tions. The *benchmarking performance* data was collected by timing one workstation performing the three tasks while the background benchmark programs were run on 4, 9, 14, and then 19 workstations. Each of the benchmark programs used a fixed record length of 80. Different LAN benchmark programs were used to simulate background activity. Performance (timing of tasks) for one workstation using *actual task activity* in the background was compared to performance for one workstation using the benchmark programs in the background.

Press presents a BASIC LAN benchmark program that opens a file and then processes a loop which writes one record, reads the record and displays the cumulative mean and variance of the times between transactions. This reading and writing of a record is designed to represent *constant network activity*—CONSTANT-A. CONSTANT-B is a modification of the constant activity program that opens a file, writes a record, closes the file, and then executes a loop which opens the file, reads the record, writes the record, and then closes the file. Another modification, CONSTANT-C is developed to more accurately represent constant network activity (input and output) on the network. This program opens a file, writes ten records, and closes the file. A loop is then processed which opens the file, randomly reads a record, randomly writes a record, and then closes a file.

Press also presents an *intermittent activity* program with the time between transactions distributed normally—NORMAL-A. A modified version of the program, NORMAL-B, opens a file, writes 10 records, and closes the file. The program then loops to process an open file, randomly read a record, randomly write the record, and close the file.

Actual Performance: Before any of the background programs were run, a single workstation took 20.7 seconds to complete the scenario—13.30 seconds to load WordPerfect from the server, 2.90 seconds to load a 50K file from the server, and 4.50 seconds to save the 50K file on the server. As the number of workstations in the background performing the scenario increased, the time to complete the scenario increased at a nonlinear rate. Normalizing the total time to complete the scenario to one workstation with 4, 9, 14, and 19 background stations all performing the same tasks as the timing station, it took 4, 9, 15, and 23 times as long, respectively, as a single workstation to complete the scenario.

Benchmark Performance: The three constant network activity and two intermittent activity (normal) benchmark programs were executed on 4, 9, 14, and 19 background stations while one timing workstation accomplished the scenario. CONSTANT-A, since it did not open and close the file, did not compare well to the actual performance for different numbers of background workstations. For all combinations the timing was the same as one workstation performing the scenario. The CONSTANT-B and CONSTANT-C programs for 4, 9, 14, and 19 background workstations took 3, 6, 9, and 12 times and 4, 7, 11, and 14 times as long as a single workstation to complete the scenario, respectively. The NORMAL-A and NORMAL-B benchmark programs took 4, 8, 12, and 16 times and 4, 7, 10, and 14 times as long as one workstation to complete the scenario for 4, 9, 14, and 19 background workstations, respectively. While none of the benchmarking programs exactly duplicated the actual performance, some performed quite well for both individual tasks and the total scenario. The task of loading WordPerfect seemed to be the most difficult task for the background programs to represent, while they appeared to more closely simulate the file loading and saving activities.

Based on the results, the benchmark programs presented by Press generally did a good job of simulating background activity. For the configuration and the scenario used, Press's intermittent activity program best simulated actual background activity. While the LAN configuration used in the test is minimal, subsequent analyses have been conducted on different LANs utilizing performance enhanced server and client configurations. With expected performance increase, the background programs continue to simulate network activity adequately, especially for smaller configurations. (Tables of benchmark performance are available upon request.)

Timothy Paul Cronan
David E. Douglas
University of Arkansas
Peggy L. Luster
Bradley University

MORE ERROR-DETECTING DECIMAL DIGITS

Wagner and Putter [22] and follow-ups by Maurer [15] and Pezzulo [19] presented a number of very interesting and useful ideas for providing error detection with decimal digits for use with numbers such as credit card numbers, that are often manually transcribed. I would like to expand on the idea presented by Pezzulo, who suggested choosing as self-checking numbers multiples of 13. Let us consider using multiples of a prime number in general. I will show how to append check digits to an existing number to get a multiple of a given prime. Encoding and error-detection algorithms can be implemented easily using ordinary arithmetic, and a method for doing this with very large numbers is in-

cluded. An important advantage of these codes is that their properties are easy to analyze, and primes can be found that can be proved to detect errors well. Optimum choices for the prime number are included here. Codes of this type were apparently first found by Diamond [6] and studied in depth by Brown and others [2, 18], for the binary number system with the idea of using them for detection and correction of transmission or hardware errors.

Simply using multiples of a prime number has one weakness. If the last digit is zero, and if it is dropped, then the resulting number is still divisible by the prime number and therefore undetected. This can be prevented by adding a small constant to the multiples of the prime. Here I will assume that we use numbers greater by one than multiples of a chosen prime.

As an example, the prime number 947 enables detection of all single and double errors in a number of 18 or fewer digits. Wagner and Putter list seven types of common errors: single errors, adjacent transpositions, twin errors, jump transpositions, jump twin errors, phonetic errors, and omitting or adding a digit. Of these, the first six are special cases of single or double errors and therefore are detected. In the last case, we can show that one or two digits omitted from either the beginning (if they are not zero) or the end will be detected, and the loss of a digit within the number will be detected about 99.9% of the time. Wagner and Putter's final recommendation to their client used four check digits. A check using multiples of 9973 would apparently give error detection probabilities very nearly equivalent to their choice of four digits. For the code presented here, the error detection capabilities are more easily analyzed and clearly understood than for Wagner and Putter's code.

Let us assume the information to be protected consists of k digits, and we will add r check digits following to make a total of $n = k + r$ digits. We propose that an r-digit prime number A be chosen, and that the check digits be chosen in such a way that dividing the n-digit resulting number by A gives remainder 1. I will refer to the correctly encoded n-digit number as the transmitted number, and the possibly corrupted n-digit number being checked as the received number, by analogy with error-correcting codes. First we will discuss how to accomplish the calculation of the check digits, and then we will discuss what errors can be detected with such a scheme.

To encode, make an n-digit number by appending to the original number, r digits that represent a number between $A - 2$ and $10^r - 2$. Then divide the resulting number by A, subtract the remainder from the n-digit number, and add 1. Now if you divide the resulting number by A, the remainder will be 1 as required. For example, I might choose $A = 971$ and I might simply append $A = 971$ to the uncoded number. If the number to be protected is 9237275, then after appending 971 it becomes 9237275971. Dividing this by 971 gives remainder 524. Subtracting this from 9237275971 and adding 1 gives 9237275448. The remainder after dividing 9237275448 by 971 is 1. Note that the original information digits are unchanged.

To check for errors, simply divide by A—if the remainder is not equal to 1, clearly something has changed and an error has been detected. If the remainder is equal to 1, you assume that no error has occurred. Now the question is, what kind of errors are not detected because the remainder is 1 even though an error has occurred? First of all, of all n-digit numbers, only one out of every successive A numbers will give remainder 1 on division by A, so the probability that a randomly chosen number will be accepted by this system is $1/A$. This suggests that A should be a large r-digit prime.

I would like, for the purpose of analysis, to define the *error pattern* E to be the difference between the n-digit received number R being checked and the error-free transmitted n-digit number T. $E = 0$ corresponds to correct reception. An undetectable error means that T and R are unequal and each is one more than a multiple of A, and therefore $E = R - T$ must be a non-zero multiple of A, and conversely if E is a non-zero multiple of A, then since T is one more than a multiple of A and $R = T + E$, then $R \neq T$ and R will also be one more than a multiple of A and therefore be an undetectable error. Thus undetectable errors correspond to error patterns that are non-zero multiples of A.

Next, I would like to define any pattern of the form $E = e \cdot 10^i$, $1 \leq |e| \leq 9$ and $i < n$, to be a *single error*. Every error corresponding to a single-digit error corresponds to a single error in this sense, but certain other errors meeting this definition of a single error involve more than one digit. For example, if 10000000 is transmitted and 09999999 is received, this corresponds to a single error of $-1 \cdot 10^0$, although all the digits are changed. Every single error is detected if $A > 10$, since no single-error pattern under those circumstances is divisible by A. Since detecting all single errors requires two digits, and since it is desirable that A be large, it would seem best in general to use the largest two-digit prime, 97.

Now define a double error to the sum of two single errors with different values of error position i. Clearly, any error that affects two digits in a number is a double error in this sense, and certain error patterns affecting more than two digits will also count as double errors. For any given prime number, there exist undetectable double-error patterns. For example, for any prime number A except 2 or 5, $10^{A-1} - 1$ is divisible by A, by Fer-

mat's theorem, and this is a double-error pattern. Note that A must have at least three digits to detect double errors—if $A = 97$, for example, the error 00 becomes 97 would be an undetectable double error. Otherwise there seem to be no simple general statements that can be made for double errors.

The number of error patterns is small enough, however, so that they can be checked exhaustively by computer for any given prime A. The search can be shortened somewhat by observing that if $E = e_1 \cdot 10^{i_1} + e_2 \cdot 10^{i_2}$ with $i_1 > i_2$ is an error pattern that is divisible by A, then $E' = e_1 \cdot 10^{i_1 - i_2} + e_2$ is also a double error pattern divisible by A but with the special property that one of the errors is in the last position. Therefore a search need be made only for error patterns with errors in the last position—if there are no undetectable double error patterns with an error in the last digit, there are none at all. Thus, for any given prime number A, we can find by computer search the largest n for which no double error pattern of n or fewer digits is divisible by A, and therefore for which all double errors are detected.

It is desirable that n, the maximum number of digits such that A will detect all double errors, be large, and also that among r-digit primes, A be as large as possible. Therefore we will call an r-digit prime A optimum if it detects all double errors in a block of n digits, and no larger r-digit prime will detect all errors in a block of n digits. By exhaustive search I have found all the optimum three, four, and five-digit primes for double-error detection:

A	n	A	n	A	n
997	3	9973	85	99989	717
983	8	9967	87	99961	1289
977	12	9931	315	99881	2497
971	16	9833	351	99349	2922
947	18	9323	377	99119	3527
761	22				

It is also possible to search for triple error patterns, and here are all the optimum four, five, and six digit primes for detecting all triple errors: (The four digit ones appear quite unattractive—with 9811, for example, you have four check digits and four information digits).

A	n	A	n	A	n
9829	6	99991	5	993983	22
9811	8	99829	9	993961	33
		99787	12	993893	70
		99721	17	992357	73
		99643	18	991663	77
		99487	19	962837	79
		99241	24	962627	83
		97429	26	903143	84

Next let us consider an error that results from interchanging two digits. Suppose that digit t_i in position i and digit t_j in position j are interchanged, and let us assume that $i > j$ and that the digits are numbered from right to left starting with zero. Then the error pattern will be

$$(t_j \cdot 10^i + t_i \cdot 10^j) - (t_j \cdot 10^j + t_i \cdot 10^i) = (t_j - t_i)(10^{i-j} - 1)10^j$$

This will be divisible by A if and only if $10^{i-j} - 1$ is. For certain values of $i - j$, this will be divisible by A. For example, by Fermat's theorem, it will be if $i - j$ is a multiple of $A - 1$. It can be verified easily by computer that for $A = 97$, for example, the only values of $i - j$ for which $10^{i-j} - 1$ is divisible by A are multiples of $A - 1 = 96$. One would expect this kind of error to occur commonly with adjacent digits or digits that are near to each other, i.e. for very small values of $i - j$, and all of those would be detected.

Now let us consider error patterns resulting from changing b successive digits—these are often referred to as bursts of length b. Suppose we have a received number R and a transmitted number T that differ in this way. Then $E = R - T$ will be of the form $e \cdot 10^i$ where e is a positive or negative b-digit number. Assuming that $A > 10$, this will be divisible by A if and only if e is divisible by A. It will certainly not be divisible by A if it has fewer digits than A, so all bursts of length less than r, the number of digits in A, will be detectable. Most bursts of length r will be detected, also. For example, for $A = 9973$, the only undetectable error pattern would be 9973. The only way for this to occur and affect only four digits would be for the affected four digits of the smaller of T and R to be $0000 + x$ and the corresponding four digits of the larger to be $9973 + x$, with $0 \le x < 27$.

Digits omitted at the beginning of a number will act like a burst, so every pattern of fewer than r omitted digits at the beginning of a number will be detected, if the omitted digits are not all zero. Suppose the received number R differs from the transmitted number T by having the last i digits omitted. Assuming that the transmitted message was $AX + 1$ and denoting the decimal value of the last i digits by D, the value of the received number is $(AX + 1 - D)/10^i$. This will be accepted if the remainder after dividing this by A is 1, or if $(AX + 1 - D)/10^i - 1$ is divisible by A. This will be divisible by A if and only if $AX + 1 - D - 10^i$ is, which in turn will be divisible by A if and only if $-10^i - (D - 1)$ is. This will certainly not be divisible by A if $i < r$ and $A > 2 \cdot 10^i$ because it will then surely be less than A in absolute value and not zero. Thus if we choose A to be a quite large r-digit prime, we can detect every omission of up to $r - 1$ digits at the end of a number.

Now let us consider the omission of a single digit inside the number. Suppose the transmitted number consists of a number a followed by a digit d followed by an i-digit number b. For example we might consider 9237275448 to consist of $a = 92372$, $d = 7$, and $b = 5448$, with

$i = 4$. Then $T = a \cdot 10^{i+1} + d \cdot 10^i + b$. If the single digit d is omitted, the value becomes $R = a10^i + b$. The difference is $E = a10^i - d10^i - a10^{i+1} = -(9a + d)10^i$ and the error will be undetectable if and only if this number is divisible by A, which in turn will be true if and only if $9a + d$ is divisible by A. Considering all possible values of a, there will be a first for which $9a + d$ is divisible by A, and then after that every Ath value of a will result in $9a + d$ being divisible by A. Therefore, if a has enough digits, about 1 in every A values of a will result in an undetectable error. If the number of digits in a is small, there may be none or one or a few.

Finally, I would like to point out that the computations involved in encoding and error detection can be done easily on a computer. This is certainly obvious if n, the number of digits in the transmitted and received numbers is no greater than the number of digits for which integer division is implemented in the system being used. For larger numbers, it is also not difficult. Suppose the number has n digits, the prime has r digits, and integer division is implemented for $p > r$ digits. Let us define $m = p - r$. Now let t_0 be the m digit number consisting of the m lowest order digits of a number t, let t_1 be the next m digits of t, etc., and t_s be the highest-order group of m or fewer digits. Then

$$t = t_s 10^{ms} + t_{s-1} 10^{m(s-1)} + \cdots + t_1 10^m + t_0$$

Then $t_s t_{s-1} \cdots t_1 t_0$ can be considered to be the representation of t in a number system with base 10^m. You can do long division in this number system using the same algorithm as is used for doing long division by a single digit divisor in decimal, now using the built-in division for dividing each "digit."

As a simple example, suppose you have available 10-digit arithmetic and wish to protect 15 digits of information using 3 check digits. $A = 947$ would be the optimum choice to use with the $15 + 3 = 18$ digit encoded numbers. Suppose the information to be protected is 123456787654321. To encode, concatenate three digits at least as great as $A - 1$ to the information. Let us choose A which is 947, giving 123456787654321947. Now we must find the remainder after dividing this number by 947. We will find that the remainder is 517. Subtracting this remainder from the original 18-digit number and adding 1 gives 123456787654321431 as the required encoded number.

To get the desired remainder we must divide 123456787654321947 by 947. Take $m = 10 - 3 = 7$, and as "digits" in the base-10^7 system, take groups of 7 digits from the original number: $t_2 = 1234$, $t_1 = 5678765$, and $t_0 = 4321947$. (Then t is a three-"digit" number in its base 10^7 representation.) Now divide $t_2 = 1234$ by $A = 947$ and get quotient 1 and remainder 287. Concatenate the remainder and t_1, getting 2875678765, and divide this by 947, getting quotient 3036619 remainder 572. Now concatenate the remainder and t_0, getting 5724321947, and divide by 947, getting quotient 6044690 and remainder 517. This is the required remainder. (The quotient is the concatenation of the three quotients, 130366196044690, although it is not needed here.)

Now let us do the same calculation assuming that we have 32-bit fixed-binary arithmetic available. Using that, we handle dividends up to 9 decimal digits, so we should choose $m = 9 - 3 = 6$, corresponding to a base 10^6 number system. Now there will be three "digits," 123456, 787654, and 321947. Dividing 123456 by 947 gives quotient 130 and remainder 346. Dividing 346787654 by 947 gives quotient 366196 and remainder 42. Dividing 42321947 by 947 gives quotient 044690 and remainder 517. This is the same remainder as before, and the quotient obtained by concatenating these three base-10^6 quotient digits is also the same as before.

I would like to acknowledge valuable suggestions by Neal Wagner.

W. Wesley Peterson
Department of ICS
University of Hawaii
Honolulu, HI 96822

ECONOMIC ANALYSIS OF COMPUTER HARDWARE PERFORMANCE: SOME THEORETICAL AND METHODOLOGICAL OBSERVATIONS

The motivation for this article is a study by Kang [14] which presents a model for evaluating computer system performance employing the economic approach. The economic approach assumes the performance of a computer is related, via a production function, to the attributes of its components. When this production function is combined with the objective function of the firm that produces the computers (e.g., cost minimization), the price (or cost) of the computer can be related to its performance and the attributes of its components. This relationship is often called the price function (see Cale et al. [3], Ein-Dor [7], Grosch [10], Ein-Dor and Jones [9], and Mendelson [16] among others).

In his article Kang examines problems with previous studies and develops a theoretical basis to justify his model. In this article we show that the model and methodology suggested by Kang are basically incorrect. We also use this opportunity to clarify some of the principles