has held positions in the marketing and the development divisions and since 1977 he has been a Research Staff Member at the IBM T.J. Watson Research Center. His research has concentrated on elements of processor design and analysis including cache performance and pipeline organizations.