

---

## CHAPTER 5

---

### *$\mu$ -Net*

#### **5.1. Preface to Chapters 5 and 6**

The following two chapters comprise a discussion of the application of Mirage to processor-memory interaction as a protocol. This, Chapter 5, is a description of the process of modeling processor-memory interaction as a protocol (Sections 5.2-5.3.), and an elaboration of the various degrees of implementation of the design that the process suggests (Sections 5.4.-5.5).

Chapter 6 compares the feasibility of the degrees of implementation, based on measured opcode statistics (Sections 6.1.-6.3). That chapter also contains the discussion of prior work specific to the processor-memory domain; more general prior work regarding Mirage is contained in Chapter 3, in Section 6.4. Chapter 6 concludes with an evaluation of the utility of this modeling process, and the value of the results it generated (Section 6.5).

A more concise version of the material in these two chapters, describing m-Net and its results, can be found by reading the following sections:

5.2. (inclusive)	Introduction
5.4. (inclusive)	$\mu$ -Net - Design
5.5.6.2.	The Total implementation of $\mu$ -Net
5.6-5.7. (inclusive)	Implications
6.1.-6.2. (inclusive)	Performance, Feasibility
6.3.2.	Other Observations
6.5. (inclusive)	Conclusions

As a preview,  $\mu$ -Net is a processor-memory interface design derived using Mirage as a model of the interaction as a protocol. The processor is augmented with a filtering device, to accept only messages (opcodes) with guards (addresses) that match the processor's state (program counter).

The memory is augmented with a mechanism, called a Code Pump, to model the state of the processor (PC) as a set of possible states, using a data structure (we call it the TreeStack). The mechanism sends data messages to the processor ([address, opcode] pairs) that increase the modeled state (i.e., send transformations), using a component called a Diverger. The mechanism receives data from the processor (PC values) that collapse the modeled state (i.e., receive transformation), using a component called a Converger.

This design is very similar to, and a superset of, current research trends in memory anticipation mechanisms, notable that of proactive memory. Performance increases are achieved by using idle bandwidth during round-trip latencies to send anticipatory information.  $\mu$ -Net sender anticipation as suggested by Mirage.

Measurements indicate that a simple, partial implementation would reduce opcode pipeline gaps and increase processor performance by a factor of 10; a complete implementation would increase performance by a factor of 300 to 3,000, when coupled with a cache. Feasible implementations require a simple stack mechanism and as little as 400 bytes of storage to achieve the same performance over high latencies as a 50 Kilobyte cache alone.

## 5.2. Introduction

Mirage provides methods for mitigating the effects of latency on communication, provided that the communication is well described by the protocol. In Mirage, the protocol doesn't merely police the information stream, it *is* the information stream. Evaluating Mirage requires the selection of a protocol in which shared state is the goal of the communication, rather than merely a means for connection management. In Chapter 4, NTP was used to describe the abstract components of the Mirage model in real terms, but many aspects of Mirage did not have analogs in NTP. For further analysis of the model, a new domain was chosen so as to exhibit some features of Mirage that existing protocols do not possess. We call the design that results from this analysis  *$\mu$ -Net* (pronounced 'MicroNet').

There are two common interpretations of a protocol - a mechanism to maintain shared state (i.e., the protocol *is* the communication), and a mechanism to manage communication (i.e., the protocol provides bit-transmission). Existing protocols manage only connection information as shared state; most of the data communicated is not part of this shared state. For the shared state to comprise the actual communication, a large state is not required; in NTP as few as 2 variables (current time and rate factor) are shared. In TCP a similarly small number of variables are shared<sup>1</sup>, but this state information is used to facilitate other data transmission. Most current protocols are not currently used to manage state spaces directly (as Mirage requires in order to show advantage).

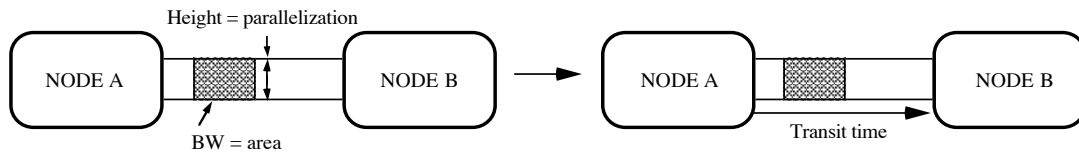
A more complete investigation of Mirage requires the selection of a domain in which communication protocols may not be normally applied, yet one in which state sharing is the goal of the communication. We have chosen a domain in which communication protocols are traditionally not used, that of processor-memory communication. Existing processor-memory communication architectures are shown equivalent to existing protocols, and application of the Mirage model yields novel communication architectures that improve performance where transmission latency is high.

As discussed in Chapters 1 and 2, Mirage adds a parameter of transit time to the conventional parameters of a communication channel, those of bandwidth and channel

---

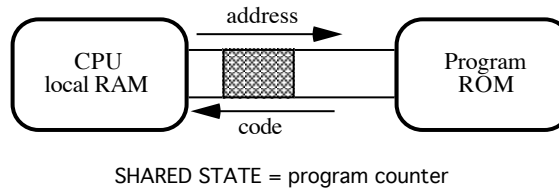
<sup>1</sup>TCP state information consists of the current connection information: the current window number, one of 11 connection states [Co91b], and endpoint address information.

width (parallelization) (Figure 5.1). Consider an analogous structure, in which the communicating nodes are that of processor/RAM and program memory (read-only) (Figure 5.2); this example is justified by its similarity to the client-server model of computation, in which code is stored on shared remote nodes, and utilized locally. This structure is a variant of the Harvard-style architecture, in which the data bus is local to the CPU and the code bus is a latency communication channel. The benefits of this architecture are not discussed here; it was chosen only as a domain in which to demonstrate the benefits of Mirage. The application of the Mirage model to this domain provides methods for reducing the performance effects of latency.



**FIGURE 5.1**

Mirage extends the channel model to include latency



**FIGURE 5.2**

Communication channel analog of processor/memory interaction

In this domain, performance is defined as the time of execution of a fixed code measure. In some cases reducing the execution time is cosmetically desirable, i.e., the computational outcome remains the same, but a speedup is desired; it is in this way that performance is usually considered. Execution time is as valid a measure of code as any other (i.e., correctness, completeness), and performance can be considered a correctness criterion, especially in time critical environments (see Chapter 3, on Real Time systems).

The characteristics of this architecture can be measured as the channel latency increases. The channel is a logical interaction between the processor/RAM and program memory, where the address of the desired code and the code itself are the communication exchanged across the interface.

### 5.2.1. The Domain - the processor / memory interface

A processor communicates with memory for two uses - retrieving data with which to operate and retrieving code to direct these operations. In the Harvard architecture, these two communication streams are both logically and physically independent<sup>1</sup>. The domain considered here is a version of this architecture, where the code stream communicates across a distance, but the data stream is local. This resembles a client-server model of computation, where the data (and perhaps I/O devices) are located at the CPU, but the code is archived at remote stations.

The advantages of this example are not of prime importance to us here, although they have been discussed elsewhere (as RPCs, or simply shared code on mounted disks in NFS). In this domain the shared state is concise and the temporal and communicative transformation functions on the state space are well understood. The shared state is the address whose code is being retrieved (i.e., the program counter), and the state space transformations are indicated in the opcode (e.g., jump, jump-to-subroutine), in some structure in the data space (return), or implied (the default for most opcodes). The nature of these transformation functions will be elaborated later; initially current architectures are described, together with their behavior as the code storage area is moved away from the CPU/data area.

The following discussion focuses only on communication between the CPU/data storage (noted as CPU) and the code storage area (noted as Memory). The effects on the communication of program code are measured as the program memory is moved away from the site of its utilization.

### 5.2.2. Description of current architectures

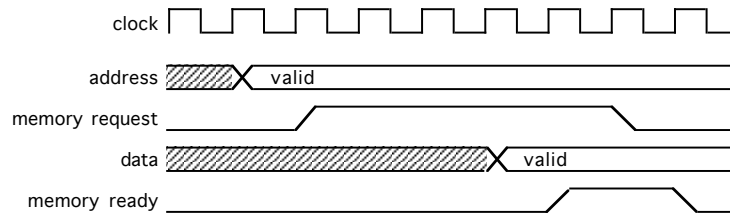
The processor/memory interface can be considered as a communication channel. In conventional computer designs, a processor communicates with program memory by either a request/response or a timed-response protocol, usually across a bus. The processor places the address of the desired code on a bus, and indicates a memory request by a pulse or level on a signal line (e.g., read/write, memory request, etc.). Memory

---

<sup>1</sup>Sometimes Harvard architectures only indicate separate data and code caches; here we intend that code and data are not only communicated separately, but also stored separately.

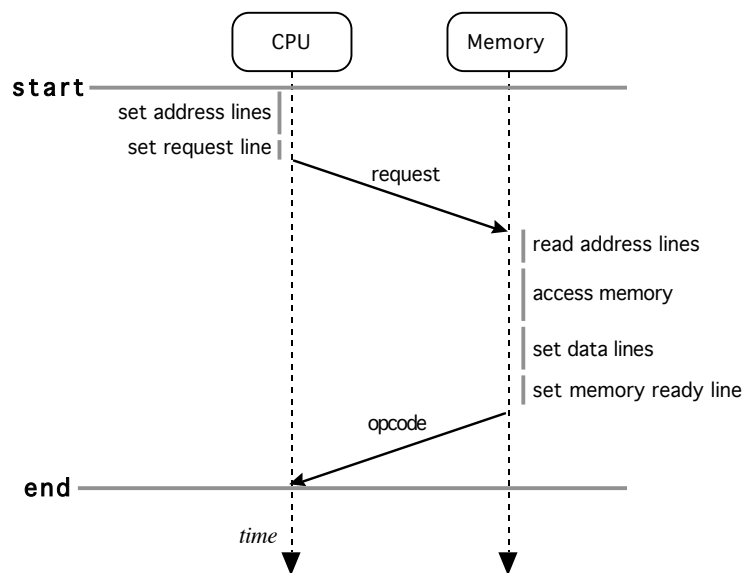
responds to this request by placing the opcode at the desired address on a bus, either setting a reply line (memory ready, etc.) or assuming that the processor will wait a fixed number of clock cycles for the reply. These two protocols are shown below (Figures 5.3, 5.4, 5.5, 5.6), both as conventionally shown as a bus interaction (time/wire label) (Figures 5.3, 5.5), and as a protocol time line (time/space) (Figures 5.4, 5.6). We denote the former protocol as *explicit*, and the latter as *timer-based*.

Explicit and timer-based protocols differ from the conventional designations of synchronous and asynchronous. Synchronous protocols ensure (and rely) on signals being aligned with the clock pulses, and asynchronous protocols permit clock signal independence. Timer-based protocols are synchronous, but explicit protocols can be either synchronous or asynchronous, depending on whether the reply signal must be aligned with the clock pulse.



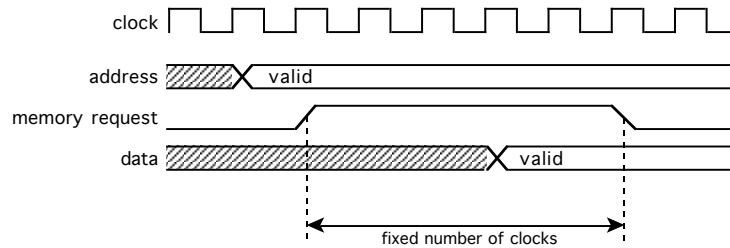
**FIGURE 5.3**

Explicit processor-memory protocol (voltage/time diagram)



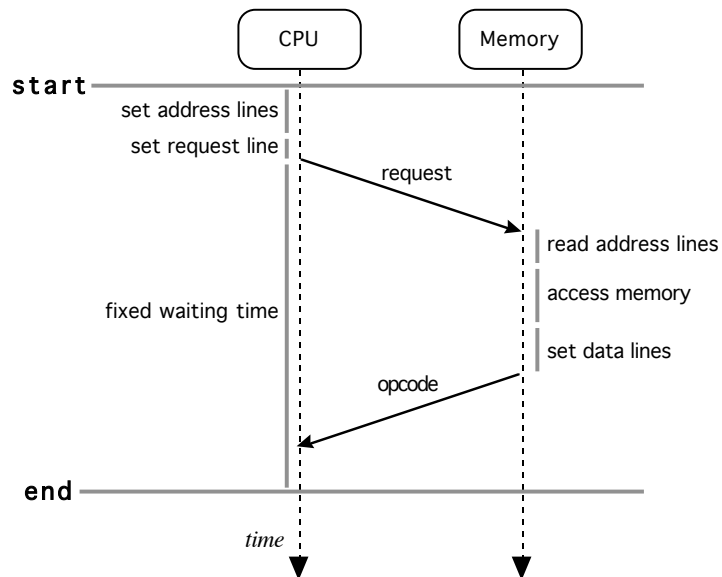
**FIGURE 5.4**

Explicit processor-memory protocol (as a protocol time line)



**FIGURE 5.5**

Timer based processor-memory protocol (voltage/time diagram)



**FIGURE 5.6**

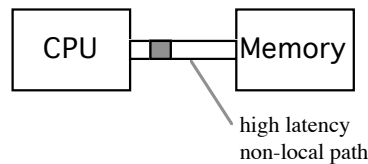
Timer-based processor-memory protocol (as a protocol time line)

There are usually few or no provisions for a failure of this request-response type of protocol; it is assumed that if this communication fails, little can be accomplished anyway (i.e., if an arbitrary opcode cannot be retrieved, neither can the interrupt handler code)<sup>1</sup>. If an explicit protocol is used, the processor would wait indefinitely; if a timer-

<sup>1</sup>In fault-tolerant systems, an external mechanism monitors operation and switches to a backup system or halts the current failure in a safe manner; systems do not perform fail-safe functions after their own (arbitrary) failure.

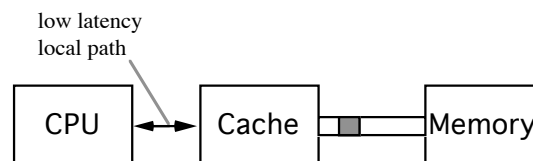
based protocol is used, the processor would read invalid data, because it would not otherwise know that the memory had failed to reply correctly.

Consider the characteristics of this communication as the processor and program memory are moved farther apart. In this description, *CPU* refers to the processor and local RAM memory (read/write data), and *Memory* refers to program memory (i.e., read only). The two components communicate by a high bandwidth path (Figure 5.7), whose latency increases as a parameter of this investigation.



**FIGURE 5.7**  
Processor-memory interaction across a distance

This configuration is usually augmented with a program cache (Figure 5.8). The cache is located with the processor and communicates to it via a low latency, high-bandwidth path. The CPU initiates a request for code to the cache, and the cache either replies directly (in the case of a cache hit) or forwards the request to the program memory over the high latency path. The memory reply is forwarded back to the processor and also copied into the cache for later reuse.



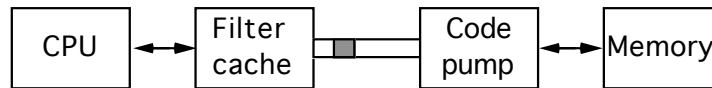
**FIGURE 5.8**  
Processor-memory interaction via a cache

There are extensions to this architecture which support prefetching, where the cache requests program code in anticipation of its use by the processor. The receiver of the data (the cache, ‘speaking’ for the processor) asks for the data before it is needed; this is called a *lookahead cache*.

The Mirage model, when applied to this domain, indicates a new way to design this architecture. In this new design, the sender (memory) anticipates the needs of the receiver (processor). This has been alluded to as *proactive memory*, although we refer to a specific

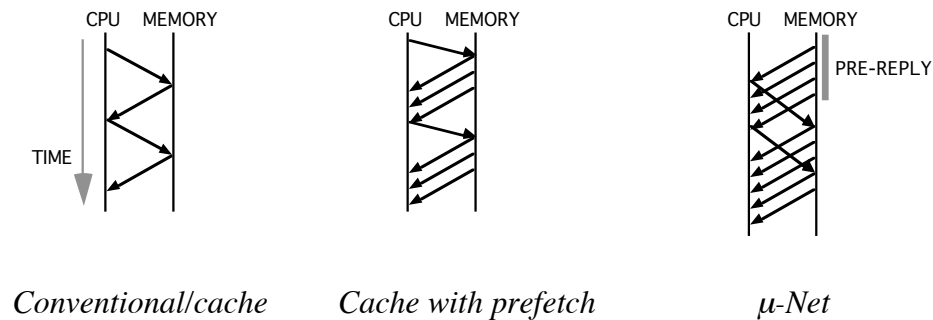
architecture here (i.e., suitable for implementation), whereas the prior work refers only to a “possible future direction” [Kr91].

In the new architecture, called  $\mu$ -Net [To91b], a *Code Pump* manages the anticipation at the site of the memory, and a *Filter Cache* emulates the function of a conventional cache, isolating the processor from the details of the mechanism (so that the system appears to the processor to be identical to Figure 5.9 (CPU-memory only), except in performance).



**FIGURE 5.9**  
 $\mu$ -Net processor-memory interaction

In terms of the earlier protocol figures (Figures 5.4, 5.6), the protocol timelines of the three architectures can be shown symbolically (Figure 5.10). The conventional and cache architectures exhibit the same protocols, except that the cache can omit communication when a hit occurs. A prefetching cache can retrieve multiple replies with a single request.  $\mu$ -Net permits memory to send replies before requests are received, i.e., the sender anticipates the receiver requests (the gray line in the  $\mu$ -Net version in Figure 5.10).



**FIGURE 5.10**  
Protocol timeline comparisons of processor-memory protocols

A conventional architecture with a cache incurs a round trip latency expense (miss penalty) whenever a miss occurs. A prefetching cache periodically requests blocks of code in advance, whereas  $\mu$ -Net is a self-adapting sender-based version of prefetching

where the memory sends code that the processor might need and concurrently receives updates of the processors actual state.

Cacheing is complementary both to prefetching and to anticipatory replies as implemented in  $\mu$ -Net. Cacheing reduces access bandwidth to memory. Prefetching increases memory use because fetched code may not be utilized;  $\mu$ -Net similarly increases memory use by sending sets of codes (isopotent sets), but only one member of each set is used. Cacheing is a way of looking into the past (of code use), making assumptions that code is reused in the future, to reduce memory bandwidth, whereas both prefetching and  $\mu$ -Net are ways of anticipating future requests (beyond just code reuse) which subsequently increase memory bandwidth.

### 5.2.3. Effect of latency on existing architectures

The effect of latency on these architectures can be considered by describing the time it takes to execute some number of instructions ( $N$ ). Assume that the processor executes 1 instruction per time unit ( $t$ ) (thus defined). A conventional architecture requires  $Nt$  time units to execute  $N$  instructions. This is the optimal case, with respect to the communication latency (i.e., latency is not considered, or zero) (Equation 5.1).

$$\text{Equation 5.1: } T_{optimal} = N * t$$

This formula is augmented to include latency (i.e., as latency increases beyond a negligible amount), in Equation 5.2, where  $r$  denotes the round trip latency. *Negligible latency* is defined as any time at least one order of magnitude smaller than the execution time of a single instruction, in which case Equation 5.2 reduces to Equation 5.1 (i.e., latency contributions are negligible in comparison to instruction execution).

$$\text{Equation 5.2: } T_{conventional} = N * (t + r)$$

Performance is described as the time required to execute  $N$  instructions vs. the time required in the optimal case. The ratio of particular execution time to optimal time is defines the *slowdown* (Equation 5.3). This ration in conventional architecture analysis is usually called the *speedup*, but we consider cases where the instances examined are slower than optimal. In the case of a conventional architecture, this is Equation 5.4.

$$\text{Equation 5.3: } Slowdown = \frac{T_{measured}}{T_{optimal}}$$

$$\text{Equation 5.4: } \text{Slowdown}_{\text{conventional}} = \frac{T_{\text{conventional}}}{T_{\text{optimal}}} = 1 + \frac{r}{t}$$

In the case of the conventional architecture augmented by a cache, the effects of the latency are reduced proportional to the effectiveness of the cache utilization (Equation 5.5).  $M$  denotes the probability of a miss in the cache. On a cache miss, both the round trip and the execution time are incurred. A cache hit costs only one instruction execution time. Equation 5.6 reduces to Equation 5.4 where the cache is 100% effective. In this case, no communication costs are incurred, because all code accesses are intercepted by the (local) cache.

$$\text{Equation 5.5: } T_{\text{cache}} = N * (t + r * M)$$

$$\text{Equation 5.6: } \text{Slowdown}_{\text{cache}} = 1 + M * \frac{r}{t}$$

The time to execute  $N$  instructions in an architecture where the cache prefetches can also be described. Let  $k$  denote the prefetch length, in instructions. Further characterization of the communication stream will be required to estimate the probability of a prefetch occurring (described later), as used in the equation.

Prefetching is usually implemented in a linear fashion only. When an opcode at address  $i$  is accessed, address  $i+1$  through  $i+k$  are prefetched, where  $k$  is the linear lookahead, and usually  $k$  is the number of opcode words in a cache line<sup>1</sup>. A miss occurs when the prefetch fails to anticipate the next opcode used, i.e., when the linear anticipation assumed by most implementations is breached. Misses also occur when the destination of a control transfer instruction (e.g., BRANCH, JUMP, CALL, RETURN) is not in the cache.

At most, prefetches occur at the miss rate of the cache alone (i.e., the same as without prefetching). The actual prefetch miss rate is also at most the rate of occurrence of control instructions, because the opcode after a control transfer may not be in the cache (Equation 5.8).

Control instructions are assumed always to prevent the prefetch of the next opcode. This assumption is valid for JUMP, CALL, and RETURN opcodes, but it is not always valid for BRANCHES. In the case where a BRANCH is not taken, the next opcode is

---

<sup>1</sup>In an Intel 80386,  $k=4$ ; in most caches the line size is between 4 and 8.

successfully anticipated. Because branches are taken about 50% of the time [He90], the formula can be further refined (Equation 5.9).

For the moment, the probability of a fetch occurring is denoted as  $F$  (Equations 5.7, 5.10).  $F$  is always less than  $M$ , because control instructions are only part of the cause of misses in a conventional cache, so Equation 5.7 is always at least as good as Equation 5.5. Empirical values of  $F$  and  $M$  are discussed in Chapter 6.

$$\text{Equation 5.7: } T_{prefetch} = N(t + r * F)$$

$$\text{Equation 5.8: } F \leq J + B + C + R + I$$

where opcode occurrences are denoted by percentages:

$J = \% \text{ JUMPS (direct only)}$

$B = \% \text{ BRANCHES (direct only)}$

$C = \% \text{ CALLS (direct only)}$

$R = \% \text{ RETURNS}^1$

$I = \% \text{ other indirect opcodes (JUMPS, CALLS, BRANCHES)}$

$$\text{Equation 5.9: } F \leq J + \frac{B}{2} + C + R + I$$

$$\text{Equation 5.10: } Slowdown_{prefetch} = 1 + F * \frac{r}{t}$$

In μ-Net, the equations are a little more complicated. The time to execute the  $N$  instructions depends on the ability of the *Code Pump* to predict correctly the opcodes desired by the processor. The probability of an incorrect prediction is denoted as  $P$ , and one round trip time will be required for each misprediction, to allow the processor to fetch the desired data which is not already available (Equations 5.11, 5.12). This case reduces to the optimal (Equation 5.1) where  $P = 0$ , i.e., when the prediction is perfect. An estimate of  $P$  will be discussed later, with the conditions under which it is smaller than  $F$  and  $M$ .

$$\text{Equation 5.11: } T_{\mu Net} = N * (t + r * P)$$

---

<sup>1</sup>The percent of CALLS and RETURNS are not always equal. Some systems permit direct stack manipulations, or returns which pop out of multiple nesting levels.

**Equation 5.12:**  $Slowdown_{\mu Net} = 1 + P * \frac{r}{t}$

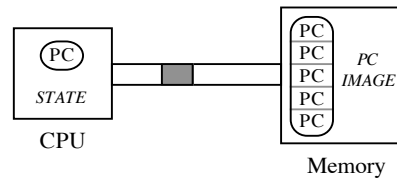
### 5.3. $\mu$ -Net

$\mu$ -Net [To91b] refers to our choice of the processor-memory interaction as being analyzed by a communication model, Mirage. This is in keeping with the tradition that all protocol research, at one time, must coin a network name suffixed in ‘Net’. This was not done with the model name (MirageNet?), so the tradition is upheld in the name of the example.

The  $\mu$ -Net domain is easily modeled in Mirage, by suitably defining the shared state space and by defining the state transformations of Mirage in terms of this space. In this example, communication is affected by *classes* of opcodes in the instruction stream. The usual techniques of cache prefetching, widening cache line sizes, and branch prediction are confirmed from the Mirage model of  $\mu$ -Net. Further,  $\mu$ -Net exhibits new techniques for active opcode anticipation, which are only recently being proposed as new solutions to latency issues in architectural design. Also, some simple data structures in the memory interface can reduce the latency for some kinds of memory access operations; these structures implement isopotent anticipation as described in Mirage.

In order to consider the Mirage model of  $\mu$ -Net, the components of  $\mu$ -Net need to be defined in terms of the model components in Mirage. The first and most fundamental of these is the shared state between the sender and receiver.

In the processor/memory interface, what state is shared? Usually, this is minimally the program counter (PC) value. The program counter denotes the address of the next instruction desired, to be placed on the address lines when a memory request is made. During the memory request, that address is received by memory, after which the desired opcode is replied. The state space is thus the space of all PC values. Conventional protocols communicate by understanding the state of the remote node as a point in this space, and moving this point explicitly. Therefore, until the memory receives the new value of the program counter, the previous value remains in effect, as far as the memory is concerned (Figure 5.11).



**FIGURE 5.11**  
CPU state and Memory image

In terms of the Mirage model, the state is a PC value, and an image is a set of PC values, i.e., a subset of the entire PC space. Transformations map PC values onto new PC values, or sets onto sets.

The value of the state space changes with the progression of time, the sending of a message, and the receipt of a message. The state space, in this case, is the program counter, because the memory models the processor's current PC value.

This analysis considers the memory's model of the processor, not the processor's model of the memory. This is appropriate because the communication is essentially unidirectional, although there are components in both directions (control in one direction vs. data in the other).

### 5.3.1. Time transformations

As time progresses, two things happen at the processor. The processor is executing the current instruction, then it must wait for opcodes to be sent from memory, sitting idle until its request for the subsequent opcode is serviced. As a result, the state of the processor (i.e., the PC, the part of the state that the memory is concerned with) is stable during the execution of the instruction, and changes only afterwards.

This assumes, however, that there is no cache at the processor. When there is, the processor may use opcodes already in the cache during the time lapse. The opcodes in the cache have already been sent from memory, so they are already accounted for in the memory's image of the processor's PC.

The passage of time then, in this domain, *denotes the scheduling of the need for opcodes by the processor*. During the time in which the current opcode is executing, the PC image does not change. At the time when the execution changes, the PC image changes to reflect the need for the next opcode.

The simplified model assumes a RISC architecture with a single opcode execution time. Variability in execution times (e.g., in CISC architectures) can be accommodated

by tables in the sender when opcode execution times are static, or by a dynamic structure, if execution times vary, as in pipelining. This latter dynamic structure emulates the pipeline activity, in order to predict the need for new opcodes at the pipeline input.

### 5.3.2. Receive transformations

When messages are received, the state space collapses accordingly. There are cases (in Table 5.1) in which the state space of the program memory can grow, by modeling both arms of a conditional, or by expanding to the limit of the state space (in the case of indirect opcodes). The messages received by the program memory are processor PC values, which collapse the PC value set in the program memory to a single value. In fact, in this case, the PC set is collapsed to the value received, and then expanded to account for the transit time of the message. In this way, the PC set in the program memory always models the PC of the processor, out of sync by the transit time of a message.

### 5.3.3. Send transformations

When information is sent, the state space expands through the unioning of the sets of state space before the message and that state space as affected by the message. The state space consists only of PC values (and possible PC values), so the actual opcodes sent are ignored, except in the way in which they affect the current PC. The memory has some set of possible PC values; it sends the opcodes as these PCs indicate (i.e., it sends the opcodes at those addresses), and transforms each PC by a function indicated by the opcode. After transmission, the new PC set becomes the union of the prior set and the transformed set. For example, if the opcode is ordinary (OTHER, i.e., not otherwise distinguished hereafter), the PC would increment when the opcode is sent. The resulting PC set is the union of the PC and PC+1. The transformation depends on the opcode sent; some opcodes increment the current PC, some transform it, some expand it to a set of two PCs (i.e., two possible PCs), and some expand the PC to the set of all PCs (Table 5.1).

In the most general case, the PC would be arbitrarily transformed with each opcode, but this is not effective in modeling the evolution of the imprecision of the knowledge in the PC as known by the program memory. Because the program memory knows the contents of the message (i.e., the opcode), it can determine how the message will potentially affect the PC at the processor (i.e., the transformation).

The image of the PC in the program memory is transformed differently for various classes of opcodes sent. The opcodes are distinguished only by the way in which they transform the current PC over time (Table 5.1).

OPCODE	transform: PC $\rightarrow$	new PC values are computed based on...
other <sup>1</sup>	PC+1	PC
direct jump	PC'	PC, opcode
indirect jump	{all PCs}	PC, opcode, any CPU register, program or data memory value
direct call	PC'	PC, opcode (prior PC is stored in a known structure in RAM )
indirect call	{all PCs}	PC, opcode, any CPU register, program or data memory value (prior PC is stored in a known structure in RAM )
direct branch	{PC+1,PC'}	PC, opcode
indirect branch	{PC+1,PC'}	PC, opcode, any CPU register, program or data memory value
return	PC'	stored in a known structure in RAM (previously saved)

**TABLE 5.1**  
Opcode time transformations

For the majority of the opcodes (denoted as OTHER), the PC is incremented. In the case of a JUMP, the program counter becomes a new value based on the destination. In the case of a BRANCH, the program counter becomes one of two new possible program counter values, one being the incremented PC, the other being the newly indicated branch destination, thus the model of the PC becomes a set of PCs. This is the expansion of the state space referred to in the Mirage model.

At some later time, at the receipt of a message from the processor, this set is collapsed down to a single member. CALLs behave similarly to JUMPs, except that in addition to performing the transformation indicated, some local state is maintained (in a

---

<sup>1</sup>I.e., not otherwise listed in this table.

stack). A RETURN utilizes this local state to perform a JUMP back to the opcode following the origin of the corresponding CALL.

The table distinguishes between direct and indirect types of jumps, branches, and calls, because in the former case the resultant PC can be computed, whereas in the latter the state space becomes completely unpredictable<sup>1</sup> (i.e., it expands completely to encompass the entire state space). Opcodes differ in the way in which they affect the expansion of the state space over time; this is the criterion for partitioning them as described.

In the case where the PC set expands to the limits of the state space (for indirect opcodes), further temporal transformations become impossible to compute. The memory must wait for a message from the processor of the actual PC value chosen, before it can proceed further. Indirect opcodes necessarily cause ‘bubbles in the communication loop’, where sender anticipation cannot be accommodated. Indirect opcodes, are, in effect, too unpredictable to model.

#### 5.3.4. Guarded messages

In the conventional processor/memory architecture, there is no need to label the opcodes which are sent from memory, because only one memory request is outstanding per unit time. Even in the cache prefetch case, the cache either requests each memory value independently, or an initial request is made and the resulting values are assumed to arrive sequentially.

The opcodes being sent from memory must be labelled, in order to permit the processor to receive them conditionally. Consider the case where the PC modeled in memory contains a BRANCH instruction. Memory sends the next instructions, but more than one instruction is sent (i.e., more than one possible next request, as discussed before). Replies to these requests must be differentiated, so that the processor (which knows its own state) can select the appropriate one. Although there are more efficient labellings, the opcode can be labelled with the part of the state space to which it applies

---

<sup>1</sup>Indirect jumps can be limited to the set of labels in the program source code, assuming that the compiler ensures such restrictions, and that the executable code contains label information. This will be discussed in the prior work of  $\mu$ -Net, in Chapter 6.

(i.e., with the PC it is located at). A single bit could be used to indicate the two arms of a branch, but any label thus chosen is limited to some finite partitioning of the state space (i.e., only 1 branch lookahead per bit).

### 5.3.5. Partitioning the state space (stability)

The state space is not partitioned into sets of PC values, which would be computationally prohibitive. The PC image may indicate two PC values which request the same opcode, which could be sent only once with a guard indicating both PCs. Such an implementation would be excessive, because the resulting reduction in bandwidth would be at the expense of excessive encoding.

Instead, the state space is partitioned into its individual points (single PC values), so each message (opcode) requires a single PC value as a guard. The state space is not separated into “last timestep / this timestep”, as in the union of the two spaces in the Mirage send transformations.  $\mu$ -Net assumes that messages are not lost, so the prior state (before the sent message) need not be retained. This prevents having to resend opcodes, under the assumption that the communication channel is lossless. If messages always arrive, in some fixed time delay, the part of the state space which corresponds to the prior state can be ignored because the probability density highly favors the send-transformed state.

### 5.3.6. Isopotent sets

There are two analogs of isopotent sets in this domain. In the first, isopotency occurs when two PC image values recombine, i.e., when instruction streams reconverge. In the second, isopotency indicates the satisfaction of both paths of a conditional by the set of both destination opcodes. This latter isopotency is accommodated in  $\mu$ -Net by the expansion of the state space when a BRANCH occurs, because subsequent message sets furnish both arms of the branch.

The reconvergence case occurs either during a single timestep (simultaneously), or at differing timesteps. For example, when two current PC image values JUMP to the same location, the streams converge simultaneously. This convergence is managed as a side-effect of our implementation; if two PCs in the image at time  $t$  become the same at time  $t+1$ , only one remains.

When a forward branch occurs, the streams converge at different time values, because the PC of the branch-taken stream is the same as the PC of the branch-not-taken stream at some earlier time. This is not handled in  $\mu$ -Net. In this latter case, isotopotency also exists where two PCs in the image become one, but only under message sequences of different length. If information were maintained on the past state in the model (it isn't - see the discussion on Partitioning, above), this recombination would be recognizable.  $\mu$ -Net cannot take advantage of the case where one stream becomes a time-shifted image of another stream. This inability is not corrected because the complexity in modeling past time images would be prohibitive, and a corresponding benefit may not exist.

As a result, the way in which a set is isotopotent depends on the ways in which the members of the transformed set are considered equivalent, i.e., the way in which the messages are equivalent.

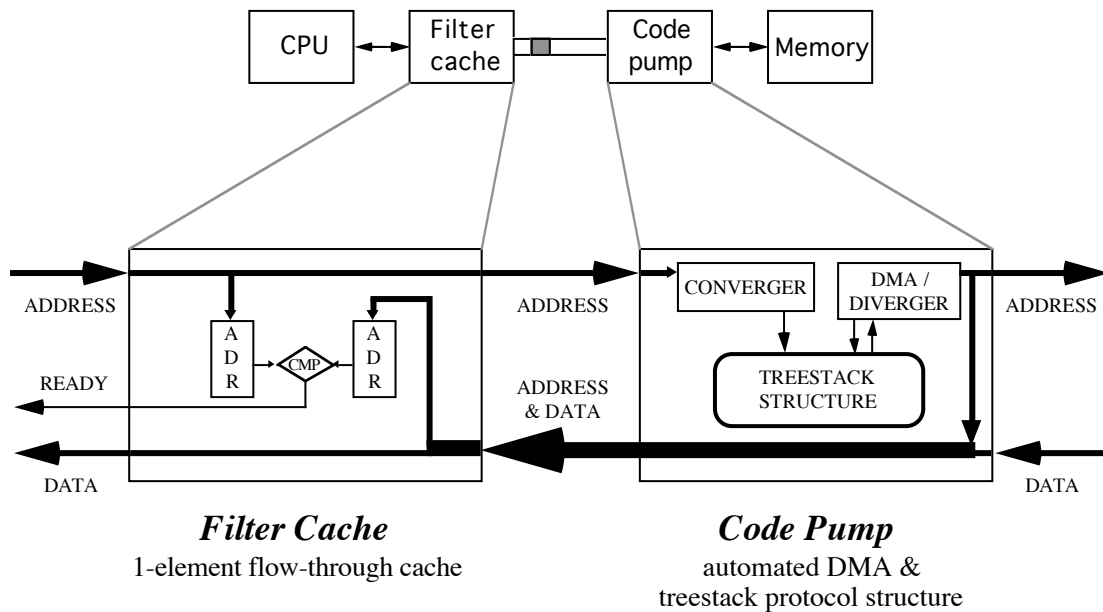
## 5.4. $\mu$ -Net - Design

Although Mirage is an abstract model whose direct implementation was argued against,  $\mu$ -Net can be implemented directly. Models are not usually intended to be implemented, but the domain of this problem has been sufficiently constrained to permit such an implementation to be reasonable, both to give a real impetus to the abstract model and to permit a better understanding of the model's advantages.

Communication in  $\mu$ -Net is unidirectional - although the processor needs to communicate its state to the memory to specify the desired opcode location, the processor is modeled in the memory, and not the converse. Only the memory is concerned with the state of the remote entity (processor) here. The model of the processor can be implemented by putting a PC in the memory. In fact, a structure is placed near memory which will permit the modeling of more than one possible PC.

### 5.4.1. The Code Pump

The *Code Pump* manages the image of the processor for the memory. It also contains the send and receive mechanisms, insofar as they affect the state space modeled within the Code Pump. The Code Pump implements the time, receive, and send state space transformations, and the state space image (albeit limited) (see detail, Figure 5.12).

**FIGURE 5.12**Detail of Filter Cache and Code Pump of  $\mu$ -Net

The Code Pump contains three main components: a TreeStack data structure, a Converger, and a DMA/Diverger. The TreeStack structure maintains the image of the processor's PC; its tree-like attribute manages bifurcations in the PC image caused by sending BRANCH opcodes, and its stack-like attribute manages the information of saved and restored PC images which are caused by CALLs and RETURNS, respectively.

The Converger manages the collapse of the PC image in the TreeStack structure, as indicated by the receipt of messages from the processor (i.e., addresses). The DMA/Diverger manages the preparation and sending of messages anticipating the processor's request (DMA), and the expansion of the PC image in the TreeStack, as indicated by the send transformations.

### 5.4.2. The Filter Cache

The *Filter Cache* serves only to make the *Code Pump* mechanism appear invisible. It buffers data coming in from memory, and passes through only opcodes whose label (PC value, i.e., address) match the requested address whose data the processor is waiting for. All other data, for addresses not requested, is thrown away. It thus implements the receiver requirements for guarded message use.

The Filter Cache needs only a small amount of space to operate. If the rate of the processor is known (i.e., the rate at which opcode requests are issued, which is nearly trivial in a RISC processor), the Filter Cache needs only one opcode/address unit of space. The opcode arrives with the address as a conditional label (guard), so that the opcode is used only if the address is pending a request, which is the purpose of the comparator in the filter. The Code Pump knows this rate, so it sends the messages only when needed (see the time transformation).

The Filter Cache is also completely compatible with a conventional cache in parallel at the same location. The Filter Cache's purpose is to manage the incoming stream of new information, and pass the relevant parts on to the processor, whereas the conventional cache passes old information back to the processor. The two caches are complementary.

### 5.4.3. Degrees of design

The model is implemented directly in  $\mu$ -Net, so there are some variations to the level of implementation which can be performed. For example, a TreeStack structure cannot be effectively implemented which models the PC after an indirect opcode, because the state space grows to its limits, and further partitioning of the state space to predict the next desired opcode is impractical. The design can be further simplified by not implementing any branch transformations, or not implementing RETURNS. The following is an enumeration of the various levels of implementation, in increasing order of complexity (Table 5.2).

In the Code Pump implementation, the effects of the sent message on the PC image must be modeled, as well as modeling the old image (i.e., unioning of the unaffected and affected PCs). One simplifying implementation decision is to ignore the possibility of message loss, and omit the old PC image, replacing it with the new one as soon as the message (i.e., opcode) is sent. This works under the assumption that messages are not lost, simply delayed by a fixed time, in a way that simplifies the design to account for a reduction in the probability of states in the old image remaining valid.

The Code Pump can be implemented as a null device, which reduces to a conventional or conventional plus cache architecture. Prediction can be limited to only OTHER opcodes, where the Code Pump requires only a single PC image and an incrementer. This is analogous to an opcode prefetch in current CPUs (680x0, 80x86,

MIPS, SPARC, etc.), but differs in that the PC and incrementer are placed in the memory, rather than in part of the processor (i.e., the PC is on both sides of the interface), thus reducing the comparable latency by half. At this point, the Filter Cache needs only 1 element of space to operate - to hold the opcode/address pair as it is received, and to compare it to the current desired address.

Level of opcodes done	Filter size	Code Pump Components	Storage required (in Code Pump, in PCs) <sup>1</sup>
(none)	(null)	(null)	(none)
other	1 element	increment	1
other, jump	(same)	adder	1
other, jump, call	(same)	(same)	1
other, jump, call, return	(same)	adder, stack	avg. pending depth
all but indirect <sup>2</sup>	2 elements	adder, TreeStack	$L * \log_2 \left( \frac{r}{L * 2 + 1} \right)$

**TABLE 5.2**

Degrees of implementation, and the implications of each

When the Code Pump is augmented to handle JUMP opcode messages, the incrementer is converted to an adder, to accommodate the ways in which JUMP opcodes alter the PC - by direct overwrite, or by PC-relative addressing (i.e., add a constant to the current PC). CALLs can be similarly accommodated with no increase in complexity, assuming we do not store the origin of the CALL opcode.

<sup>1</sup> $L$ =limb length,  $r$ =round trip time.

<sup>2</sup>The extra bits in the Filter Cache encode the choices made at each branch pending in the round trip communication, and are used only if the guards on incoming messages are so labelled. If the incoming guards are labelled with complete addresses, this storage can be omitted.

In order to accommodate RETURN opcodes, a more complex structure in the Code Pump is required to image the PCs in the processor. When a CALL is encountered, the Code Pump must hold the current PC in storage, so that when the corresponding RETURN occurs, the destination address implied can be determined. The state space image models the way in which a CALL causes the current PC state space location to be moved to a new point, for later return. This is a recursion in state space, such that a CALL is an entry point into a fresh copy of the state space, and a RETURN is the lone exit point from this state space, back to the original. Recursive state space manipulation was not envisaged when Mirage was developed, but it appears a natural extension of the application of the model.

The natural data structure for maintaining this recursive space image of the PC space is a stack.  $\mu$ -Net proposes to put a stack on the memory side of the interface, to permit the memory to model the PC transformation of a RETURN opcode message. The memory can then proceed to subsequent messages (following the RETURN, in logical sequence), rather than being required to wait for an explicit request from the processor.

There are only two other classifications of opcodes, whose message transformations have not yet been considered: INDIRECT (e.g., indirect branch, indirect jump, and indirect call), and direct branches. INDIRECT opcodes transform the existing PC image to one which encompasses the entire PC space, as noted before. The result of this transformation is to prevent subsequent partitioning of the state space, to permit messages to be determined. If an indirect opcode can indicate a jump to any PC, there is no way to predict the next PC to send without further assumptions. Memory is forced to wait for an explicit request from the processor, because only the receipt of a request message will cause the state space to collapse (to the PC indicated by the request).

The assumptions under which an INDIRECT opcode may be predictable are those which restrict the definition of those opcodes. Analysis of source code, or suitably supplemented object code, can indicate a list of possible indirect jump, call, or branch locations. If the compiler restricts indirect opcodes to jump to computed values which are members of this list, then the opcode could be predicted. Indirect opcodes so restricted are equivalent to a fixed dispatch handler with a passed offset argument. In the handler, the passed argument denotes the requested jump destination. This mechanism requires compiler participation in the restriction of the action of indirect opcodes.  $\mu$ -Net makes no assumption about cooperation of the compiler, and is intended to be compatible with

existing object code, which is not subject to INDIRECT jump restrictions. True INDIRECT opcodes do not limit the control flow to known compiler labels.

The forced wait for communication from the processor upon execution of an indirect opcode is sensible, because the new PC value depends on information which the memory does not have, such as values in RAM or processor registers. Indirect opcodes provide a transformation of the state space which is too powerful to model. This can be used as an argument for a more restricted form of indirect branching, such as a multiway table jump, especially because indirect opcodes are used mainly for such table jumps anyway. Any form of indirect opcode which permits the PC space to expand in a finite way, but not to its complete limits, would be able to be accommodated by the Code Pump.

One such version of a limited indirection is a BRANCH, in which only two resultant PC values are permitted (one is always PC+1, and the other is specified in the opcode by either a PC offset or a new PC value). When a BRANCH occurs at a single PC value, the transformation becomes a set of two PCs. The Code Pump requires a much more complex, but realizable, structure to implement the image of PCs under BRANCH message transformations. We call this structure a TreeStack.

## 5.5. Elaboration of degrees of design

These variations on implementation require various degrees of complexity in design of the components of  $\mu$ -Net. All involve the maintenance of data structures in the Code Pump, where the Diverger manages creation and extension of the data structure, and the Converger manages reduction. The Filter Cache also varies in design, although only minimally so, in comparison to the Code Pump. Here we elaborate on the previous descriptions, and provide implementation designs.

These designs are not intended to be computationally optimal. Delays caused by the complexity of an implementation would further limit potential anticipation. Only the combined recursion and branching anticipation design is susceptible to such complexity; all other designs can be implemented as effectively as existing CPU/memory components.

### 5.5.1. No opcodes anticipated (Null implementation)

The null implementation uses a Null Filter Cache (i.e., no Filter Cache), and a nearly-null Code Pump. All memory requests from the CPU are forwarded to the Code Pump, where they are latched onto the opcode memory address port (as in a conventional design). Opcodes in reply are sent back to the CPU. The Filter Cache is non-existent, and the Code Pump is a memory address latch. Tables 5.3 and 5.4 describe the Null Filter Cache, and Tables 5.5 and 5.6 describe the actions of the Null Code Pump components (*Converger* and *Diverger*, respectively). Figure 5.14 denotes the data structure maintained. Figure 5.13 denotes the Null design of the Filter Cache.

The Null  $\mu$ -Net implements the point model of communication. The Code Pump models the CPU as a point in state space, i.e., the last address requested is stored in a latch (PCvalue), representing the last known PC state of the CPU. The latch value is updated only after the current opcode returns to the CPU, is executed, and a new address (PCvalue) is communicated to the Code Pump; the alternation of control between the CPU execution and the Code Pump latching is denoted by the use of the signal variable (Flag).

Current CPU opcode	Action
any opcode	send current PC to Code Pump Converger

**TABLE 5.3**  
Null Filter send actions

Message received	Action
any opcode	send opcode to CPU

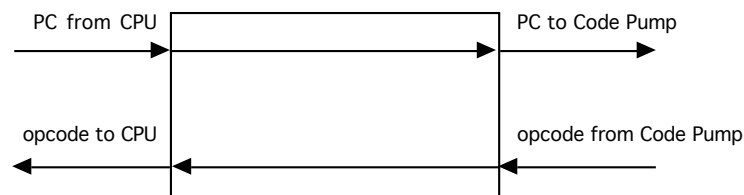
**TABLE 5.4**  
Null Filter receive actions

Message received	Action
any PC	overwrite PCvalue set Flag

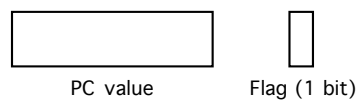
**TABLE 5.5**  
Null Converger actions

Data area item	Action
PCvalue & Flag set	fetch opcode at PCvalue send opcode to Filter Cache set Flag

**TABLE 5.6**  
Null Diverger actions



**FIGURE 5.13**  
Null Filter Cache design



**FIGURE 5.14**  
Null data space

### 5.5.2. Unit Linear opcodes anticipated

Unit Linear anticipation extends the Null design to accommodate the anticipation of regular (OTHER) opcodes, i.e., those opcodes which transform the PC by a unit addition

(i.e.,  $PC \leftarrow PC + 1$ ). The Filter Cache can be more selective in sending PC values to the Code Pump, because when ‘other’ opcodes are encountered, no PC need be sent.

In the same way as the Null implementation, the Flag variable indicates whether the anticipation may proceed or when it must cease and await resynchronization. The latter occurs whenever the Code Pump Diverger encounters an opcode whose future path cannot be determined.

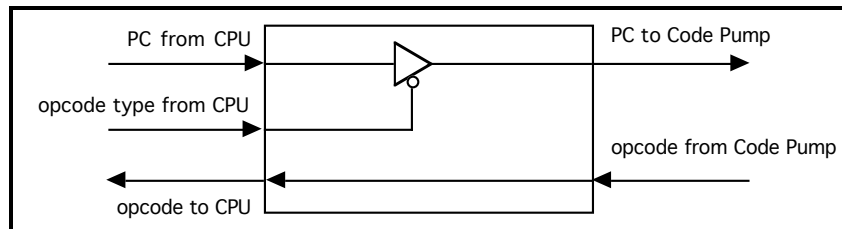
The Unit Linear Filter Cache has a new set of send actions (Table 5.7), but the same receive actions as the Null Filter Cache (Table 5.4). The Unit Linear Converger is identical to the Null Converger (Table 5.5), but the Unit Linear Diverger is augmented to follow the PC path during pumping of opcodes (Table 5.8). Figure 5.14 denotes the data structure maintained, as before in the Null implementation, because no other data is required. Figure 5.15 shows the modified design of the Unit Linear Filter Cache, to accommodate the required opcode type information.

Current CPU opcode	Action
‘other’ opcodes	{ no action }
jump, call, return, branch, or indirect	send current PC to Code Pump Converger

**TABLE 5.7**  
Unit Linear Filter send actions

Data area item	Action
PCvalue & Flag set	fetch opcode at PCvalue send opcode to Filter Cache Type of opcode: ‘other’: PC $\leftarrow PC+1$ jump, call, return, branch, or indirect: reset Flag

**TABLE 5.8**  
Unit Linear Diverger actions



**FIGURE 5.15**  
Unit Linear Filter Cache design

### 5.5.3. Linear opcodes anticipated

Linear opcodes transform the state space by some constant amount, not just '1' as in the Unit Linear case. These include JUMP and CALL opcodes. Linear anticipation can be accommodated by a minimal modification of the unit linear anticipation design. The Linear Filter Cache receive and Linear Code Pump Converter remain unchanged, and are still the same as in the Null implementation. Filter Cache send data changes only in which opcodes activate message emission (Table 5.9).

Similarly, the Diverger is extended to add arbitrary fixed offsets, as extracted from within the JUMP and CALL opcodes (Table 5.10). The data space remains unchanged.

Current CPU opcode	Action
'other', jump, call	{ no action }
return, branch, or indirect	send current PC to Code Pump Converter

**TABLE 5.9**  
Linear Filter send actions

Data area item	Action
PCvalue & Flag set	fetch opcode at PCvalue send opcode to Filter Cache Type of opcode: ‘other’: $PC \leftarrow PC+1$ jump, call: get ‘k’ from opcode $PC \leftarrow PC+k$ return, branch, or indirect: reset Flag

**TABLE 5.10**  
Linear Diverger actions

#### 5.5.4. Recursion and Linear anticipation

Extending the Linear  $\mu$ -Net design to accommodate recursion (RETURN opcodes) involves extending the data structure to model a kind of ‘space embedding.’ The data structure is a simple stack of PC values, with the same (single) signal Flag as before (Figure 5.16). The Recursion Filter Cache receive mechanism remains unchanged (same as in the null design), and the send mechanism is changed to omit messaging upon ‘return’ opcodes (Table 5.11). The Recursion Converger remains unchanged from the Null design.

The Recursion Diverger includes not only the linear transformation components of the linear anticipation design, but also adds the ‘push’ and ‘pop’ operations on the data structure to model the state space embedding (Table 5.12).

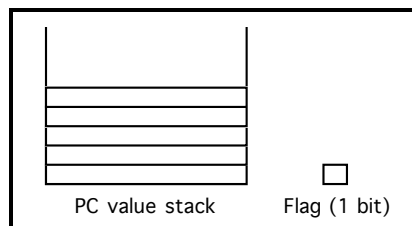
Current CPU opcode	Action
‘other’, jump, call, return	{ no action }
branch or indirect	send current PC to Code Pump Converger

**TABLE 5.11**

## Recursion Filter send actions

Data area item	Action
PCvalue & Flag set	fetch opcode at PCvalue on stack top send opcode to Filter Cache Type of opcode: 'other': $PC \leftarrow PC+1$ jump: get 'k' from opcode send opcode to Filter Cache $PC \leftarrow PC+k$ call: push PC+1 onto stack get 'k' from opcode opcode to Filter Cache $\leftarrow PC+k$ return: pop top off stack and discard branch or indirect: reset Flag

**TABLE 5.12**  
Recursion Diverger actions



**FIGURE 5.16**  
Recursion data space

### 5.5.5. Branching and Linear anticipation

Before attempting to augment the Recursion anticipation design to accommodate branching, it is easier to show the extension of the simple linear anticipation for

branching alone. Later recursion and branching are combined, but here each is shown as independent extensions to the linear case.

The point model of the PC of the CPU of the linear case is replaced with a branching set model. The single PC value is replaced with a tree of values, each element of which is shown in Figure 5.17. This allows multiple simultaneous active PC values, denoted by the active leaves. ‘Active’ denotes a valid PC model, which was indicated in the prior designs by a set Flag value. Each leaf of the tree can be active or inactive. Interior nodes of the tree are inactive by definition.

Now that branching has been added to the set of opcodes accommodated, the components of the design take on activities denoted by their names. The Branching Filter Cache send portion emits messages indicating either which branch was taken, or how to reactivate an inactive leaf of the data structure (caused by indirect and return opcodes) (Table 5.13). The Branching Filter Cache has a different internal structure, modified to retain PC values in a shift register in order to enable indirect and return opcode messages (Figure 5.18).

The Branching Filter Cache receive portion of the branching design performs the filter function, matching outgoing PC values to incoming (PC,opcode) pairs (Table 5.14). In this way multiple alternate streams of opcodes sent by the Branching Code Pump can be distinguished.

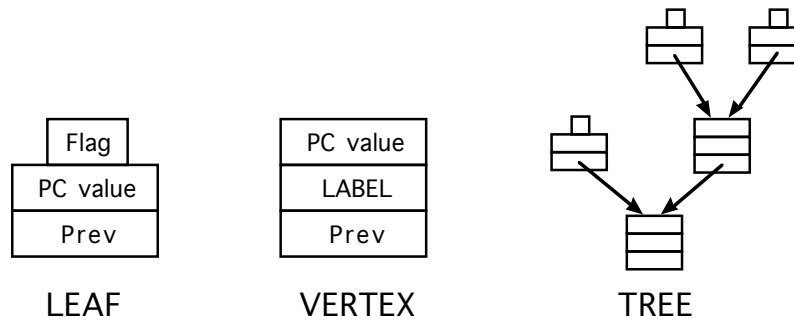
The Branching Diverger extends the state space of the Branching Code Pump’s model of the CPU’s PC value, by splitting a single leaf into a branch with two leaves, in the case where a branch is encountered. Each arm of the branch has its own new PC value, and is labelled with the first PC value encountered on the branch path. The Branching Converger matches incoming branch selection PC values to the set of all branch labels, indicating a node in the tree where the CPU state has been in the past. The nodes in the tree superior to the indicated node are possible subsequent states to the past CPU state, and are kept; all other states denote possible states which the received PC value has invalidated, and are deleted.

Only leaves in the tree indicate currently active paths, and so the ‘DMA<sup>1</sup>’ in the Branching Diverger sends opcodes for each active leaf. Leaves which encounter indirect or return opcodes are inactivated. A separate message type from the Branching Filter

---

<sup>1</sup>DMA stands for Direct Memory Access, and represents a component similar to the system-level component of the same name.

Cache reactivates these leaves and deletes the whole remainder of the tree when received. These activities are indicated in the specifications of the Branching Converger (Table 5.15) and Branching Diverger (5.16).



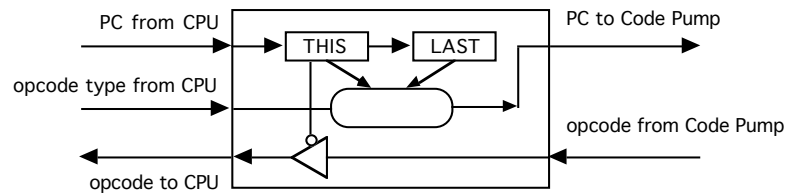
**FIGURE 5.17**  
Branching data space

Current CPU opcode	Action
'other', jump, call, return	{ no action }
branch	send ('B', current PC) pair
indirect	send ('I', current PC, previous PC) triple

**TABLE 5.13**  
Branching Filter send actions

Message received	Action
(PC, opcode) pair	if (current PC = PC) then send opcode to CPU else { ignore pair }

**TABLE 5.14**  
Branching Filter receive actions



**FIGURE 5.18**  
Branching Filter Cache design

Message received	Action
('B',thisPC)	find thisPC among branch labels delete all but tree superior to found node
('I',thisPC,lastPC)	find lastPC among inactive leaves delete all but found leaf activate found leaf

**TABLE 5.15**  
Branching Converger actions

Data area item	Action
----------------	--------

for each active leaf	fetch opcode at leaf_PC send opcode to Filter Cache Type of opcode: 'other': leaf_PC $\leftarrow$ leaf_PC+1 jump, call: get 'k' from opcode send opcode to Filter Cache leaf_PC $\leftarrow$ leaf_PC+k branch: mark current leaf as inactive get 'k' from opcode add child leaf1 leaf1_PC $\leftarrow$ leaf_PC+1 activate leaf1 leaf1_label $\leftarrow$ leaf_PC+1 leaf1_prev $\leftarrow$ leaf            add child leaf2 leaf2_PC $\leftarrow$ leaf_PC+k activate leaf2 leaf2_label $\leftarrow$ leaf_PC+k leaf2_prev $\leftarrow$ leaf        return or indirect: inactivate leaf
----------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

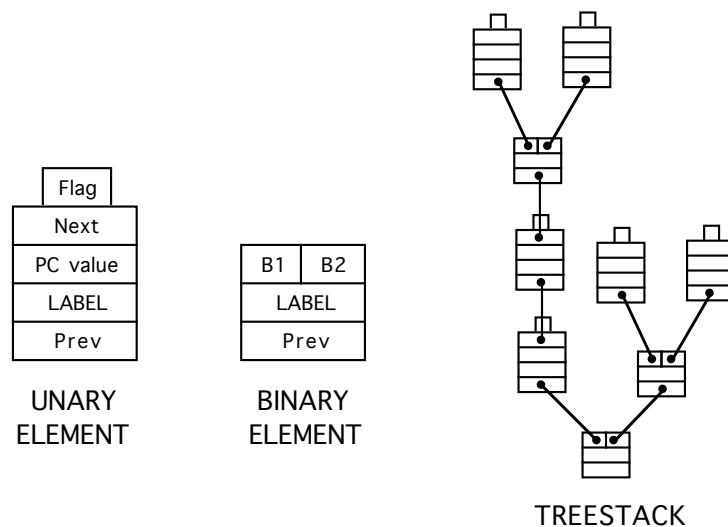
---

**TABLE 5.16**  
Branching Diverger actions

### 5.5.6. Combining Recursion and Branching anticipation

The final and most complete  $\mu$ -Net design includes anticipation of both branches and recursion. This requires a merging of the aspects of the Recursion anticipation and Branching anticipation, which is called Total anticipation. Indirect opcodes are still not anticipated in the Total version, because they cannot be anticipated at all<sup>1</sup>.

First, the data structure is augmented to provide aspects of the stack required for recursion and the internal vertices required for branching (Figure 5.19). There are two types of data structure components, unary and binary elements. A leaf is a unary element with an empty 'next' pointer, and denotes one of the possible CPU PC values being modeled (either active or inactive). A restoration is an inactive unary element, and encodes a past call location, to be used when a return opcode is encountered. A vertex is a binary element. Active leaves denote modeled PC values, and inactive leaves denote pending indirect opcodes, which cannot be modeled.



**FIGURE 5.19**  
Total Anticipation data space

---

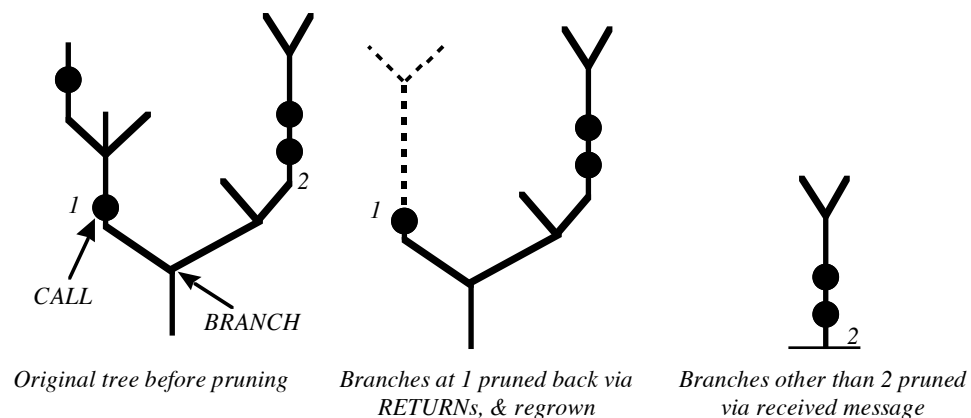
<sup>1</sup>Again, we claim that indirect opcodes which are limited to compile-time jump labels are equivalent to a fixed set of branches, and preclude the intent of an indirect opcode, which is to permit unpredictability.

The TreeStack is a more general structure than either a stack or tree. The Recursion anticipation data space consists of unary elements only, i.e., in a linear stack. The Branching anticipation data space consists of vertices and leaves only, i.e., a simple binary tree, with no internal unary elements.

### 5.5.6.1. The TreeStack

A TreeStack is a union of the notions of a tree and a stack. It is both a tree of stacks and a stack of trees. The stack implements the maintenance of the pending stack information (as produced by CALLs and consumed by RETURNS), in concert with path divergence information as indicated by BRANCHes. With respect to the stack, a CALL causes a push, and a RETURN refers back to the most recent pending CALL on the path previous to that PC. When all arms of a branch have been RETURNED or if one is selected (via a received message), the unused arms disappear (are pruned); in this way the tree aspect of the structure is managed.

The purpose of a TreeStack is to permit storage of RETURN addresses when a CALL message is sent (to model the recursion of the state space), and to permit the storage of the pairs of resultant PCs when a BRANCH message is sent. RETURNS should cause the structure on the path back to the originating CALL to be popped, unless these structures remain in use by pending BRANCHes. Multiple branches are pruned either when specifying information is received from the processor, or when all arms are popped as a result of RETURNS (Figure 5.20).



**FIGURE 5.20**

TreeStack structure: RETURNS prune, received messages specify

For example, if a CALL occurs, then a BRANCH, a RETURN from the result of either branch should cause a transformation to the same stored CALL address. Further, when a both arms of the branch have thus been transformed, i.e., when RETURNS have been sent from all PCs in the space of the CALL, that space disappears from the image.<sup>1</sup>

There are other ways to consider the interaction between the sending of code, received updates of the processor's actual state, and the structure of the TreeStack. Sending out code causes the tree to grow, except in the case where a branch arm indicates a RETURN, in which case the branch arm collapses to reflect the effect of the RETURN. Received messages cause the TreeStack to be pruned, such that the received address is the root of the resulting tree, branches superior to that tree location remain, and branches subordinate to that tree location are pruned.

The transformations describing the TreeStack as a general data structure appear in Appendix G. This structure may be of more general use.

#### **5.5.6.2. The Total implementation of $\mu$ -Net**

The Filter Cache sending mechanism of the Total implementation is changed only slightly from the Branching version, because return opcodes are ignored, and indirect opcodes send triples consisting of an indicator, the current PC, and the previous PC (where the indirect opcode occurred) (Table 5.17). The Total Converger needs the previous PC to match the indirect opcode execution to the correct leaf in the TreeStack, because the path back to the root must be maintained to encode the recursion stack. This requires extending the Filter Cache internally to store PC values through a 2-element shift register (Figure 5.21). The Filter Cache receive design is unchanged (it is repeated here for convenience in describing the Total design) (Table 5.18).

The Total Converger is modified to maintain the recursion stack information as well as the superior tree graph of possible futures of the CPU PC (Table 5.19). Some compression of the paths returning to the root may be possible, by deleting vertices on the path when the vertex is not required to join two converging root paths. Multiple simultaneous root paths are possible, because more than one label can be matched by an incoming branch message.

---

<sup>1</sup>This is similar to virtual/real particle pair interaction, such that eventually the pair collapses. Mirage was conceived of in terms of such physics analogs, as described in Appendix B.

The Total Diverger operates as a combination of the tree maintenance mechanism of the Branching diverger and the stack mechanism of the Recursion diverger (Table 5.20). ‘Other’, jump, and branch opcodes are accommodated as in the Branching diverger. Call opcodes transform a leaf into an internal unary element, in the fashion of a simple stack push operation.

The return opcode is slightly more complex to accommodate in the TreeStack structure. The problem is that elements on the path back to the corresponding prior call (i.e., elements on the path back to the first internal unary element on the path back towards the root) cannot be popped, because they may encode information which other pending branches still need. Also, when branch resolution is indicated (by a received message), the Total Converger may attempt to find a label in part of the tree which was deleted.

In terms of recursion and branching, consider a call opcode then a branch. One branch may return and continue execution in the prior environment, but the information of the call (i.e., its PC) cannot be destroyed, because the other branch will need it to execute its eventual return. Also, even when all branches in the recursed space return, the tree of branches must be maintained, because incoming branch resolution information indicates which returned path is valid. In other words, because a branch can induce multiple paths back through the embedded spaces, the paths must be maintained.

The Total Diverger accomplishes this by copying the found prior unary node, and replicating it. The replicate (now a leaf) is activated, and the current leaf (where the return opcode occurred) points back to the replicate. The tree can therefore have branches whose limbs recombine, but no circular paths are ever created by these operations, so the notions of ‘superior tree’ and ‘path to root’ are maintained.

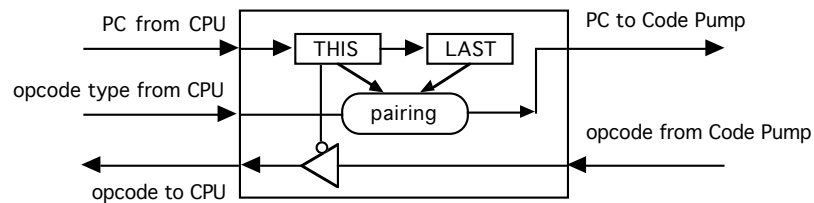
Current CPU opcode	Action
‘other’, jump, call, return	{ no action }
branch	send (‘B’, current PC) pair
indirect	send (‘I’, current PC, last PC) triple

**TABLE 5.17**  
Total Filter send actions

Message received	Action
(PC, opcode) pair	if (current PC = PC) then send opcode to CPU else { ignore pair }

**TABLE 5.18**

Total Filter receive actions (same as Branching filter)

**FIGURE 5.21**

Total Filter Cache design

Message received	Action
('B', thisPC)	find thisPC among branch labels unmark all elements for each found element: mark all elements in superior tree mark all elements on path to root delete all unmarked elements
('I', thisPC, lastPC)	find lastPC among inactive leaves unmark all elements for each found element: found_PC $\leftarrow$ this_PC activate found element mark all elements on path to root delete all unmarked elements

**TABLE 5.19**

Total Converger actions

Data area item	Action
----------------	--------

---

---

```

for each active leaf
  fetch opcode at leaf_PC
  send opcode to Filter Cache
  Type of opcode:
  'other':
    leaf_PC  $\leftarrow$  leaf_PC+1  jump:
    get
  'k' from opcode
    send
  opcode to Filter Cache
    leaf_PC  $\leftarrow$  leaf_PC+k  call:
    get
  'k' from opcode
    create new leaf
      new_prev  $\leftarrow$  leaf
      new_PC  $\leftarrow$  leaf_PC+k
    activate new leaf
    deactivate this leaf
    leaf_next  $\leftarrow$  new_leaf  branch:

    replace current leaf with vertex
      vert_label  $\leftarrow$  leaf_label
      vert_prev  $\leftarrow$  leaf_prev
      vert_b1  $\leftarrow$  leaf1
      vert_b2  $\leftarrow$  leaf2
    get 'k' from opcode
    add child leaf1
    leaf1_PC  $\leftarrow$  leaf_PC+1
    activate leaf1
    leaf1_label  $\leftarrow$  leaf_PC+1
    leaf1_prev  $\leftarrow$  leaf      add
  child leaf2
    leaf2_PC2  $\leftarrow$  leaf_PC+k
    activate leaf2
    leaf2_label  $\leftarrow$  leaf_PC+k
    leaf2_prev  $\leftarrow$  leaf      return:
    find first
  unary antecedent
    copy found
  element
    new_prev  $\leftarrow$  found_prev
    new_PC  $\leftarrow$  found_PC+1
    this_next  $\leftarrow$  new
    activate new leaf
    deactivate this leaf
  indirect:
    deactivate leaf

```

---

**TABLE 5.20**  
Total Diverger actions

## 5.6. Implications

The implications of the  $\mu$ -Net architecture on the performance of the processor/memory system can now be considered. The performance equations defined before are repeated below:

**Equation 5.1:**  $T_{optimal} = N * t$

**Equation 5.2:**  $T_{conventional} = N *(t + r)$

**Equation 5.5:**  $T_{cache} = N *(t + r * M)$

**Equation 5.7:**  $T_{prefetch} = N *(t + r * F)$

where  $F \leq J + B / 2 + C + R + I$

**Equation 5.11:**  $T_{\mu-Net} = N *(t + r * P)$

where  $T$  = total time to execute  $N$  instructions

$t$  = time for the processor to execute 1 instruction

$r$  = round trip time between the processor and memory

$M$  = probability of a conventional cache miss

$F$  = prob. of a conventional prefetching cache hit

$P$  = probability of the  $\mu$ -Net Code Pump miss

The values of  $M$ ,  $F$ , and  $P$  can be compared. As the time separation between memory and the processor increases, the  $r$  component of the equations begins to dominate. The goal is to determine the conditions under which  $P$  is less than  $M$  or  $F$ .

The value of  $P$  can be further elaborated, in the various levels of implementation of  $\mu$ -Net. The levels of implementation describe the degree for which various opcode classes are included in the send transformations. Implementations model 'no' send transformations, send transformations on regular opcodes only, on regular and jump

opcodes, etc. These correspond to predicting the next opcode (and sending it) after regular opcodes, jumps, calls, etc.

The following formulae describe the ways in which the prediction can alleviate the effects of round trip latency (Equations 5.13, 5.14, 5.15, 5.16). Evaluation of these formula, thus far, depends only upon measurements of the probability of each class of instruction type.

**Definitions:**  $O$  = other instructions  
 $J$  = jumps  
 $C$  = calls  
 $R$  = returns  
 $B$  = branches  
 $I$  = indirect (calls, branches, or jumps)  
 where  $N = O + J + C + R + B + I$

**Equation 5.13:**  $T_{null} = N * (t + r)$

**Equation 5.14:**  $T_{other} = N * (t + (J + C + R + B + I) * r)$

**Equation 5.15:**  $T_{other, jump} = N * (t + (C + R + B + I) * r)$

**Equation 5.16:**  $T_{other, jump, call} = N * (t + (R + B + I) * r)$

These formulae require a constant sized Code Pump (i.e., only a finite amount of storage to implement the imaging of the PC of the processor), because they consider only transformations which preserve the size of the image.

If the Code Pump is augmented to accommodate an arbitrary amount of stack space, as required to handle the pending *calls* that could occur during one round trip time, the transformations indicated by RETURN opcodes can be modeled, as shown in Equation 5.17. Because all the types of opcodes whose transformations have been modeled thus far preserve the size of the PC image, issues of bandwidth in the communication can be ignored. Only one PC value is ever active, so the instruction communication consists of a single stream.

**Equation 5.17:**  $T_{other, jump, call, return} = N * (t + (B + I) * r)$

If the Code Pump is further augmented to accommodate an arbitrary amount of TreeStack space, as required to handle the pending *branch executions* that could occur in

one round trip time (i.e., the entire branch, not just the calls), the transformations indicated by BRANCH opcodes can be modeled as well.

Recalling the discussion of the Mirage model, there are limitations to this predictability beyond that of the data storage in the Code Pump. When the image space becomes large, the state space must be partitioned coarsely enough to be able to send a small enough number of messages such that the entire space is covered by the guards of the set of the messages sent; this is from our definition of stability (entropic, as also discussed in Chapter 2).

At this point that limitation becomes applicable. Expansion of the PC image is limited not only by the space needed to represent it (the size of the TreeStack structure), but also by the total number of messages in transit. All possible destinations of all pending PCs due to BRANCH instructions cannot be sent if the round trip time does not permit it. Once a BRANCH is encountered, one round trip time exists to send the messages corresponding to its isopotent set (the set of opcodes of every possible branch destination, in this case).

Partitioning the state space coarsely is not a concern here; there is not enough information in the opcodes of a program to collapse the information efficiently, i.e., to send a single opcode with a label of the five locations where it occurs. This kind of dynamic partitioning is less effective than simply repeating the opcodes for each address sent. Further, no compression of the state space (i.e., encoding of the addresses used as labels) is really necessary, although some useful assumptions can be made. For example, in many CPUs, branches, jumps, and calls are limited to some fixed maximum distance from the current PC, usually less than the limit of the entire PC space (i.e., short jumps vs. long jumps). If a short jump opcodes cause transformations of the PC which are, at most, 8 bits offset from the current PC, only the lowest 8 or 9 bits of address need be sent as the label. This latter minimization of communication is not the same as partitioning the space coarsely; it reflects only a useful encoding of the guards.

A version of Equation 5.17 reflects the use of the bandwidth to accommodate the multiple possible values in the memory's image of the processors PC (Equation 5.18).

**Equation 5.18:**  $T = N * (t + (I + X * B) * r)$

where  $X$  = percent of branch possibilities which cannot be communicated

$X$  can be measured directly, from an implementation (either direct or emulated), or can be approximated through the application of some assumptions. Recall the prior

discussion (Chapter 2), which defines *communicability* in the abstract model. This refers to the ability to partition the image space coarsely enough, and to send a small enough set of short enough messages, that the entire set that covers the image (the isopotent set) can be communicated in the information separation (bandwidth-delay product) available; if this can be done, and if the set sufficiently constrains the image space, then the image (and thus the communication) is stable (entropically, or otherwise).

In the abstract model state space volumes are introduced to represent images of remote state spaces. Each image either continues in time, or is transformed into a set of images. In  $\mu$ -Net, each PC value either continues as a single point (via JUMP, CALL, RETURN, or regular opcodes), splits into two points (BRANCH), or becomes a set too large to partition (INDIRECT).

Under the simplifying assumption that branches are indistinct, the ways in which branch possibilities can be communicated can be computed, thus determining the size of a isopotent set which can be communicated per round trip time. This formula was also developed earlier (Chapter 2, Branching Streams).

The ultimate goal is to determine the probability of not predicting an opcode, i.e., the ‘miss’ rate of the Code Pump, and to determine the storage required to maintain the TreeStack structure.

Let  $L$  be the limb length (typically 6 to 8 opcodes in length), let  $D$  be the branch degree (usually 2, because only binary branches are modeled), and let  $rtt$  be the round trip time, as defined in prior discussion of channel utilization. Also let  $OP\_BW$  be the bandwidth, in opcodes per unit time. In time  $rtt$ , the original PC of the processor attempts to execute opcodes (Equation 5.19, where  $CPI = \text{clocks per instruction}$  [He90]). In that time, only a portion of the tree of possible execution streams can be transmitted (Equation 5.20). With respect to an individual execution path, only a portion of the path is sent, i.e., the portion equivalent to the tree depth (Equation 5.21). The ratio of the path length which the CPU attempts to execute in the round trip time to the path length in the tree sent is the probability of success (Equation 5.22).

$$\text{Equation 5.19: } \textit{opcodes\_executed} = \frac{rtt}{t}$$

$$\text{where } t = \frac{CPI}{\textit{clock\_rate}}$$

**Equation 5.20:**  $depth$  such that  $rtt * OP\_BW = \frac{D * (D^{depth} - 1)}{D - 1} * L$

**Equation 5.21:**  $depth = \log_D \left( \frac{rtt * OP\_BW * (D - 1)}{L * D} + 1 \right)$

**Equation 5.22:**  $tree\_success = \frac{\log_D \left( \frac{rtt * OP\_BW * (D - 1)}{L * D} + 1 \right)}{opcodes\_executed}$

This analysis focuses on prediction failure based on the inability to transmit enough alternates in the branching stream; it ignores the effects of INDIRECT ipcodes, because they are so infrequent (less than 0.3%).

Equation 5.17 refers to Equation 5.23, which depends on the growth of the state space image as determined by the latency, and the management of that growth which is limited by the information separation (i.e., Equation 5.22). The result is the effective communicability in  $\mu$ -Net (Equation 5.24).

**Equation 5.23:**  $X = 1 - tree\_success$

i.e.,  $X = 1 - \frac{\log_D \left( \frac{rtt * OP\_BW * (D - 1)}{L * D} + 1 \right)}{\frac{rtt}{t}}$

**Equation 5.17:**  $T = N * (t + X * r)$

**Equation 5.24:**  $T = N * \left( t + \left( 1 - \frac{\log_D \left( \frac{rtt * OP\_BW * (D - 1)}{L * D} + 1 \right)}{\frac{rtt}{t}} \right) * r \right)$

Before this point in the discussion, the round trip latency did not figure into the calculation of performance. In considering the way in which the state space image can expand beyond the ability to manage it, latency becomes an issue. Latency determines to what extent the expansion of the image can be managed. As a result, it plays a role not

only in the penalty assessed when prediction fails, but also in the evaluation of the frequency of penalization ( $X$ ).

In the limit of this equation as the available bandwidth goes to infinity, all isopotent sets can be sent, and  $X$  consists of only those instructions whose transformations cannot be modeled, those of INDIRECT opcodes. These opcodes require a round trip latency in which to determine the resultant PC, as computed at the processor (Equation 5.25). This equation represents an ‘Amdahl’s Law’ of communicating interaction, i.e., that the speedup is limited to the predictable component, so the speedup is limited to the reciprocal of the percentage of indirect instructions 5.26.

**Equation 5.25:**  $T_{optimal \mu-Net} = N * (t + I * r)$

**Equation 5.26:**  $Speedup_{max} \leq \frac{1}{I}$

In these equations,  $OP\_BW$  is the number of opcodes sent per unit time, and  $t$  is the time to execute an opcode, the two of which are related via the CPU  $clock\_rate$  and the  $CPI$  (cycles per instruction). Typical values for  $CPI$  values are as low as 1.0 for true RISC processors, 1.3 for a heavily pipelined CISC (68040) [Mo89], and up to 14 for other CISCs (68000) [Ma84] (Table 5.21). The  $clock\_rate$  of a CPU is typically 10 to 30 Mhz.

Processor	CPI
Motorola 68000	13.5 [Ma84]
Motorola 68010	11.4 [Ma84]
Motorola 68020	6.6 [Ma84]
Motorola 68040	1.3 [Mo89]
DecStation 3100 (MIPS)	1.87 [He90]
Sun 3/75 (680x0)	10 [He90]
DLX (RISC) [He90]	6.28 [He90]

**TABLE 5.21**  
CPI (EXEC) values of common CISC and RISC CPUs

## 5.7. Conclusions

This chapter presented equations describing the performance of processor-memory protocols, and developed a new protocol called  $\mu$ -Net based on the abstract Mirage model.  $\mu$ -Net's design, and the various degrees of its implementation were also presented. Performance equations require either detailed modeling of the expected opcode stream, or measurement of real opcode streams. Initial modeling here indicates that the performance of  $\mu$ -Net depends on the internal structure of the stream, specifically, how the stream branches and how long each branch persists.

In order to compare these equations, detailed measurements are required of real opcode streams. These measurements include the expected branch degree, and the expected limb length. Other required measurements include the distribution of the opcode classes. Prior work can indicate the expected values of cache miss ( $M$ ) or miss with prefetch ( $F$ ). Description of these measurements and analysis of the results appears in Chapter 6.