CHAPTER 6

μ-Net under a μ-Scope

Evaluation of μ–Net requires measurement of cache miss (*M*), prefetch cache miss (*F*), and μ–Net miss (*P*) probabilities. Cache performance parameters are available in the literature [Sm82]. Some measurements of opcode statistics have also been published [He90], but some of the statistics required to evaluate μ–Net are not available, and were made by direct measurement of opcode execution. The measured design is called μ–Net (MicroNet), so this measurement method is called μ–Scope (MicroScope). A description of μ-Scope appears in Appendix H.

Required measurements which were not available in the literature include statistics of opcode distributions according to the partitioning indicated by μ–Net. μ–Net also requires measurements of the average number of non-control transfer opcodes between control transfer opcodes; the definition of control transfer differs for the various levels of design discussed (Chapter 5). μ-Scope was developed to perform these measurements. Other tools for opcode distribution measurement were considered (as discussed in Appendix H), but were either insufficient for the measurements desired (Pixie) or proprietary and not available to us.

These measurements are made on the dynamic trace of opcode executions of various benchmarking programs. The applicability of benchmarks for general analysis is not advocated here. This selection represents widely available benchmarks which

compiled and ran without explicit error in μ-Scope. Other benchmarks were considered, but omitted because they were not available in either SPARC Assembler or C language source code (which μ-Scope is limited to), or because of system limitations during the preparation of this dissertation (i.e., the benchmark was of limited interest due to its specificity, and there was insufficient disk space and insufficient processing capability to examine all available benchmarks).

The GNU C, T_EX, Linpack, and Dhrystone benchmarks were chosen for these measurements. GNU C is a freely distributed C language compiler; our version (1.35) and the test set weightings used were extracted from the SPEC Benchmark Release 1.0 [Wa90]. T_EX is an embedded text typesetting program; our version was contained in the benchmark distribution of [He90][1]. The C language versions of both Dhrystone and Linpack were obtained from standard Internet libraries[2]. Each benchmark has particular characteristics, listed below:

- GNU C - a large program, complex, recursive; uses many language features
- T_EX - a typical large frequently-used program
- Linpack - a set of routines collected from a mathematical software library
- Dhrystone - a contrived program claimed to represent intense integer computing

Both Linpack and Dhrystone represent intense mathematical computing, but lack sophisticated use of compiler and language constructs. They are often used for estimating measurements of theoretical CPU performance, such as FLOPS or MIPS.

T_EX represents a common but complex application program, exhibiting significant levels of recursion and opcode proportions in general purpose systems. The GNU C compiler represents a typical upper-bound (or at least a reasonable bound) of complexity, both in its static structure and dynamic execution.

Some of the measurements performed here have also been presented elsewhere. They are repeated here because it is important to compare the measurements of various

---

[1]The software supplement to [He90] is available via anonymous FTP at max.stanford.edu.

[2]These benchmarks are available via Internet e-mail; send queries to netlibd@surfer.epm.ornl.gov.

characteristics on a common set of benchmarks, and because some necessary measurements were not found in the literature, notably the variability in limb length with various treatments of jump, call, and return instructions, as discussed in detail below.

6.1. Performance gains

The performance gain of μ–Net is measured by the extent to which *P* is less than either *M* or *F*. We need to compare actual measurements in order to make these measurements. As noted before, there is no incompatibility between conventional caches and μ–Net, so the comparison is simplified by the assumption that μ–Net also has a conventional cache inside the Filter Cache. The Filter Cache then behaves like the conventional cache, with the exception of the one entry which represents the latest opcode/address pair as received from the incoming communication stream. Further, if this 'streaming' entry is hit, it is copied into a conventional entry, just as when a memory reply occurs after a conventional cache miss.

In prefetching caches, a miss of a particular datum causes a sequence of data to be fetched. This is equivalent to the assumption that all opcodes are of type OTHER (as described in Chapter 5) (i.e., simply increment the PC), and that the memory will send a number of opcodes equivalent to the line size or lookahead of the cache upon a miss.

Communication can be modeled as a branching stream (as described in Chapter 2). The branching of this stream is 2 because BRANCH opcodes accommodate 2 alternates. The limb length of the stream describes the extent to which a particular image value (PC) remains predictable by transforms. In μ–Net, limb length is the average number of non-control flow opcodes between control flow opcodes. Opcodes which alter the flow of control are those that cause branching in the stream (BRANCHes), or that cause a round trip latency penalty because they are not anticipated (INDIRECT, and opcodes not anticipated in a given implementation).

Dynamic opcode traces (i.e., execution traces) indicate the following distributions of opcode classes. I-Call' and 'I-Jump' denote indirect CALL and JUMP opcodes, respectively; SPARC CPUs have no indirect branches (Figure 6.1). Other published measurements [Ka91] of *troff* (GK troff) and *cc* (GK cc) are included for comparison.

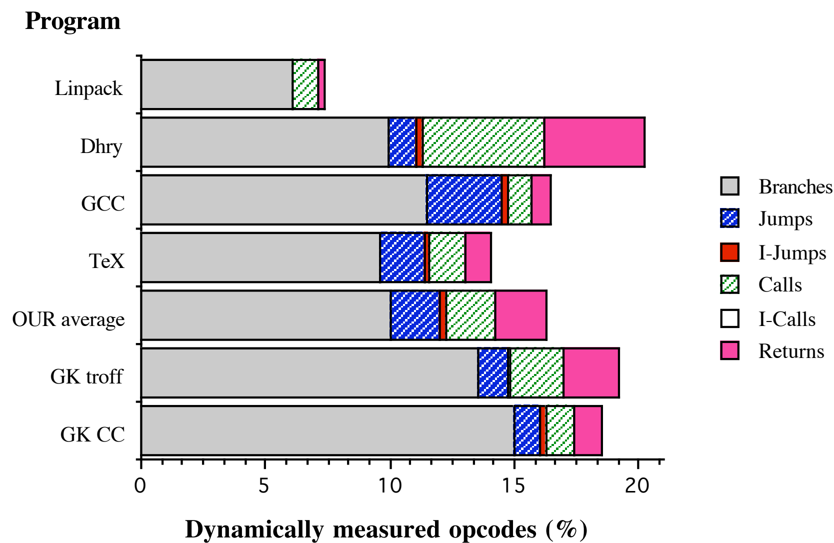

FIGURE 6.1
Dynamic control opcode distributions

In the following discussion, INDIRECT refers to the aggregate of all indirect opcode types. These measurements were made on the actual execution of the benchmarks on a SPARC (SUN-4), which has no indirect branch opcode. The vast majority of the indirect opcodes were indirect JUMPs, due to a preference in the C compiler used[1].

The code was measured by compiling the benchmarks (using optimization), and generating SPARC assembler code. The assembler was modified by an AWK script, which inserted additional instructions to count the various classes of opcodes during execution, as well as measuring the number of OTHER opcodes between control of flow instructions, such a JUMP, BRANCH, CALL, and RETURN. Existing tools, such as PIXIE and PIXIESTATS for the MIPS processor, were not used because the method for determining basic blocks causes errors in the measurement of opcodes due to the effects of indirect JUMPs and CALLs. The SUN equivalents, SPIX and SPIXTOOLS, as well as the SUN SPARC emulator SHADOW, were requested but not available for public use. See Appendix H for a further discussion on the measurement techniques used here.

---

[1]In the test code we measured, the SPARC C compiler generated indirect CALLs only where structures were returned as function values (4 times), and only in the GNU C compiler code. Indirect JUMPs appeared in almost all of the test code, in very small (<0.3%) amounts.

The variance in these measurements is large, due to the arbitrary choice of canonical code examples, and the variability between benchmarks. Estimates of the measurements are summarized in Table 6.1. These are verified by similar, albeit more limited published measurements [Ka91][1].

OPCODE	%
Other	84
JUMP	2
CALL	2
RETURN	2
Indirect	0.3

TABLE 6.1
Approximate dynamic opcode distribution

Comparing μ–Net to an architecture without a cache, μ–Net reduces the effect of latency by the percent of instructions which can be accommodated by the Code Pump (Figure 6.1, Table 6.2).

These comparisons assume that the majority of time spent in execution is due to round trip latency, i.e., that $r >> t$ by at least one order of magnitude, preferably at least two. This necessitates a latency of at least 100 instruction execution times, which in a 1 gigabit/sec pipe, assuming 32-bit instructions, corresponds to a latency of 3,200 bits, or 3.2 μsec, or 960 meters, which is about 6 city blocks. For example, these results apply where the processor and memory are separated by at least 3 city blocks, which is completely reasonable for a client/server system located on a small college campus.

[1]Our method of dynamic tracing, μ-Scope, was developed using methods similar to those which appear in the Katevenis paper. To use a common euphemism, we reinvented their wheel, then checked theirs to tune ours. A further discussion of μ-Scope appears in Appendix H.

Opcode groups anticipated	% not predicted	Speedup	Caveats
none (conventional)	100	1	
Other	16	6.3 x	
Other, JUMP	14	7.1 x	
Other, JUMP, CALL	12	8.3 x	
Other, JUMP, CALL, RETURN	10	10 x	unlimited stack
all but Indirect	0.3	333 x	unlimited TreeStack

TABLE 6.2
Approximate speedup in various degrees of implementation

Measurements of cache utilization have appeared often in the literature [He90] [Sm82]. The miss rate M is approximately 3%, which with prefetching (F) drops to 1.5%. These assume extremely large (or infinite) caches. A 1K cache has a miss rate of about 20%, a 4K about 12%, a 128K about 5%, and the miss rate approaches 3% only for caches larger than 256K.

Large (infinite) caches miss mostly due to first time code use, whereas smaller caches miss both from first time code use and from most control opcodes, because the cache is so small that changes in control flow are likely to require opcodes not in the cache. A prefetching cache misses only from control flow opcodes because first time code use is predicted by the prefetch.

Changes in control flow occur from JUMPs, CALLs, RETURNs, and INDIRECT opcodes, as well as from half of the BRANCHes, the latter assuming that BRANCHes are taken about 50% of the time [He90]. Such control opcodes comprise 11.3% of the instruction stream (JUMP, CALL, RETURN, INDIRECT, and half of the BRANCHes), so control opcode misses comprise 11.3% of the cache misses, i.e., prefetching caches are estimated to miss 11.3% of the time.

This estimate can be further refined because branches can be separated into backward and forward branches, where forward branches are more likely to cause misses. About 25% of branches are backward (empirically [He90]), because backward branches

are generated for loops, whereas forward branches are generated for IF and CASE statements, which are more common in source code. Backward branches are more likely to hit existing cache entries, so a revised formula for misses in small prefetching caches is 10% (JUMP, CALL, RETURN, INDIRECT, and half of 3/4 of the BRANCHes).

Prefetching reduces the miss rate up to 50% for large caches, compared with non-prefetching caches [Sm82]. Misses occur due to CALLS and RETURNS more than BRANCHES, because most local code remains in the cache. A small cache has a 20% miss rate and 50% of these misses are predictable (due to the opcode distribution difference), so the miss rate of a small prefetching cache is expected to be 10%, equivalent to our approximation based on opcode distributions.

Cacheing and cache anticipation are compared to μ–Net anticipation, in Table 6.3, assuming nothing about branch outcome. Opcodes are called JUMP, CALL, RETURN, BRANCH, INDIRECT, and OTHER for all other types, and labelled respectively J, C, R, B, I, & O [He90]. The table notes the cache size required to approximate various versions of μ–Net, based on which sets of opcodes are anticipated. In the Recursion case, storage is required in the Code Pump of 100 addresses; justification of this figure is discussed later.

μ–Net Version	Opcodes anticipated	miss rate	equivalent cache	μ–Net storage
Unit Linear	O	16%	2K bytes	none
Linear	O+J+C	12%	4K bytes	none
Recursion	O+J+C+R	10%	8K bytes	100 addresses

TABLE 6.3
μ–Net implementations and cache equivalents (no branch assumption)

Assuming branches are taken 50% of the time, i.e., that half the branches are predicted, more dramatic comparisons result (Table 6.4).

A naive comparison of these numbers results in the conclusion that most degrees of implementation of μ–Net are far worse than either caches or prefetching caches. Although cache miss rates are often quoted as 3% (1.5% for prefetching caches), these rates apply only to very large caches. Current microprocessors have small caches (8K for an Intel 80486, 256 bytes for a Motorola 68030, 4K for a Motorola 68040 and Intel

80860 [Mo89]). In comparison, μ–Net beats most of these on-chip caches, if transmission latency is large.

μ–Net Version	Opcodes anticipated	miss rate	equivalent cache	μ–Net storage
Unit Linear	O	11%	8K bytes	none
Linear	O+J+C	7%	16K bytes	none
Recursion	O+J+C+R	5%	50K bytes	100 addresses

TABLE 6.4
μ–Net implementations and cache equivalents (equiprobable branching)

Opcode	% of control	estimated cache miss distribution
OTHER	-	50 %
JUMP	12 %	6 %
CALL	12 %	6 %
RETURN	12 %	6 %
BRANCH	62 %	31 %
INDIRECT	2 %	1 %

TABLE 6.5
Adjusted dynamic opcode distributions (approx. a cache miss stream)

These results can be extrapolated to a system where μ–Net complements cacheing, rather than replacing it. Opcode distributions change with the inclusion of a cache, and measurements including a cache implementation were beyond the scope of this dissertation. One estimate assumes that the proportions of control opcodes (relative to each other) remains the same, between a conventional instruction stream and the stream of cache misses. Assuming that control opcodes increase to a total of 50% of the cache

miss stream [He90], keeping all other relative proportions the same, this results in an adjusted opcode distribution (Table 6.5), and expected speedup (Table 6.6). The combined entries (cache with μ–Net) indicate that a resulting miss is the result of the combination of a cache miss and a failure of μ–Net to have anticipated the request for that opcode (i.e., the miss and failure rates are multiplied).

Implementation	Failure rate	Speedup (ratio to null)
cache alone (large- 256K)	3%	33 x
cache alone (small- 4K)	10%	10 x
prefetch cache (large- 256K)	1.5%	67 x
small cache + μ–Net(O)	10% * 50% = 5%	20 x
small cache + μ–Net(O,J)	10% * 44% = 4.4%	22 x
small cache + μ–Net(O,J,C)	10% * 38% = 3.8%	26 x
small cache + μ–Net(O,J,C,R)	10% * 32% = 3.2%	32 x
small cache + μ–Net(O,J,C,R,B)	10% * 1% = 0.1%	1000 x
large cache + μ–Net(O,J,C,R,B) [BEST]	3% * 1% = 0.03%	3,333 x

TABLE 6.6
Performance increases where latency dominates design parameters

According to this table, a small cache with a Recursion μ–Net (using 400 bytes of internal stack space) uses a total of 4.4K bytes of space, but achieves the miss rate equivalent to a 256K byte cache. Note that the Recursion μ–Net uses the same memory bandwidth as a system without a cache; only the Branching and Total μ–Nets require bandwidth higher than the Null design.

These are overestimates of the expected speedup, because the distributions of cache miss code are different from distributions of overall code use. Misses will be generated disproportionately by JUMPs, CALLs, RETURNs, and INDIRECTs, because these kind of control flow changes are not likely to be contained in code already accessed, due to the

principle of locality. Measurements of code distributions in cache misses were unfortunately beyond the scope of this dissertation, but we do not expect they would significantly change the conclusion that μ–Net/cache exceeds the speedup capability of cache/prefetch or cacheing alone.

More sophisticated experiments are required to extend this analysis to the Branching and Total μ–Net cases, as described later.

6.2. On the feasibility of implementations as architectures

The feasibility of these implementations as realizable architectures depends on the real size of the stack or TreeStack data structures in the Code Pump.

For example, the size of the stack structure in the Recursion μ–Net is equal to the largest number of pending RETURN opcodes during the entire code execution (i.e., the depth of recursion). Cumulative percentages of the CALL opcodes at each level of recursion are plotted in Figure 6.2. For a particular stack size, the plot indicates the probability of not being able to store a CALL address, for use by its corresponding RETURN. For example, 85% of CALL/RETURN pairs can be accommodated in the T_EX benchmark by a stack of size 17, whereas the stack would have to be increased to a size of 50 to handle the same likelihood of overflow in the GCC benchmark.

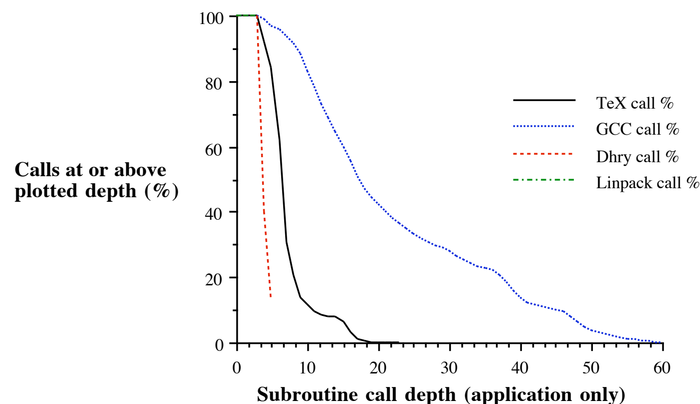

FIGURE 6.2
Percent of CALLs occurring at or above a given depth of recursion

Some of these depth measurements are not very accurate because the depth of pending calls could not be traced within the operating system with μ-Scope (Figure 6.3). This analysis is similar to that presented in [He90], where their results indicate that C compiled code has 5% overflow at a depth of 6, LISP at a depth of 7, and SmallTalk at a depth of 9, so the results depend on the program type as well as language and compiler characteristics.

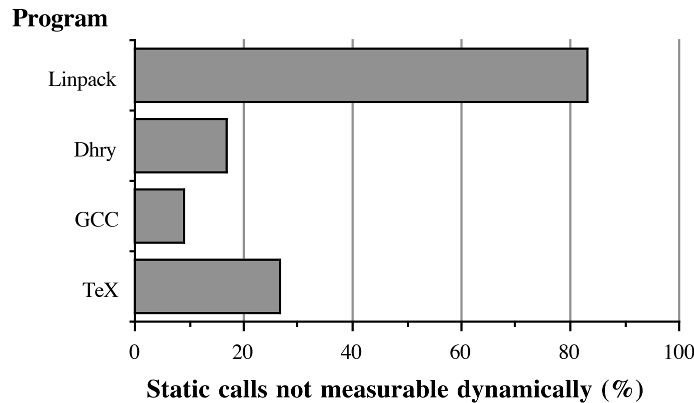

FIGURE 6.3
Percent of CALLS not depth-traced (system calls)

Because only RETURN addresses need to be stored, a stack as small as 10 items will handle all of the Dhrystone and Linpack RETURNs, and miss only 10% of the T_EX RETURNS, although it would fail in 85% of RETURNS in a highly recursive program, such as the GCC compiler. Increasing the stack to as little as 100 values allows the Recursion μ–Net to anticipate 100% of the user-code RETURN opcodes. Thus storage for a stack structure in the Code Pump which is large enough to accommodate all pending levels of recursion should be trivial to implement.

The size of the TreeStack structure also limits the feasibility of implementations. The TreeStack represents not only the information in the stack of pending recursions, but also the information on the expansion of the state space image. In order to estimate the size of this structure, there are two options. The first involves emulating the Code Pump, and determining the maximum extent of the structure as the benchmarks were executed. This level of experimentation was beyond the scope of this dissertation, but may be considered as a future research area.

The second involves some assumptions. Rather than directly measuring the size of the TreeStack, an estimate of its size can be made using the same simplifying assumptions as the branching stream analysis (Chapter 2). Assuming that the state begins in a finite set of values (i.e., a single PC, in this case), and that the sequence of PCs can be expressed as a tree, a snapshot of the image of the remote PC at any given time is expressed by the set of PCs in a level of the tree thus formed.

The branch degree of the tree is 2, modeling only BRANCH opcodes, because INDIRECT opcodes expand the space infinitely (actually, they expand the space by a degree of the size of the PC, at which point that level in the tree covers all possible PCs, as described before). The other relevant tree characteristic is the average limb length, defined as the number of opcodes between those which cause the tree to either branch or terminate (due to a terminated modeling).

For example, in the Total μ–Net, limb length is the number of opcodes between BRANCH or INDIRECT opcodes. In the Branching μ–Net, limb length is measured between BRANCH, INDIRECT, and RETURN opcodes. Figure 6.4 shows the average limb length in each benchmark, as measured for different degrees of implementation. In these graphs, ***linearity*** refers to limb length, i.e., linearities in the flow of control in the opcode stream.

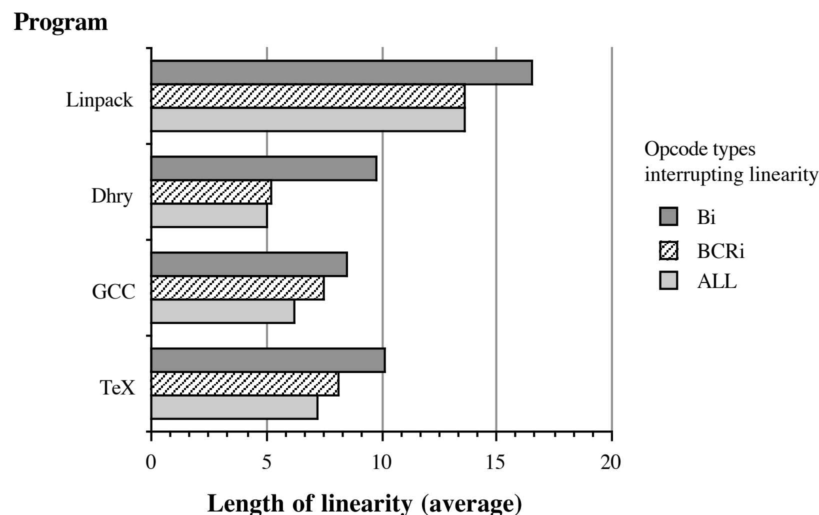

FIGURE 6.4
Average limb length (linearity)

For this graph (Figure 6.4), ALL refers to the Unit Linear μ–Net (i.e., all control opcodes delimit limb lengths). BCRi refers to the Linear μ–Net (BRANCH, CALL, RETURN, and INDIRECT), and Bi refers to the Total μ–Net. Other combinations were not measured, but would be interpolations of these values (e.g., Branching μ–Net and Recursion μ–Net), because the Total μ–Net exhibits the longest possible limb length, and Unit Linear exhibits the smallest. Modeling JUMP instructions (ALL/Unit Linear vs. BCRi/Linear) increases the average length by 10-20% from around 6 to around 7 opcodes, but that modeling CALLs and RETURNs (Recursion) increases the averages by about 40% (or as much as 100%), to around 9 (Figure 6.5).

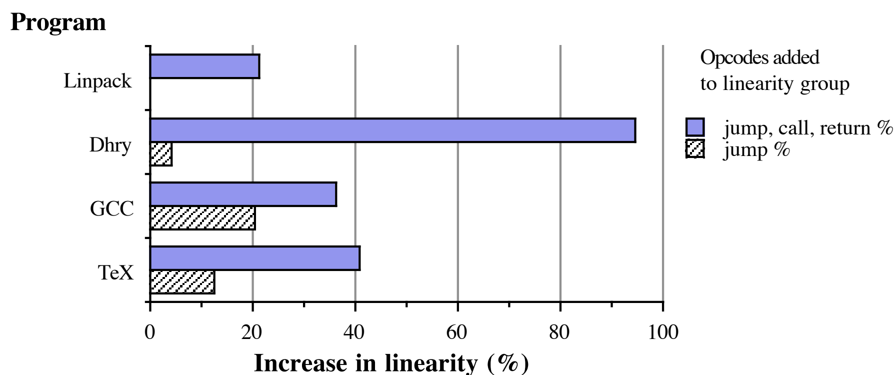

FIGURE 6.5
Percent increase in limb length (linearity), adding calls and returns

The distributions of limb lengths was also measured because the variance in these averages was very large (greater than 100%). These figures plot cumulative probabilities, i.e., for a given length, the ratio of the number of limbs at that length or less to the total is plotted, so that the point indicates “the probability that a random limb is shorter or equal to the limb length indicated” (Figures 6.6, 6.7, 6.8, 6.9). The mode of these distributions can be read directly from these plots, as the 50% value (i.e., 50% of the arm lengths are less than or equal to the plotted value).

In this set of plots, ALL indicates the limb length distribution of the Unit Linear μ–Net, BCRi indicates the limb length distribution of the Linear μ–Net, and Bi indicates the limb length distribution of the Total μ–Net. Linpack exhibits a jump in limb length statistics, presumably because it consists primarily of a set of nested FOR loops, and data from the inner portion of these loops overwhelms the statistics. The scientific programs (Linpack and Dhrystone) exhibit large limb length increases if CALLs and RETURNs are

accommodated within the limbs, because the repetitious structure of the code ensures interleaving of CALL/RETURN opcodes with BRANCHes; if CALLs and RETURNs are then included within the limb, limb length increases.

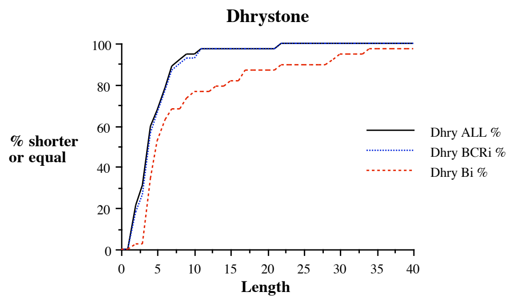

FIGURE 6.6
Dhrystone limb length distribution

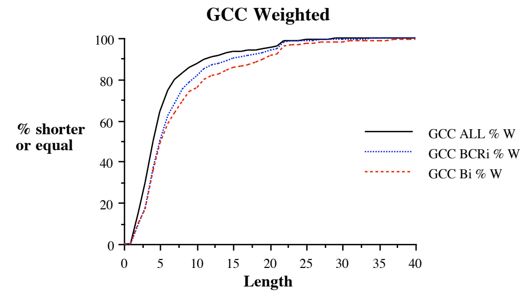

FIGURE 6.7
GCC (weighted) limb length distribution

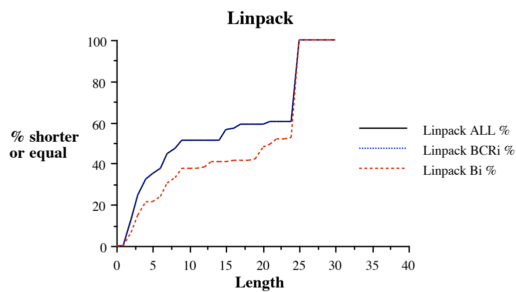

FIGURE 6.8
Linpack limb length distribution

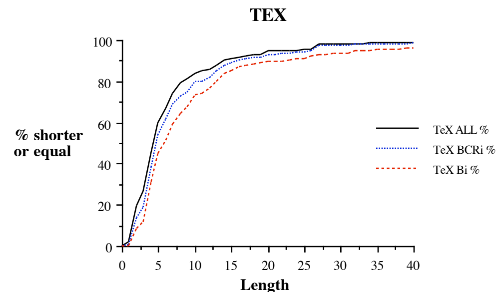

FIGURE 6.9
TEX limb length distribution

Cumulative branch arm length distributions of various benchmark programs

Rather than using the complete distributions shown above to evaluate the effects of branching on channel utilization, the mean and median of these distributions are used (Table 6.7). These measurements can be used to compare the effects of limb length increases on the Total μ–Net implementation. In effect, the performance of the Total μ–Net is compared to a Total μ–Net without recursive accommodation, called *Total-Recursive*, and to a Total μ–Net without either recursive or linear accommodation (i.e., Branching μ–Net without linear accommodation), called *Branching-Linear*.

Now that reasonable values for limb length (Table 6.7), and branch degree (2)have been measured, μ–Net's channel utilization can be evaluated. μ–Net utilization is determined by the miss rate of the μ–Net anticipation mechanism; Equation 6.1 describes

the performance measure, and Equation 6.2 is the miss probability component which has been simplified for a branch degree of 2.

μ–Net implementation	Limbs delimited by	Mean L	Median L
Branching-Linear	BRANCH, CALL, RETURN, JUMP, INDIRECT	7	4.3
Total-Recursive	BRANCH, CALL, RETURN, INDIRECT	8	4.75
Total	BRANCH, INDIRECT	10	5.8

TABLE 6.7
Mean and median limb lengths for μ–Nets implementing branching

Equation 6.1: $$T = N * \left(t + \left(1 - \frac{\log_D \left(\frac{r * OP_BW * (D-1)}{L * D} + 1 \right)}{\frac{r}{t}} \right) * r \right)$$

where OP_BW = opcode bandwidth, i.e., opcodes per unit time
D = branch degree (usually 2)
L = limb length, in number of opcodes
N = total number of opcodes executed
T = resulting execution time
r = round trip time
t = time to execute an opcode = $\frac{CPI}{cpu_speed}$

Equation 6.2: $$\mu Net_miss_rate = \frac{\log_2 \left(\frac{r * OP_BW}{L} + 1 \right) - 1}{\frac{r}{t}}$$

The anticipatory miss probability is determined by the sum of the probability of an INDIRECT opcode occurring (0.3%) and the probability that the Code Pump has

insufficient bandwidth to send all the possibly requested opcodes. Performance is proportional to the channel utilization, which is proportional to hit probability, which in turn is equal to 1-miss_probility. The performance of the three variations of anticipation (Branching-Linear, Total-Recursive, Total) can be plotted vs. bandwidth (*OP_BW*) round trip latency (*r*), with branch lengths as indicated above (Table 6.7). Figure 6.10 describes the Branching-Linear μ–Net (limb length L=7), Figure 6.11 describes the Total-Recursive μ–Net, and Figure 6.12 describes the Total μ–Net. These figures allow comparison of mean limb length values.

The graphs show how performance (i.e., channel utilization) decreases as latency increases, and increases as bandwidth increases. They also indicate that small increases in average limb length, as derived from more complete implementations of the Code Pump, achieve large increases in utilization. *Increases in bandwidth compensate for increases in latency, although a linear increase in latency requires a corresponding exponential increase in bandwidth (see shaded areas).*

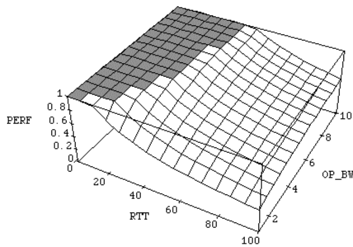

FIGURE 6.10
Mean limb length (L = 7) hit probablity vs. BW vs. rtt.

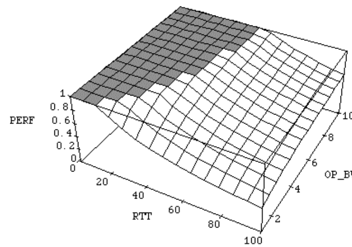

FIGURE 6.11
Mean limb length (L = 8) hit probablity vs. BW vs. rtt.

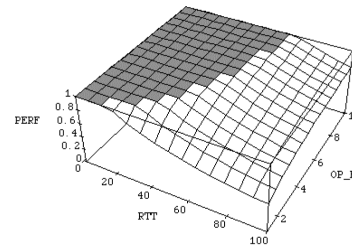

FIGURE 6.12
Mean limb length (L =10) hit probablity vs. BW vs. rtt.

μ–Net anticipatory hit probability vs. bandwidth (OP_BW) and round trip latency (rtt)

In order to better compare these three graphs, 2-dimensional slice of each graph were plotted together. μ–Net anticipation channel utilizations are compared for fixed values of bandwidth (OP_BW = 2), and for the mean and median values of branch length (Table 6.7). μ–Net is compared against a non-anticipatory protocol in which only a fixed deterministic prefix (of length 7) can be prefetched. The non-anticipatory protocol is denoted by the solid curve, and Branching-Linear, Total-Recursion, and Total μ–Net

versions achieve increasing channel utilizations, as indicated by wide-dashed, narrow-dashed, and dotted lines (Figs 6.12, 6.13).

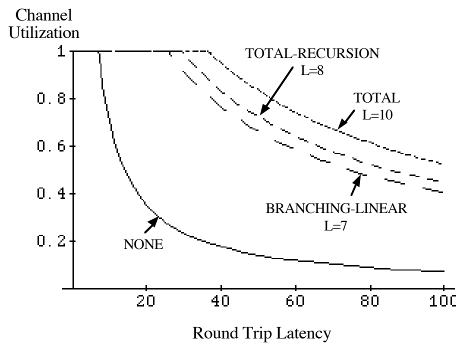

FIGURE 6.12
Mean limb length rtt. vs. utilization

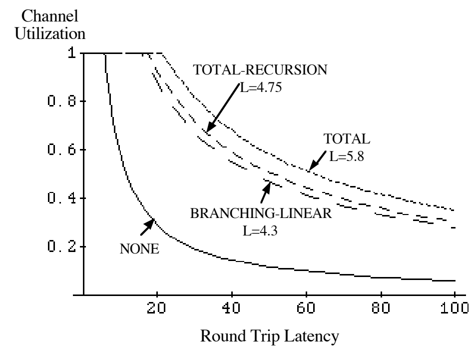

FIGURE 6.13
Median limb length rtt. vs. utilization

A view of the above, for OP_BW=2 (mean, median, for each of 3 implementations)

Examination of the ratio of the μ–Net channel utilizations to the non-anticipation prefetch shows the utilization increases possible (Figs. 6.14, 6.15). *The initial utilization increase is linear, representing the μ–Net anticipation of the entire tree of possible opcode streams. There is a point in the stream when the entire tree cannot be transmitted given the available bandwidth; at this point the utilization increase is logarithmic as latency increases.*

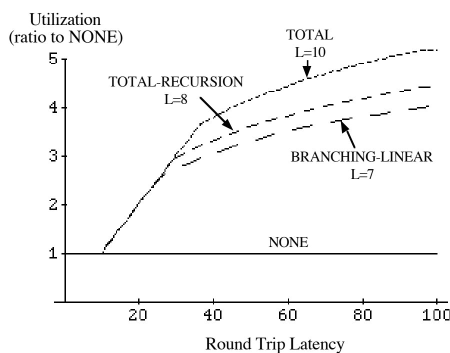

FIGURE 6.14
Mean relative performance

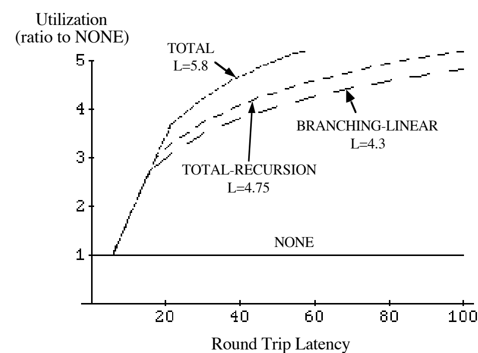

FIGURE 6.15
Median relative performance

Relative performance, vs. non-anticipatory implementation

There may be optimizations to the Code Pump which would further constrain the size of the TreeStack. In some cases, especially those of short branch jumps, the state space collapses back down. If the set of all addresses currently in transit is known, further messages with those addresses may be omitted. Consider the opcode sequence in Figure 6.16.

FIGURE 6.16
Short forward branch opcode sequence

The PC image, after the execution of the conditional, evolves into two identical but temporally shifted sub-images. One image expresses the branch not taken, the other expresses the branch taken. It would be more efficient for the Code Pump to recognize this shift, sending the opcodes as if the branch were not taken, thus covering all code for both branches. The drawback in this scheme is that the opcodes must be scheduled as if all were executed; they cannot be interleaved as if they were two mutually exclusive opcode sequences, each at the rate of execution, resulting in a higher sent rate (utilizing high bandwidth). As a result, there are times (when the branch jump is taken) when the processor is idle, but this is still often less than the time which would have been incurred by not implementing the branch as a sent message.

For example, if the round trip time is 100 instruction execution times long and the branch jump is less than 100 opcodes long, the above simplification, of sending opcodes as if no branch occurred, remains more effective than waiting for a round trip time before replying with the opcodes. This simplification further reduces the size of the image in the Code Pump, reducing its storage and computational requirements.

A proper analysis of the effective size of the TreeStack structure will require an emulation of the architecture because there are many dependant characteristics of the code stream which affect the results. These include the way in which the size of the TreeStack structure depends on the number of pending branch operations down all possible computational paths, and the ways in which the existence of a code cache at the processor affects the distribution of opcodes requested from memory. An emulation of the TreeStack in the Code Pump was beyond the scope of this dissertation.

6.3. Observations

There are some observations which were discovered through the process of casting the processor/memory communication architecture into the Mirage model. These include a better understanding of the ways in which opcode classes affect the communication channel, and the ways in which the μ–Net solution fails and the reasons why it fails.

6.3.1. Kinds of instructions

Opcodes have been partitioned into classes based on the ways in which they transform the image of the PC in memory. Most opcodes only increment the current PC, and so transform the PC image by moving it through state space in a linear fashion. Other classes of PCs transform the space non-linearly, either by a simple function (old PC → new PC), by a function with memory (old PC → new PC + state memory), or by a set mapping (old PC → set of new PCs).

6.3.1.1. Regular opcodes (OTHER) and JUMPs

Usually the PC image is transformed by a simple mapping function, e.g., by the addition of a constant to the current PC values. For regular opcodes (i.e., OTHER), the constant is implied and fixed, with a value of 1 (i.e., to increment the PC); for JUMPs the constant is contained in a field in the opcode, and is called an *offset* (PC-offset JUMP). In other kinds of JUMPs, the PC values can be completely replaced by the fixed constant in the opcode (direct JUMP).

These opcodes also perform a function in terms of the ways in which they alter the path traversed in state space. Regular opcodes provide the default traversal of the state space, a linear sequence. JUMPs permit discontinuities in the path, and are means of permitting paths to evolve which are not dependant on the natural ordering of the space. Backward JUMPs provide only for code reuse because the code could have been repeated at the location of the JUMP. Forward jumps skip over code which must be otherwise accessible, i.e., they permit use of places in the state space which other paths otherwise ignore. JUMPs permit the efficient use of a linear address space to contain a nonlinear path.

6.3.1.2. BRANCHES

Branch transformations are conditional upon the state of the processor. The PC is incremented by a value which depends on whether the branch is taken; if it is, the increment is contained in the opcode, if not, the increment is fixed and has a value of 1. Branches behave as either a regular opcode or as a JUMP, depending on the processor state. Memory in μ–Net does not model the entire processor state, only its PC value; as a result, BRANCHes induce an ambiguity in the transformation upon the memory's image of the processor's PC.

The processor PC undergoes one of two transformations; here modeled as a set mapping from one PC value to two, so that the size of the PC image doubles as a result. The particular resulting PC remains unknown, but it is one of a finite and explicit set, so the transformation is modeled as an expansion of the image.

Later, when the processor communicates the need for an opcode that is on the resultant path of one of these PC values, the other PC path precedents can be omitted. As memory sees it, the PC image becomes two images, which are collapsed down to a single image after later communication.

The communication which collapses the image can be made explicit, by making the Filter Cache encode and send only branch decisions to the Code Pump, in which case the reduction in the size of the image would be that proscribed by the communication limit (Chapter 2). Instead, the interface is simplified by having the Filter Cache transmit entire addresses (PC values) to the Code Pump, where the decision as to how the image is affected is subsequently determined.

BRANCHes implement a level of context sensitivity, where the actions of the Code Pump are affected by some portion of the state of the processor not modeled. This is a minimal context sensitivity because PC image is partitioned into two resultant PC images as the result of the branch execution. BRANCH context sensitivity derives from the implementation in code of Dijkstra's guarded commands [Di76]; communication stream context sensitivity parallels BRANCHes in Mirage, via *guarded messages*.

6.3.1.3. INDIRECT

Indirect opcodes provide a level of context sensitivity which may be too powerful for most intents. A PC value is transformed by the execution of the indirect instruction into *any* PC value. There is no information in the opcode which restricts this transformation, as occurs in a JUMP or BRANCH (because they contain fixed PC offsets

or fixed new PC values). Whereas a BRANCH is modeled as transforming a PC into one of two resultant PCs, an INDIRECT opcode transforms a PC into the set of *all* PCs.

The net effect of an indirect opcode is to partition the PC image into an infinite number of components, each of which may be the result of its execution. Actually, because the size of a PC is fixed, the size of this set is also fixed, but the partitioning results in a set of SIZEOF(PC) components, which is effectively infinite to the μ–Net model.

In attempting to model the transformation of its PC image by the transmission of an INDIRECT opcode, the Code Pump fails. It cannot sufficiently partition the resulting image (which encompasses the entire PC state space), in order to send messages (opcodes) pertaining to portions of it. Indirect opcodes introduce too much information to model effectively. The Code Pump is required to wait for a message from the Filter Cache, indicating the destination of the indirect opcode. An indirect opcode expands the PC image to completely cover the state space, so the size of this message must be the log of the size of the space (i.e., the size of the PC value), in order to collapse the image down to a single value.

Indirect opcodes permit the PC path to be completely dependant on part of the state of the processor which is not modeled, which means that a round trip time is required in order to obtain that (unmodeled) state information.

These assertions require qualification. As mentioned, there is no information in an INDIRECT opcode, or in the entire opcode stream, which limits the destination of an INDIRECT offset, other than to the raw size of the address (i.e., the entire PC space). In some compilers, INDIRECT address destinations are restricted to explicitly labelled destinations. In this case, μ–Net would suggest that the INDIRECT jump be compiled into a table lookup of existing label destinations, i.e., a fixed JUMP handler. Using such a JUMP dispatch table, the INDIRECT jump's potential destinations are reduced from infinite (or SIZEOF(PC)) to some small constant. This table also permits the Code Pump to provide anticipation of the fixed branching of the data stream, rather than being thwarted by the (effectively) infinite branching of INDIRECTS.

6.3.1.4. CALL and RETURN

CALLs and RETURNs are, respectively, entry and exit points to recursive state space instances. A CALL behaves like a combination of an entry point to this space (in terms of its effect on the stack in both the Code Pump and at the processor), and a JUMP,

in the way it specifies an address at which to enter this recursive space. Within the recursive space, most opcodes transform the space as usual, not affecting the parent space (i.e., the space of the source of the CALL opcode); a RETURN is the only exception, it being the lone exit point from that space (back to the parent space).

Subroutines, as facilitated by CALLs and RETURNs, provide a structure for code reuse, which does not affect this analyses. They also permit a spatial recursion which is relevant because our modeling of them emulates this recursion in the management of the PC image, via the TreeStack structure.

6.3.2. Other observations

There are a few other observations that can be made. Comparing our method of sender anticipation in the Code Pump vs. the anticipation made by prefetching of branch targets is equivalent to comparing breadth-first search (BFS) vs. depth-first search (DFS) techniques. The BFS method utilizes increased bandwidth requirements of the μ–Net domain, and also avoids the need to backup or cancel the code search, because opcodes are sent as isopotent sets (each level of the BFS).

Also, μ–Net has a problem with pointers. Fortunately, it is not alone in this regard; pointers also inhibit many types of static code analysis as well. Indirect opcodes are code-space pointers, and these opcodes are advantageous only if they outweigh the latency they necessarily induce. Opcodes should be limited to only the required degree of branching of the code stream, i.e., limit indirection to multi-way jumps where possible.

μ–Net indicates the difference between sender-anticipation and prefetch caches. The latter assumes a correlation between past and future code use (via the cache), and anticipation assumes infrequency of some types of control opcodes. μ–Net system uses semantic information about the traversal of the code space, i.e., by actually determining the effect of each opcode on the code space sequencing (i.e., PC values).

6.4. Relation to prior work

μ–Net is a unique system of anticipation, derived from the abstract Mirage model of communication in the presence of latency. Since its invention, the literature has been investigated for comparable processor/memory mechanisms, and very few of these references were applicable.

An analog to the Code Pump was investigated in early microprocessor research, which has since been abandoned (Instruction Issue Logic). Current trends in system architecture research have returned to this area (SPA), but use a restricted form of μ–Net's TreeStack structure for control. Prior research, both past and present, has been developed from empirical evidence alone, whereas μ–Net was derived from the abstract Mirage model.

6.4.1. Instruction Issue Logic (Code Pumping)

'Instruction-issue logic' is the closest precursor to the Code Pump mechanism in μ–Net. One of the first 'instruction-issue logic' implementations is the Fairchild F8 system (1976), another more recent investigation was the Distributed Logic Instruction Issue System (1987). Most of these schemes differ from μ–Net's Code Pump in that multiple branches are not accommodated.

The most recent version of this research is the Sustained Performance Architecture (1991), which is guided by the need to investigate active memory architectures (coined *proactive* therein). All of these versions of implementation can be considered restricted instances of the more general μ–Net concept, however none mentions such an abstract version. There are also ways of extending each of these implementations to accommodate opcodes not currently handled, which would not complicate their design, and were not investigated.

Further, μ–Net is derived from the consequences of applying the Mirage protocol model to the processor-memory interface domain. All these prior designs were created via other means, i.e., engineering issues such as simplified system design (F8), or arbitrary visions of appropriate memory interfaces.

6.4.1.1. Fairchild F8

The Fairchild F8 was a microcomputer system based on a 3850 CPU and 3851 Program Storage Unit (PSU) [Fa76b],[Fa76c],[Os76]. Its goal was to reduce the overall chip count, from 7 in a typical Intel 8080A or Motorola 6800 design to 2, at the expense of partitioning the CPU according to the complexity of its components into a separate ALU and instruction sequencer. Other comparable microprocessor chip sets were partitioned along functional lines. The F8's low chip count helped it lead world sales of CPUs in 1977, when it became a dominant embedded controller in consumer products.

The F8 chip set was complemented by the Mostek 3870, a single chip combination of the 2-chip set which was not extensible, but isolated the CPU/PSU interface from the designer and was thus more accepted in competition with traditional microprocessors.

Although the F8 was first noted as an "offbeat product with a strange set of chips and a ridiculous instruction set"[Fa76c], it is examined here for its CPU/PSU interface, especially in its similarity to μ–Net. The 3870 CPU has no program counter, as in current CPUs; instead, the function of instruction sequencing has been relegated to separate devices on the bus, implemented by instances of the 3851 PSU. The advantages of removing the memory access logic from the CPU are reduced pin count (no address lines needed), and availability of additional real estate on the CPU.

The communication between the CPU and PSU is provided by a set of control lines, ROMC 0-4, and a pair of clock signals. Each PSU has an internal PC, and a factory-preset mask (upper 6 of 16 bits). The PC in each PSU contains the PC that would have occurred in the CPU, at all times. Operations which affect the PC, i.e., load the PC from the data bus, write the PC to the data bus, increment the PC, add the offset on the data bus to the PC, etc. affect all PCs in all PSUs identically. When the CPU, by the control lines, requests an opcode, any PSU whose mask matches the PC's high bits responds by placing the opcode on the data bus.

This can be interpreted as hardwiring each PSU to respond only to its own mapped address space, even though the entire PC is imaged in every unit. The internal mask acts like an address decoder and tri-state output enable together.

The similarity to the μ–Net design is obvious; each memory unit has its own Code Pump (PSU), which models only increment transformations. Other transformations occur only in direct response to messages from the CPU. The PSUs need only model a single point in state space (i.e., a single PC value) and send a single opcode out, because latency is not a design criterion. Recursion of the state space image is modeled, but only one level of pending recursion; this is sufficient because it was intended for interrupt servicing (where interrupts are not subsequently interruptible, in this design), and it was assumed that further PC storage to accommodate recursion of the normal execution would be provided in software, i.e., in external storage which was not part of the PSU design.

6.4.1.2. Distributed Logic Instruction Issue

The Distributed Logic Instruction Issue System (DLII) [Ha87] implements a Code Pump where conditional branches are modeled, as well as regular (OTHER) opcodes. Only one pending branch is permitted, and opcodes are buffered at the processor, rather than in the communication stream. The design was intended to accommodate access latency induced by bus contention, rather than transmission delay. The system can also be interpreted as a prefetch, where both alternates of a branch are prefetched.

The intent was to move instruction logic to the memory module, as was done in the F8 (on which the idea was based). In contrast to μ–Net, no instruction addresses are transmitted, this being substituted by a bit indicating to which arm of a conditional each opcode belongs (branch taken/not taken). This implements the minimal encoding of message guards because the PC image space is permitted to grow to only two points in space. Consequently, only one bit of reply information from the processor is required to collapse the space to the PC actually used.

DLII partitions opcodes into 5 classes: JUMPs, CALLs, RETURNs, BRANCHes, and regular (OTHER) opcodes. INDIRECT opcodes, and their effect on the instruction stream, were not considered separately. It also recognized the ability to predict the successor to JUMPs and CALLs, and also RETURNs, the latter-most with the addition of a stack in the program storage module (PSM, akin to our Code Pump/program storage). The PSM stack implements the Recursion μ–Net, with one addition.

The PSM also contains two PCs, the set of which is similar to a limited version of our PC image, as maintained by the TreeStack structure in the Code Pump. The instructions accumulate in a dual buffer in the CPU, which executes instructions from one side of the buffer at a time. When a branch is indicated, the CPU switches to the indicated buffer (i.e., either remains or switches), and clears the contents of the alternate, because those opcodes are no longer needed. The PSM is notified of this selection, and the unused PC is cleared and released for reuse.

Subroutine return addresses are contained in the PSM and only the PSM and CPU need know of their existence, so that no other copies of the stack exist. The PSM transmits both the RETURN opcode and the destination address, in effect, translating the opcode into a direct JUMP. This may be a useful modification to the Code Pump design, although it requires modification of the CPU design. Proposed μ–Net designs do not require CPU modification.

The PSM also recognizes short loops, such that repetitions of instruction sequences that are capable of being held completely in the CPU cache are omitted. This is a useful optimization of the Code Pump, but necessitates a more intelligent (or Code Pump/Cache aware) compiler. Again, although there may be benefits, the optimization requires that the compiler writer be familiar with the architecture. μ–Net performs the same functionality and also permits the compiler to be independent of the mechanism (the mechanism is hidden in the abstraction).

Similar to μ–Net, the PSM assumes its memory is read-only code. Opcode loops are permitted to have only one internal branch, or a branch which may jump to the opcode immediately following the loop only; if these conditions are violated, the instruction issue waits. If either instruction stream (of the two possible, emanating from the two possible PC values), encounters a loop start (i.e., set of instructions assumed to remain within the cache), call, return, or subsequent branch, the instruction issue again waits until the opcodes at the CPU are completely consumed.

The restriction that only one return address stack is managed, and that only one pending branch is permitted (and only at the top of that stack), means that here the TreeStack data structure is reduced to a simple stack. The internal entries of the stack contain single PC values, whereas the top of the stack is contained in the pair {PC1,PC2}, i.e., a stack whose top element *only* may contain a tree with one branch *only*.

The analysis of the architecture is similar to that of μ–Net, although indirect opcodes are not considered. Their effect on the architecture is identical to that of loop starts, jumps, etc. in the presence of a pending branch - that the PCU waits until the opcode in question is executed by the CPU before proceeding further. There is no mention that indirect opcodes by their nature require such latency, whereas the latency incurred by other opcodes in this design are the result of the limited implementation of the TreeStack structure.

PSM proponents further acknowledge the speedup that is the result of the parallelism in the anticipation mechanism in the PSU being decoupled from the CPU. Additionally, communication between the PSU and CPU is reduced by as much as half because addresses need not always be sent with the opcodes from the PSU. It is not clear that a complete instance of the TreeStack structure would result in a similar bandwidth savings, because μ–Net's message guards are complete addresses to simplify implementation. The communication per opcode is reduced by half, and the resulting bandwidth (and any additional available bandwidth) is used by the transmission of two

code streams (i.e., two alternative opcodes). This shows a method where the necessary additional bandwidth required is created by the design, whereas μ–Net assumes a further bandwidth surplus exists.

6.4.1.3. The Sustained Performance Architecture

There is an excellent overview of existing instruction sequencing methods which has just recently appeared in the literature [Kr91]. These include methods of prefetching, branch prediction, and cacheing. In the conclusion, the authors allude briefly to their vision of the future of instruction sequencing, where they define *proactive* memory, where "...memory should produce the correct instruction before the execution unit needs it...".

In the event of concurrent potential opcode sequences, "...the program-flow graph can be used during program execution to initiate the prefetch of *all* instruction sequences...". This architecture has been named the "Sustained Performance Architecture." [Do89]

The SPA paper describes the conditions under which opcodes can be predicted by memory, and notes the need for a stack to store return destinations, and also notes that indirect opcodes require round trip communication regardless of any type of prediction. The goal of this architecture, as in μ–Net, is to avoid main memory access latency.

They do not discuss the complexity involved with the maintenance of the PC image in their version of the Code Pump, the Instruction Decode Unit (IDU). They acknowledge the need for a stack structure to store return destinations, but do not note the interaction between pending conditional branches and levels of recursion.

They claim a 30-40% increase in processor performance, for the levels of latency they envision (about 2-8 opcodes), and also claim that sequences of opcodes without control flow changes have an average length of 7 and a median length of 4 (this includes branches not taken). Our measurements differ, but not substantially (average 6, increasing to 9 if jumps, calls, and returns are not considered control flow changes, with respective medians of 4 and 6). They also achieve multiple instruction streams by replication of the IDU; the problem with this design is that an IDU is associated exclusively with a single opcode sequence. Their Program Execution Controller (PEC) is charged with the task of controlling this exclusivity, prefetching the actual instructions from memory. In essence, the PEC is our Code Pump, and each IDU manages a stack in the TreeStack, which is

here restricted to a tree of stacks (i.e., a tree whose elements are stacks). The wiring and control of the IDUs implements a hardwired tree structure.

This is, therefore, a more restricted form of the general TreeStack structure, because it cannot handle branching subsequent to recursion. A branch instruction causes one side of the branching stream anticipation to be relegated to a separate IDU, thus partitioning the anticipation between two IDUs; subsequent rebranching causes the IDUs to organize into a tree structure, as in the TreeStack. This opcode organization is common in OS handler subroutines, a monitor program, or a dispatcher. It is a reasonable tradeoff, but the general TreeStack structure is not acknowledged in the SPA, nor are the reasons for its existence.

Once recursion occurs, a stack is required. In μ–Net the stack structure would be analogously created by adjoining a stack of IDUs, whose interior elements would be inactive. SPA notes the necessity for a stack to manage recursion, and for a tree structure of IDUs to manage branching, but does not indicate where the stack would exist. Were the stack within the IDUs, the occurrence of a CALL would inhibit subsequent branching, because such branch alternates could 'pop' at different times. This would cause inconsistencies in the IDU stack. μ–Net explicitly specifies the structure of the mechanism required to handle branching and recursion together.

Further, the PEC is guided by the program call graph, which is loaded at run time. This introduces another level of complexity, especially if two concurrent processors attempt to use the same PEC, or if the call graph of a program were superseded by the call graph of the interrupt handler, or operating system, or other intervening code. μ–Net determines all requisite control information from the instruction stream itself, and can dynamically reallocate resources for multiple pending branches because it is monolithic in design, and manages the data structure centrally.

The SPA appears to be an empirically derived, specific instance of the more general, more abstractly derived μ–Net architecture. This research, although state-of-the-art for architecture design, is presented with no abstract basis. μ–Net was developed concurrently to and without knowledge of the SPA, as the logical consequence of the protocol interaction between the processor and memory and the limitations of such communication as latency grows.

6.4.2. Cache issues

As discussed before, there are similarities between some cache issues and our Code Pump, especially if the Code Pump is viewed as complementary to a cache, with which it should be coupled. Some similarities exist between the anticipatory nature of the Code Pump to that of cache prefetching, as implemented explicitly in hardware, or in software, or as partially effected by widening cache lines.

6.4.2.1. Prediction/prefetching

As was discussed in the general Mirage model, the expansion of state spaces can be further specified by attaching a probability distributions function (PDF) to the expansion. If guarded messages are sent, utilization of the communication channel can be optimized by sending messages which affect the most likely subspace, as indicated by the guard. The limit of this analysis is explicit prediction, i.e., assuming all branches are taken, not taken, or that their behavior is static over time (i.e., if a branch was taken last time, it will be taken this time). The assumptions of probability behavior and extrapolation from a set of events to a future event depends on whether the average is temporal or ensemble, as discussed earlier.

As also mentioned before, ensemble averages predict individual behavior from behavior of a group, i.e., within a single execution of a program, if most branches are not taken then it is likely a given branch is not taken. In branch prediction, this translates to the assumption, given statistics gathered over all executions, that branches in general are not taken, and governs the use of that assumption in the absence of temporal information.

Temporal averages predict future behavior of a branch based on its own past behavior, which may have been measured during prior executions or at some earlier time in the current execution. Temporal average information translates into the assumption that whatever way a branch went in the past, it is very likely to do the same in the future. Tag bits associated with each branch encode the last path through it, and are used by prefetching mechanisms to predict the current most likely path. This information usually encodes the last two execution paths because it would be undesirable to override the dominant probability due to a single prior event of lower probability.

Branch prediction achieves [Le84] 90-95% accuracy, using temporal average information. Indirect jumps can also be predicted, using temporal averages only because

ensemble averages are meaningless. These are less effective, achieving between 50-75% [Wa91] accuracy, but these estimates are completely useless if the prediction fails, because failure does not imply a result address.

6.4.2.2. Wide lines

Wide cache lines have a similar effect to linear prefetching. Consider a line size which accommodates 4 opcodes. A miss on an opcode fetch from the cache generates a request of that opcode and the opcodes that surround it in the line. If the instructions execute linearly, the miss on the first opcode causes the next 3 to be fetched as well.

The normal cache fetch of one opcode causes a prefetch of the next three opcodes. Such a mechanism would be optimal where the prefetched opcode sequence was at least as large as the expected length of a linear sequence of opcodes; this is borne out by empirical results, as the analysis of cache line sizes shows the most effective size between 4-16 opcodes [Sm82], whereas some measurements claim a mean linearity of 6 and a median of 4 [Kr91], and our results indicate a mean of 7 and a median of 4.

Cache line size is related to prefetching, but a prefetching cache is more effective than an equivalent non-prefetching cache with twice the line size. Doubling the line size is beneficial only where opcodes execute in linear order, otherwise the width of the lines causes larger sets of opcodes to be swapped out when a miss occurs. In addition, increasing the cache line size only prefetches within the line (i.e., prefetches the subsequent opcodes from the head of the line), which indicates that a miss penalty is incurred when each new line is loaded. Prefetching can retrieve the next cache line when the last opcode in an existing line is used, removing this penalty.

6.4.2.3. Software

Software prefetching has also been suggested as a means to avoid memory access latency [Ca91]. The system assumes a model where the processor prefetches its own opcodes (as in standard caches and prefetching caches), rather than having them sent ahead by the memory (as in μ–Net). It differs from both regular and prefetching caches in the way in which opcode fetches are guided.

Regular caches reuse code already executed, and prefetching caches retrieve opcodes in linear sequence only. Software prefetching uses compile-time information to anticipate the future needs of the processor, and initiates the prefetching by an explicit

instruction. Pseudo-opcodes in the stream control the prefetch mechanism, and are executed by the cache mechanism, and never reach the processor.

Software prefetching has been suggested as a general scheme for guiding prefetching. A more specific kind of software prefetching was developed for the TI-ASC supercomputer [Wa72]. Its compiler (for FORTRAN) generates *prepare-to-branch* instructions, in order to quicken loop execution.

By comparison, long cache lines are an implied prefetch, but sustain a hit penalty when the first item in a line is accessed, and prefetch only where access proceeds in sequence. Hardware prefetching, as it is commonly implemented, fetches a line whenever the address preceding that line is accessed, i.e., whenever address (i) is accessed in the cache, address (i+1) is tested and fetched if missing. The hardware avoids the hit penalty of long cache lines, but again works only where accesses are sequential.

Software prefetching avoids the need for sequential access as a precondition to successful prefetching. The compiler initiates a fetch in advance of code use by inserting explicit cache load instructions; this works because it has prior knowledge of static code sequences, regardless of their order, at compile time. As a result, an execution cycle is spent communicating an upcoming sequence of address accesses.

The mechanism requires a cache supporting a list of pending access requests, which in our system corresponds to the sequence stored in the round trip latency. This form of prefetching has been implemented only for data accesses, where 'DO-loop' structures inform the compiler of data access patterns which can be easily translated into prefetch instructions. Software prefetching can be applied to code prefetching as well, using a static opcode lookahead in the compiler (i.e., peephole optimizer). This would not exceed μ–Net's performance, as μ–Net can look ahead into the instruction stream at runtime almost as easily as software prefetching can do at compile time, and μ–Net incurs no runtime penalty for prefetching. Software prefetching incurs a penalty of one instruction execution time for each prefetch, in order to communicate the prefetch information which μ–Net extracts from the code sequence at runtime.

6.4.2.4. Prefetching vs. cacheing

Prefetching has been compared to caching, although the two can be complementary, because cacheing assists in reducing latency in accessing instructions already issued, whereas prefetching assists during its first use. Another way to view the comparison is that caches look into the past, whereas prefetching looks into the future.

Cacheing and prefetching have been compared as alternates [Le87]. In the cases measured, prefetching performs at least as well as cacheing, in cases where memory access has high latency.[1] This makes sense because caches cannot remove the first hit penalty, whereas prefetching can work both in first use and reuse cases.

6.4.2.5. Prefetch v.s pre-reply

Prefetching is distinct from the pre-reply proscribed in μ–Net. Prefetching is processor-directed, receiver based anticipation, whereas pre-reply is memory-directed, sender based anticipation. Some versions of prefetching [Le87] perform as detailed a management of future state as the Code Pump, but there are several reasons for our placing the Pump at the memory side of the communication channel.

μ–Net's partitioning permits a reasonable distribution of work. The anticipation mechanism is off-loaded from an already overloaded side of the channel; other research in general protocols (e.g., the Universal Receiver Protocol [Fr89]) suggests that such balanced systems are more effective because neither side of the mechanism is unduly overloaded.

Also, receiver-based anticipation involves twice the managed lookahead of the sender-based equivalent because the sender can collapse half the lookahead into the image. In effect, the receiver would work with a single large tree of possibilities, whereas the sender works only with parts of some branches.

6.4.2.6. Guarded messages

Guarded messages permit multiple streams of opcodes to be issued by the memory, where only one stream is actually used. A similar form of conditional labelling of opcodes occurs in pipelined processing, where "conditional branches can be converted to guarded jumps..." [Hs86], and store instructions are converted to guarded stores, to avoid delaying the store in the pipeline.

These techniques are usually associated with pipeline scheduling, either with additional hardware support or compiler participation. Guarded communication differs

---

[1]Clearly caches outperform prefetching where memory access is restricted by bandwidth, because cacheing reduces memory bandwidth use, whereas prefetching increases memory bandwidth.

from guarded execution, but the principle of using extra power (bandwidth or pipeline stages) to overcome latency is the same.

6.4.3. Other related architectures

Prefetching has been examined in other architectures. The IBM Stretch implements branch prediction, where failed predictions are backed over later on. The IBM 360/91 prefetches both arms of a conditional, but down only 2 branch arms, and neither prefetched arm is executed.

Prefetching down both arms of a conditional branch has been called "branch bypassing" and "multiple prefetching." [Li88]. Predictions from existing code estimate that branch bypassing benefits vary with the level of bypassing performed [Ri72]. Code execution is speeded by a factor of $\sqrt{j}$, where j pending branches are bypassed, at a cost of 2^j. These performance increases are the result of empirical estimates, based on FORTRAN and CDC-3600 assembler, and are not the direct result of utilizing latency for prefetches. The more primitive languages tested in this research exhibit branching largely as the result of loop statements of explicit conditionals in the source code, rather than being generated by the compiler to express complicated nested structures or data access mechanisms.

6.4.3.1. IBM Stretch (7030) - one of the first prefetch

One of the first computer architectures to implement opcode prefetching was Project Stretch, which resulted in the design of the IBM 7030 computer [Bu62]. At the time (1955), IBM had produced several computers, including the 650, 704, and 705. This project was intended to stretch (thus the name) the capabilities of the existing technology, in order to design a computer with a performance of 100x its immediate predecessor.

The IBM Stretch has several similarities to the methods shown here. It had two prefetch units, one performs operand prefetch up to 6 instructions linearly consequent to the current PC, which are then in various stages of decoding. The other, called the Lookahead Unit, prefetches operands for up to 4 of these opcodes, and provides the required interlocking to maintain execution integrity.

Thus the Lookahead Unit prefetches data, whereas the instruction unit prefetches opcodes. The instruction unit models branch instructions as regular opcodes, and forces a flushing of its cache when this assumption fails (i.e., when the branch is taken, because it

assumes branches are not taken). This system was designed to compensate for the disparity in access time of the memory and execution time of the CPU. The high memory delay was due to the storage technology of the time (2.1 µs core memory, 1.5 µs add time via 10-20 ns gate times), and the propagation time through a series of switches and registers which manage access to components (much like a single bus controller).

The addition of prefetching was examined, varying the amount of prefetch [Bu62]. The result was a marked performance increase out to a prefetch of 8, with further prefetching of little advantage. This result is similar to our analysis, which concluded that prefetching was useful only to the average (?? expected) branch arm length. The performance increase found with prefetch was between 20% and 200%, depending on the benchmark application measured.

6.4.3.2. IBM 360/91 - dual prefetch

The IBM 360/91 [An67] instruction unit attempts to fetch opcodes faster than they are used, so that when branches occur, the gap can be accommodated by emptying the buffer while the branch is resolved. The design goal was a buffer (linear lookahead) of 6 instructions, because memory access is 6x slower than instruction execution; this goal was extended to a 10-instruction prefetch on startup, and the implementation provides for a total of 16 opcodes of prefetch.

Branches in the opcode sequence cause both paths to be prefetched: the not-taken path continues down the lookahead of 16, whereas the branch-taken path is prefetched with a lookahead of 4. Forward branches are handled this way, as are backward branches beyond the scope of the lookahead buffer. Backward branches which refer to existing lookahead entries (i.e., which refer to targets less than 15 opcodes away), signal a 'loop mode' in the prefetch unit, and instruction prefetching ceases. The loop mode permits unimpeded use of instructions in the prefetch buffer, providing enhanced performance for some code; this feature occurs in the cache and instruction fetch units of some current microprocessors, including Motorola's 680x0, (68010 and later), and latter versions of Intel's 80x86 CPUs as well.

The 360/91 thus implements a Code Pump where regular opcodes are modeled, as well as 1 pending branch. The TreeStack structure consists only of the tree component, whose main trunk (root trunk and major branch) are limited to a total length of 16, and whose minor branch arm is limited to a length of 4. Other opcodes halt the prefetching

because they are not modeled; this includes jump and call instructions which could have been modeled with only an additional adder in the instruction unit, as in μ–Net.

6.4.3.3. Rope multiple prefetch

Pipelined and very long instruction word (VLIW) CPU research has lead to the need to overcome memory latency as well. The Ring Of Prefetch Elements project (ROPE) is aimed at alleviating memory latency and the difficulties in prefetching caused by conditional branches, without the need for complex hardware scheduling [Ka86], [Ka85]. It also provides a new prefetch mechanism which supports multi-way branching, which we discussed in μ–Net as a viable alternative to most uses of indirect opcodes.

ROPE recognizes that "caches offer no speedup for loops larger than the cache size," and so looks for other ways to alleviate memory latency. They abandoned the μ–Net methods, claiming that the prefetching of all branch destinations is prohibitive in communication costs; we accept that conclusion because we are designing an architecture where the communication bandwidth is very high.

ROPE reduces memory latency and removes the need to flush the pipeline when conditional jumps fail, because each opcode prefetch path is supported by a separate prefetch unit. Control passes cyclically around the ring, where each opcode is fetched in turn by its corresponding unit. Explicit prefetch instructions (similar to those of software prefetching), initiate the fetching of branch targets in two or more prefetch units. When a branch occurs, control can pass either to the next unit in the cycle, or it can jump to the unit fetching the destination of the branch; control is automatically assumed by the appropriate prefetch unit because all scan their condition masks to the datapath, and only one subsequently replies with an opcode.

ROPE handles regular opcodes, and binary branches (multiway as well, with some modifications). Jumps and calls are not handled, although they could have been. There is no local memory to the prefetch element ring because it would have to be distributed among the elements but commonly accessible, so a shared or central stack cannot be managed, and return opcodes cannot be accommodated.

The utility of multiway branching is supported by the measures that code is 15-33% branches, and many optimization algorithms (trace scheduling, and percolation scheduling) tend to cluster these decision points. The ability to evaluate simultaneously several binary branches in a multiway branch can reduce the branch occurrence by as much as the fanout of the multiway, in some cases.

The ROPE research measured an increased execution rate of up to 5x conventional architectures, which is consistent with a 6.3x increase predicted for architectures which prefetch only regular opcodes (Unit Linear μ–Net, see Table 6.2).

Both the ring architecture and cyclic transfer of control would evolve naturally from the μ–Net architecture, if we assume that prefetch requests managed by the Code Pump occurred through a pipelined set of memory access registers. μ–Net provides a mechanism for the management of both multiple pending branches and multiple pending levels of recursion, whereas ROPE provides for only a single pending branch. ROPE also requires compiler cooperation via the insertion of software prefetch instructions, whereas μ–Net is self-managing from existing code.

6.4.3.4. Access / execute architectures

Access/execute architectures decouple instruction fetching from ALU operation; in this way, they are similar to the partitioning exhibited in μ–Net [Sm84], [Be91]. There are various implementations of A/E architectures, most notably the IBM RS/6000, the Intel i860, and other less general purpose systems. There are two aspects to these designs: first, instructions and data are communicated between the instruction unit and the ALU via queues, and second, most utilize multiple functional units, usually complementary integer and floating point units. Many of the performance increases cited result from the combination of parallelism and pipelining which this organization achieves, but we are concerned only with their functional partitioning.

The queue communication between the ALU and instruction fetch unit (IFU) is similar to the buffering provided by the communication channel in μ–Net, although in the A/E mechanism the buffer length varies with load. Due to the restricted communication across the partition, branches were computed by the IFU wherever possible; a side-effect of this decision was the ability to permit the ALU to continue ahead of the IFU, which in turn permits the IFU to reduce opcode fetch delays associated with the branch. In our design, this was an engineered result.

Further, only branch decisions and actual opcodes are sent across the partition, as our design confirms to be required. Some studies also examined the effect of the A/E architecture on memory latency tolerance, but most focus on dedicated compiler issues, rather than generalizations to hardware implementation, as in μ–Net.

6.4.3.5. IBM RS/6000

The IBM RS/6000 System [IB90], [Ba90b], [Oe90] implements dual branch stream lookahead, and provides 1 level of pending recursion in the prefetch processor, called therein the “Branch Processor.” The branches provided by the CPU permit indirect jumps, calls, and conditional branches, as well as instructions as complex as indirect conditional calls; this is a result of a design where a branch can be conditional or not, indirect or not, and provide a return address or not.

The Branch Processor has been designed so that it is logically independent from the rest of the processor, so that all branch functions use and modify registers and condition codes local to it which would facilitate its replication in the Code Pump of μ–Net.

The processor provides some level of instruction prefetch, but there is insufficient information in the published literature to determine its exact extent. The instructions are placed into a 32 entry 2-way TLB, so it appears the lookahead is limited to 16 instructions in each of 2 possible paths, at best.

The RS/6000 implements a TreeStack 2 levels deep, where the first level is a simple stack entry, and the second level permits one pending branch. The prefetch appears to provide a total of 16 outstanding opcodes, cumulative to all levels of the TreeStack. The result is a very limited and restricted implementation of prefetching, which is of limited use in highly latent systems.

6.4.4. Remote Evaluation

Remote evaluation (REV) is a version of remote procedure call (RPC), which examines variations which avoid the conventional RPC tradeoffs of performance vs. generality of interface [St90]. RPC traditionally permits the movement of work to a remote location, for execution there, with collected and returned results. REV also permits the movement of code to the data, which reduces memory bandwidth requirements for some applications, notably data collection and reduction in database systems. This research, in conjunction with file server systems such as NFS, provide justification for the domain chosen for μ–Net, where read-only code is temporally remote from the processor/RAM set.

6.4.5. Multiple alternates

μ–Net (and Mirage, in the 'abstract') and its version of exploring multiple possible states of a remote node differs from conventional forms of the exploration of alternates. Exploring branch alternatives is commonly done depth-first, such that some path in the tree of possible executions is fetched in advance of the decision of which branch is actually desired.

Similarly, the Code Pump advances beyond the known state of the receiver (i.e., last state communicated). The way in which it advances, suggests a breadth-first exploration instead, as a direct result of the description of the stability of the communicating system, and the conditions under which such stability can be ensured (Chapter 2).

Another method of anticipating multiple paths is to instantiate each path uniquely, then remove those which are not used later. The idea, described in [Sm89], is to spawn multiple processes, each exploring a different path, such that results are collected from the process which terminates first, the others being destroyed at that time. In μ–Net, this is analogous to the way in which the Code Pump sends multiple instruction streams, in the hope that one stream will be of use.

The difference between this exploration of alternates and μ–Net's is that [Sm89] relies only on the members of the set being mutually exclusive, whereas μ–Net relies on the entire set covering the space of alternates. This cover-set principle is used in the stability equations, where stability is based on the ability to send messages to the entire space of alternatives (communicability). Once messages have been sent to the entire current state, the Code Pump progresses to the next state (or set of states) and can ignore the possibility of backing up to a previous state. The Code Pump doesn't wait for any single alternate to terminate; it waits for additional bandwidth with which to send further messages, or for state resolution information from the remote state.

6.5. Conclusions

Now that μ–Net has been described (Chapter 5) and measured via μ-Scope (here, in Chapter 6), some conclusions can be discussed. μ–Net was intended to provide a vehicle for the description and elaboration of components of Mirage that were not exemplified in

the discussion of existing protocols (Chapter 4). To that end, issues of communicability, guarded messages, and isopotency were all applicable to μ–Net, and so the purpose was served.

Beyond its use as an example, μ–Net also exhibits the advantage of the Mirage model as a method for reexamination of existing disciples with a new viewpoint, another goal of any model. We have applied for a patent for the designs of μ–Net [To91b], as the result of these investigations. The application of the Mirage model to the domain of processor/memory communication led to the independent discovery of 'proactive' memory, and the description an instance (μ–Net) which generalizes current proactive memory research. μ–Net also indicated the the complementary nature of caches and this proactive memory, because caches handle reuse of code, whereas proactive memory handles first use.

6.5.1. Notes for designers

As a design for implementation, μ–Net provides a design for shared code systems to reduce the effects of latency. μ–Net requires that the executed code be read only (i.e., not self-modifying), and that the bandwidth-delay product of the communication network be an order of magnitude larger than the local memory available at the processor (i.e., workstation). For a 33 Mhz 32-bit RISC processor, and a 1 Gigabit channel, this implies a distance of about 3 city blocks between the workstation and the server, so that the speed of light propagation is 6 blocks. Further, μ–Net reduces the effects of latency, but does not reduce the bandwidth requirements on the code memory; μ–Net requires that the access bandwidth of the code memory is the same as the bandwidth of the communication channel.

μ–Net is a feasible design, given current technology. Preliminary measurements indicate that a substantial gain in speed (8x) is possible, assuming transmission latency dominates execution time by at least two orders of magnitude. The implementation of μ–Net mechanism is reasonable because even a design which models only JUMP, CALL, RETURN, and OTHER opcode types can achieve substantial speedup, with as little as 400 bytes (100 addresses) of additional storage. Extension of this design to handle a limited amount of BRANCH pre-reply would further increase performance, but measurements have not yet been made which allow us to predict the extent of this increase. The limit of the performance has been measured at near 330x.

The limitation of implementation is due to both the space complexity of the TreeStack, and the time complexity of the associative leaf- and node-matching required within the Total implementation. The set of active leaves is known, so the size of the leaf-matching required in Converger can be sufficiently restricted, and a reasonable implementation may be possible. Internal node-matching required for the TreeStack to be collapsed are more difficult to estimate, but the virtue of the design is that a delay in pruning is managed by an overgrowth in the TreeStack, and a subsequent halting in the Diverger. The system is thus self-controlled, and the communicability is constrained by the ability of the implementation to keep up with the communicating entities.

μ–Net indicates that INDIRECT opcodes are an impediment to an anticipation mechanism. Indirect opcodes are very infrequent (0.3%), have a very high penalty (1 round trip latency), and μ-Scope measurements indicate that these opcodes usually implement a table-lookup, and could be replaced by a dispatching procedure that avoids their use. Further, the removal of these opcodes may simplify processor design as well.

6.5.2. Notes for researchers

As a research vehicle, μ–Net demonstrates the utility of the abstract Mirage model. Memory management methods were confirmed by thinking of the μ–Net domain in terms of the Mirage model. μ–Net indicates a proactive memory version of Amdahl's law limiting parallelism speedup, and can also be viewed as another interpretation of the missing class of MISD in Flynn's taxonomy.

6.5.2.1. Memory management

Possible partitions of the state space permit stability, as discussed in Chapter 2. There are three obvious ways in which to partition the state space, and they have architectural analogs which were obvious upon such examination.

The state space of opcodes (i.e., the address space of a system) can be partitioned many ways, but a few are most obvious. The first is according to a finite state space volume, i.e., to partition the space uniformly, regardless of the probability density within that volume. This loosely corresponds to the way in which paged memory models handle the address space. The second corresponds to a partition which separates volumes of equal density, i.e., such that the probabilities (i.e., integral of the PDF within the partition) are uniform among the partitions. There is unfortunately no clear architectural

analog of this partitioning. The last most obvious partitioning is based on semantic information, in a way which is an attempt to emulate equiprobable partitions, under the assumption that the programmer semantically partitions the problem nearly equally. This is analogous to segmented memory, where partitioning of the address space corresponds to the semantic partitioning of the program.

6.5.2.2. Opcode anticipation Amdahl's Law

Communication is limited by interrupts or indirect opcodes which cause arbitrary discontinuities in flow path. This can be considered a communications version of Amdahl's Law, which describes the limit in execution speedup under unlimited bandwidth, space, and lookahead power, in the presence of time delays. It states that indirect instructions must incur a round-trip time cost, whereas any other instruction can be predicted, given sufficient capability. The limitation in speedup is expressed in Equation 6.3.

Equation 6.3: $$SPEEDUP_{\max} = \frac{1}{percent_indirect}$$

6.5.2.3. Flynn's taxonomy

Finally, it is possible that μ–Net's contributes to Flynn's taxonomy. There have been other claims of discovery of a real instance of the 4th category of the taxonomy, MISD. In some cases, A/E (Access/Execute) architectures are claimed to exhibit MISD characteristics, because many employ multiple ALU units, such that two instructions are executed on different streams of data within the CPU during a single cycle. The claim is that because the data set in the CPU (i.e., one data item per ALU) doesn't change during the execution of two opcodes, this is MISD; by that argument, any MIMD system can be considered MISD where the multiple data elements are considered a single set.

Another proposed MISD contender is that of VLIW (very long instruction word) machines. They operate as if multiple instructions are send in parallel to the processor. Unfortunately, the scheduling of these instructions is static; opcodes are paired for storage in the instruction word at compile time, not dynamically adjusted during execution, as our streams are.

We believe μ–Net is one of the closest attempts to a true MISD architecture, from the memory's point of view. The Code Pump sends multiple instructions to the processor, where, although only one is executed, the set permits execution at rates which a single

instruction stream could not achieve. Our speedups are the result of sending a set of instructions to the processor, so μ–Net relies on the MISD nature of the system for performance.