Reflections on the Pentium Division Bug

Manuel Blum and Hal Wasserman

Abstract

We review the field of **result-checking** and suggest that it be extended to a methodology for enforcing hardware/software reliability. We thereby formulate a vision for "self-monitoring" hardware/software whose reliability is augmented through embedded suites of run-time correctness checkers. In particular, we suggest that embedded **checkers** and **correctors** may be employed to safeguard against arithmetic errors such as that which has bedeviled the Intel Pentium Microprocessor. We specify checkers and correctors suitable for monitoring the multiplication and division functionalities of an arbitrary arithmetic processor and seamlessly correcting erroneous output which may occur for any reason during the lifetime of the chip.

Keywords

Built-in testing, concurrent error detection, fault tolerance, Pentium, reliability, result-checking, verification.

I. A Gentle Introduction to Result-Checking

A. *Issues of Reliability*

The discovery of an occasional yet embarrassing bug in the Intel Pentium Microprocessor led to a flurry of controversy and an unexpected public-relations debacle for Intel. And yet, as a stunned Intel management pointed out, occasional bugs are the rule rather than the exception in released microprocessors.

Thus, the issues here go beyond the Pentium. The general public has made an unexpectedly passionate demand for a radically higher standard of reliability in computer hardware and software. But how are such demands to be met? Indeed, as the potential consequences of malfunctioning computer systems grow more alarming, while the systems themselves grow more complex, the need for new assurances of reliability becomes ever more pressing.

The usual approach to reliability is via **testing,** an elaborate, expensive, and essential process. Unfortunately, testing is often insufficient to find "occasional bugs"—bugs which manifest themselves only on certain rare combinations of inputs. An alternative technique is **formal verification:** proving a system correct. While ideal in theory, verification is not currently, and may never be, a pragmatic route to reliability. Yet another technique, employed occasionally in critical software, is that of **redundant coding (fault toler-**

ance): two programming teams create separate versions of a program; at run-time, both versions are executed and their outputs compared. This method is evidently inefficient, and other objections to it shall be made below.

We take a different approach, one via the field of **result-checking,** which has developed in the Computer Science Theory community over the past seven years. We believe that result-checking, as extended in a natural way by more informal heuristics, now points the way to a new methodology for hardware/software reliability: a methodology of supporting reliability via suites of embedded, run-time correctness-checks. In particular, the Pentium division-bug evokes a textbook situation of "occasional errors," which result-checking was explicitly developed to deal with.

We begin, then, with a general review of result-checking and its applications.[1]

B. Checking

Simple checking, formally introduced in [2], is based on the following observation: For certain mathematical functions f, the task of determining, given inputs x and y, whether or not $f(x) = y$ is easier than the task of, on input x, computing $f(x)$.

As a trivial example, consider the following computational task: on input c, an integer known to be composite, output any non-trivial factor d of c. This task is believed to be difficult. And yet, given c and d, we can determine whether or not d is a correct output on input c simply by checking if d divides evenly into c.

These ideas lead to the simple checker, defined as follows: *A* **simple checker** *for function f is a program which, given inputs x and y, returns the correct answer to the question, "Does $f(x) = y$?" The checker may be randomized, in which case, for* any *x and y, it must give the correct answer with high probability over its internal randomization. Moreover, the checker must take asymptotically less time than any possible program for computing f.*

The time limit is crucial here, as it prevents us from checking whether $f(x) = y$ by simply recomputing $f(x)$. Essentially, we must think of some way to take advantage of the additional information implied in being given y as well as x. This is a creative process, and one which may be specific to a particular function f. For some functions, there is no simple checker.

[1]Portions of Section I parallel [6, Section 1].

$$\begin{pmatrix} 1 & \text{-}2 & 1 \\ 0 & 3 & 3 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & \text{-}1 & 0 \\ 2 & 2 & 2 \\ 3 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \text{-}1 & \text{-}5 & \text{-}3 \\ 15 & 6 & 9 \\ 0 & \text{-}1 & 0 \end{pmatrix}$$

Fig. 1. Example of matrix multiplication.

$$\begin{pmatrix} 1 & \text{-}2 & 1 \\ 0 & 3 & 3 \\ 1 & 0 & 0 \end{pmatrix} \cdot \left[\begin{pmatrix} 0 & \text{-}1 & 0 \\ 2 & 2 & 2 \\ 3 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ \text{-}3 \end{pmatrix} \right] = \begin{pmatrix} 1 & \text{-}2 & 1 \\ 0 & 3 & 3 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} \text{-}2 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \text{-}2 \\ 0 \\ \text{-}2 \end{pmatrix}$$

$$\text{while} \quad \begin{pmatrix} \text{-}1 & \text{-}5 & \text{-}3 \\ 15 & 6 & 9 \\ 0 & \text{-}1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ \text{-}3 \end{pmatrix} = \begin{pmatrix} \text{-}2 \\ 0 \\ \text{-}2 \end{pmatrix}$$

Fig. 2. Checking the matrix multiplication from Fig. 1.

As an example, we will consider Freivald's simple checker for matrix multiplication [9]. First recall that computing the product of two $n \times n$ matrices by the most straightforward method—i.e., the top left component of the product in Fig. 1 is calculated to be $(1 \cdot 0) + (-2 \cdot 2) + (1 \cdot 3) = -1$—takes time $O(n^3)$. More sophisticated methods have also been developed: methods which take time $O(n^c)$ for various values of c, $2 < c < 3$. While these fast algorithms have often been considered impractical, they are now employed in some computer systems. A program embodying such an algorithm could be quite complex and so, perhaps, buggy and in need of a checker.

The Freivald matrix-multiplication checker has as its inputs $n \times n$ matrices A, B, C; it must determine whether or not $AB = C$. It begins by generating an n-high column-vector r of random numbers.[2] It then calculates and compares products $A(Br)$ and Cr. Note that these calculations can be done in time just $O(n^2)$, using only straightforward vector-by-matrix multiplication. For example, in Fig. 2 we pick a random value for r and use it to check the matrix multiplication from Fig. 1. We observe that the two column-vectors calculated in Fig. 2 are identical, which suggests that the original matrix multiplication is

[2]We won't go into details such as specifying the domains from which the components of A, B, C, and r are to be taken.

indeed correct.

Why does this work? If $AB = C$, then evidently $A(Br) = (AB)r = Cr$ for any r. On the other hand, if $AB \neq C$, it *is* possible that, for certain "bad values" of r, $A(Br)$ will nevertheless equal Cr. However, it may readily be proven that the probability (over the random choice of r) of being fooled in this way is small. Thus, if we repeat the above process for several values of r, and accept C as correct if $A(Br) = Cr$ for all r tried, we have a simple checker whose probability of error can be made arbitrarily low.

What if the matrix-components are real numbers, so that a correct matrix-product may include small round-off errors? [1] extends the Freivald checker to this case.

Note the applicability of such techniques to pragmatic run-time checking of software. For we have here checked a complicated, $O(n^{>2})$-time program with a simple, $O(n^2)$-time program. The checker is thus easy to code compared to the program it checks, unlikely to be buggy compared to the program it checks, and quick to execute compared to the program it checks.

C. Applications of Checking

We have sketched above checkers as they are envisioned in Computer Science Theory. These are **complete checkers:** they have the desirable property of being unfoolable (with high probability over internal randomization). For more general software-engineering situations, in which we deal with less clean, less mathematical computations, we should also consider **partial checkers:** that is, checkers which, while not unfoolable in the strong sense exhibited above, test for *some* of the properties of correct output. Partial checking is a heuristic rather than a rigorous concept: a partial check is worthwhile if we are reasonably convinced that it will catch many naturally occurring instances of buggy behavior.

For example, physicists often use a partial checker to assure the reliability of their simulation software. At occasional stages in the simulation of a physical system, the computer will pause to calculate the system's total energy or momentum. Comparisons of the results, applying the principles of conservation of energy and momentum, allow the computer to catch many errors.

We have a vision for software of the future: software which is enriched with embedded run-time checkers. We theorize that, when the total self-checking in a software package reaches a "critical mass," the package will be saturated to the extent that most bug occurrences will trigger checker warnings. During the conventional software-testing phase, these checkers will then serve as an efficient and reliable **testing oracle,** assuring that the programmers don't miss buggy behavior generated in response to any of their tests. And, even after the product has been shipped, the checkers may be allowed to remain in the code, and so will continue to help with the process of incremental correction and of software porting, updating, and maintenance. In particular, such checkers will constitute an automatic **regression test.**

D. Checking a Symbolic-Mathematics Library

Questions remain as to the utility of checking as a software-engineering tool. We shall approach these through a consideration of checkers that might profitably be employed in a symbolic-mathematics software library. Such libraries indeed stand in need of checking: one professor at U.C. Berkeley is fond of posting outside his office a list of problems for which standard symbolic-mathematics packages return incorrect answers.

- As a first example, consider a routine which, given as input an equation in x, outputs a value for x which solves the equation. For this problem, a straightforward check is to plug this value back into the equation and verify that the two sides are now equal.[3]

This is a complete check. Note also that the checker need employ only code for simplifying and comparing expressions: code which presumably exists already in the symbolic-mathematics library. Thus, creating this checker will take little extra work. Note also that simplifying/comparing may reasonably be considered a simpler computational task than finding roots. Thus, our checker should be both quick and simple relative to the routine it checks, and so, hopefully, less likely to be buggy. Moreover, even if the checker is buggy, will the simplifying/comparing bug correlate with the root-finding bug so as to make the two sides of the equation seem to match when an incorrect root is plugged in? Such a correlation seems unlikely.

[3] Indeed, any math student knows that, after solving an equation, one should plug the solution back in to check its correctness. But computers seldom carry out such checks. After all, computers never make mistakes. Right?

• As another example, consider a routine which, given as input a symbolic function $f(x)$ and real numbers a and b, outputs the exact value of definite integral $\int_a^b f(x)dx$. A possible check is to approximate the same integral by summing thin rectangles under the curve of the function. We then reject unless the two values are approximately equal.

The implementation of this checker involves several tricky issues. First, the checker works best for functions $f(x)$ of bounded slope on interval $[a, b]$, so that we can accurately approximate the integral with a small number of rectangles. Moreover, a result which is incorrect but very close to the correct answer may be mistakenly accepted. Thus, this is a partial checker.

Nevertheless, observe that we have checked a complex computation—calculating a definite integral exactly, with all the tricky rules-of-thumb which this requires—by means of a simple, mechanical approximation. The code for carrying out this approximation should be easy to write and would likely be bug-free. Heuristically speaking, it seems probable that many naturally occurring bugs would be caught by this straightforward check.

• As a final example, consider a routine which, given symbolic function $f(x)$, returns a symbolic function $g(x)$ which is supposed to equal indefinite integral $\int f(x)dx$. A check for this routine is to differentiate $g(x)$ and verify that $g'(x) = f(x)$.

This is a complete check. Moreover, observe that this checker for integration requires only code for differentiation—code which no doubt already exists. We also recall that differentiation is in general a simpler, more mechanical process than integration.[4] Thus, our checker is likely to be both quicker and more reliable than the routine it checks. Furthermore, even if the differentiation routines are buggy, will these differentiation bugs exactly cancel out and hide the integration bugs? It doesn't seem likely.

An alternative, more eccentric check would be to pick random numbers a and b, to approximate $\int_a^b f(x)dx$ by summing thin rectangles, and to check that this value is approximately equal to $g(b) - g(a)\ldots$ which should be true if $g(x)$ is indeed the indefinite integral of $f(x)$. Of course, we could pick "unlucky values" of a and b such that $g(b)-g(a)$ is equal to $\int_a^b f(x)dx$ even though $g(x)$ is not the proper indefinite integral. This is a partial

[4]There are exceptions: in particular, inverse functions such as arcsin may be trickier to differentiate than to integrate. It is also unclear how to check when the program claims that $\int f(x)dx$ cannot be expressed as a composition of elementary functions.

check; nevertheless, it seems likely to catch many naturally occurring bugs.

E. Issues of Checking

Through the above examples, we have hinted at several key issues of designing and evaluating checkers. First, there is the issue of efficiency. Checkers are required—at least per their formal definition—to take asymptotically less time than the programs they check. Thus, checking should not significantly slow a software package.

Second, there is the issue of coding effort. We have seen that checkers may often be simpler and easier to code than the programs they check. Moreover, the checks may sometimes be accomplished using code modules which already exist. Thus, the decision to add checkers may be a good trade-off in terms of programming man-hours.

Third, there is the difficult question of reliability. To what extent should we believe that a checker will succeed in catching software errors? Complete checkers inspire most confidence; yet partial checkers, if thoughtfully designed, may generate a strong heuristic assurance of catching many naturally occurring bugs. But what if the checker itself is buggy? We have suggested two responses to this objection: *(1).* Checkers are often simpler than the programs they check, and so, seemingly, less likely to be buggy. In particular, it is often possible to check a computational resource by making use of only more primitive resources. This leads to a productive methodology of **hierarchical checking.** For example, trigonometric functions may be corrected by reference to multiplication (see [12], [13]), which may in turn be corrected by reference to addition (see Section II-B). *(2).* Even if both checker and program are buggy, we note that the checker is often substantially *different* from the program. Thus it seems unlikely that the bugs in the checker and those in the program will correlate so as to mask each other.

At this point, checking may be favorably compared with **redundant coding.** Checking should be efficient; redundant coding doubles run-time or parallel-hardware requirements. Checking may require little additional programming; redundant programming doubles man-hours. Finally, the two versions of a redundant program, unlike a program and its checker, are not substantially *different* from each other. Thus, it has been found that redundant programs often turn out to be quite similar and so suffer from correlated bugs [8].

Let us also consider whether it does any good for a program to know when its behavior is buggy: for it could be argued that the user would notice buggy behavior in any case. But we argue that checking allows for a more effective identification of bugs. In particular, checkers can observe subtle bugs within program modules—bugs invisible to the user. Checkers also obviate problems of users who do not bother to report bugs, and of output which may be difficult for a human being to recognize as incorrect. Thus, a bug can be detected and fixed early, before it goes on to cause a catastrophic failure.

We have argued above for the utility of checkers as a testing oracle and as an aid to software maintenance and regression testing. It can also be argued that a checker, if nothing else, could warn the user of incorrect output. For example, a symbolic-mathematics library, which scientists rely upon in their work, might at least warn the user when an output is fallacious.

Finally, there is the possibility that, having recognized its own output to be incorrect, a program could attempt to correct itself. We turn now to a consideration of this intriguing possibility.

F. Self-Correcting

We note first that **self-correcting** should be needed only when a checker identifies buggy behavior. In a well-tested program, this should be a rare event. Thus, even if a correcting procedure is fairly time-consuming, it will have little effect on the system's average-case behavior. Hence we will be more lenient in evaluating the performance-cost for correctors than for checkers.

Say that a program has discovered, by means of a checker, that a value it has computed is incorrect. It might then employ heuristic patches in an attempt to correct itself... or at least to return a value which it judges to be partially correct or more benign to the system as a whole. For example, when correcting the evaluation of a smooth function, it might alter the input data slightly, hopefully getting out of the situation which generated buggy behavior while only slightly changing the output. Or it might have available an old or alternative version of a software module, which it could load and run when the primary version fails. Consider our example from Section I-B of checking a complicated, $O(n^{>2})$-time matrix-multiplication program. We might also have available a straightforward,

$O(n^3)$-time matrix-multiplication program. Such a program would be easy to code, and, while slow, would hopefully be needed only rarely.

Complex correcting [4] is a more rigorous methodology for correcting certain clean mathematical functions. Complex correcting is based on the fact that, for many functions f, we can efficiently compute $f(x)$ if we know the value of f at several random-looking inputs other than x. And thus it is possible to trick our way around occasional "bad inputs" at which a program for computing f fails. Since it is easy to determine, via a simple stage of random testing, that such a program is correct on *most* inputs, a self-corrector of this sort will then suffice to patch over the remaining, occasional errors, making the program effectively perfect.

As an example of a complex corrector [4], let us return to the problem from Section I-B of multiplying matrices $A = \begin{pmatrix} 1 & -2 & 1 \\ 0 & 3 & 3 \\ 1 & 0 & 0 \end{pmatrix}$, $B = \begin{pmatrix} 0 & -1 & 0 \\ 2 & 2 & 2 \\ 3 & 0 & 1 \end{pmatrix}$. Say that a program has calculated this product incorrectly, necessitating a self-correction. Now, we can easily write each of A, B as a sum of two matrices in an essentially random way. For example:

$$A = \begin{pmatrix} 3 & 4 & 0 \\ 2 & 1 & 3 \\ -1 & 0 & 1 \end{pmatrix} + \begin{pmatrix} -2 & -6 & 1 \\ -2 & 2 & 0 \\ 2 & 0 & -1 \end{pmatrix}$$

$$B = \begin{pmatrix} -2 & 3 & 3 \\ -1 & 0 & 1 \\ 3 & 2 & 0 \end{pmatrix} + \begin{pmatrix} 2 & -4 & -3 \\ 3 & 2 & 1 \\ 0 & -2 & 1 \end{pmatrix}$$

We are just writing A as the sum of a random matrix R and $A - R$, and similarly for B.[5]

But then, as illustrated in Fig. 3, in place of calculating AB directly we may substitute four *entirely different* matrix multiplications. Of course, this method multiplies run-time by a factor of 4. Such a small-constant-factor increase in run-time is usually required for complex correcting. Also needed are matrix addition and subtraction, which we assume to be simple, quick, and reliable.

Why does this complex corrector work? We assume that, through random testing, we have previously determined that our program does matrix multiplication correctly for *most* input matrices: say, for all but at most a one in a million fraction of possible inputs. Now,

[5]Again, we won't specify the domains from which the components of A, B, C, and the random matrices are to be taken.

$$
\begin{aligned}
AB &= \begin{pmatrix} 1 & -2 & 1 \\ 0 & 3 & 3 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & -1 & 0 \\ 2 & 2 & 2 \\ 3 & 0 & 1 \end{pmatrix} \\
&= \left[\begin{pmatrix} 3 & 4 & 0 \\ 2 & 1 & 3 \\ -1 & 0 & 1 \end{pmatrix} + \begin{pmatrix} -2 & -6 & 1 \\ -2 & 2 & 0 \\ 2 & 0 & -1 \end{pmatrix} \right] \left[\begin{pmatrix} -2 & 3 & 3 \\ -1 & 0 & 1 \\ 3 & 2 & 0 \end{pmatrix} + \begin{pmatrix} 2 & -4 & -3 \\ 3 & 2 & 1 \\ 0 & -2 & 1 \end{pmatrix} \right] \\
&= \begin{pmatrix} 3 & 4 & 0 \\ 2 & 1 & 3 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} -2 & 3 & 3 \\ -1 & 0 & 1 \\ 3 & 2 & 0 \end{pmatrix} + \begin{pmatrix} 3 & 4 & 0 \\ 2 & 1 & 3 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & -4 & -3 \\ 3 & 2 & 1 \\ 0 & -2 & 1 \end{pmatrix} + \\
&\quad \begin{pmatrix} -2 & -6 & 1 \\ -2 & 2 & 0 \\ 2 & 0 & -1 \end{pmatrix} \begin{pmatrix} -2 & 3 & 3 \\ -1 & 0 & 1 \\ 3 & 2 & 0 \end{pmatrix} + \begin{pmatrix} -2 & -6 & 1 \\ -2 & 2 & 0 \\ 2 & 0 & -1 \end{pmatrix} \begin{pmatrix} 2 & -4 & -3 \\ 3 & 2 & 1 \\ 0 & -2 & 1 \end{pmatrix} \\
&= \begin{pmatrix} -10 & 9 & 13 \\ 4 & 12 & 7 \\ 5 & -1 & -3 \end{pmatrix} + \begin{pmatrix} 18 & -4 & -5 \\ 7 & -12 & -2 \\ -2 & 2 & 4 \end{pmatrix} + \begin{pmatrix} 13 & -4 & -12 \\ 2 & -6 & -4 \\ -7 & 4 & 6 \end{pmatrix} + \begin{pmatrix} -22 & -6 & 1 \\ 2 & 12 & 8 \\ 4 & -6 & -7 \end{pmatrix} \\
&= \begin{pmatrix} -1 & -5 & -3 \\ 15 & 6 & 9 \\ 0 & -1 & 0 \end{pmatrix}
\end{aligned}
$$

Fig. 3. Correcting matrix multiplication.

each time we employ this self-correcting method, our problem of multiplying A by B is remapped to a problem of doing four matrix multiplications, each on a pair of matrices which are essentially random.[6] Thus, for each of the four matrix multiplications, the multiplication will be correct, with chance of error at most one in a million. Hence, the chance that *any* of the four multiplications will be incorrect is at most four in a million. But this means that, for *any* A, B, each time we employ this randomized correcting method, there is a very high probability (over internal randomization) of getting the correct value

[6]Our language here is deliberately informal. This argument can be made rigorous.

of AB.

Such complex correctors are possible for a broad variety of mathematical functions. Rubinfeld [12] has shown that many naturally occurring functions satisfy **robust functional equations** which allow for correcting. Vainstein [13] has shown that a large class of functions satisfy polynomial functional equations, and has suggested that this may allow for correcting.

We conclude our discussion of correcting with a note of caution. An advantage of checkers is that they are at least guaranteed not to make a program worse.[7] The worst that can happen is that the checker may miss a bug, or that a programmer will be briefly inconvenienced by a checker's false alarm. A corrector, on the other hand, can throw unpredictable, non-deterministic curves into program behavior, and can even undermine a perfectly good program if spawned by a mistaken checker. It may therefore be best not to go beyond mathematically rigorous complex correctors; more subjective correcting heuristics should be approached with care.

II. Applications to the Pentium Division Bug

In November, 1994, Thomas Nicely, a mathematician doing work meant to test the Pentium Microprocessor, discovered that Pentium division occasionally returns inaccurate values. This is a true "occasional error," in that less than one in eight billion inputs generates bad output. Nevertheless, while Intel argued that the error was so occasional as to be of no significance, a firestorm of controversy resulted. Even this small amount of uncertainty, it would seem, was too much for the general public. The discovery that Intel already knew of the bug, and had opted not to share this information, served to worsen the public-relations debacle.

The Pentium bug also generated something of a sensation in the computer science community. The problem of combating the bug was a mirror in which each scientist saw reflected his own research. We of the result-checking community were no exception. But the dominant suggestion for dealing with the bug was a software patch implemented by Cleve Moler, Terje Mathisen, and Tim Coe. They concluded—empirically, and appar-

[7] Of course, we must assure that the checkers do not seriously reduce program efficiency. Also, there should be safeguards to prevent the checkers from crashing the system or contaminating memory.

ently correctly—that the Pentium made errors only for inputs in certain easily recognized "bands." They also noted that, for any numerator/denominator pair N, D in one of these bands, multiplying both numerator and denominator by $\frac{15}{16}$ would translate the problem out of the band without changing the quotient. This, then, formed the basis for a simple, efficient corrector.

We found this solution unsatisfying, and felt that we should go beyond methodologies requiring previous knowledge of a specific bug. The public also seems to have disliked the idea of ad hoc software patching, and Intel was soon forced to provide replacement chips *en masse.*

Our desire was, rather, for general methods of checking and correcting *any* chip's arithmetic, before the exact nature of a bug is known and, indeed, before it is known whether or not there *is* a bug. We proceed, then, with a specification of how we would like to augment an arithmetic chip's multiplication and division functionalities with built-in checkers and correctors. It is our argument that microprocessors of the future, by employing checker/correctors of this sort, may protect themselves from embarrassments such as that which has humbled the Pentium.

A. A Simple Checker for Multiplication

We first consider the task of multiplication within a microprocessor such as the Pentium. Numbers are represented according to IEEE double-precision floating-point standard. Each number has a sign bit, an exponent, and a **normalized significand** on range $[1, 2)$: this significand is specified by 52 binary bits, $b_1, b_2, \ldots, b_{52}$, corresponding to value $1.b_1b_2 \ldots b_{52}$. Special cases also exist: zero; "not-a-number"; and ultra-small, denormalized numbers.

We assume that multiplication of input numbers A, B would be handled as follows. Special cases—e.g., one of A, B zero or not-a-number—may be handled separately. Ultra-small numbers may be shifted to look like ordinary numbers; we just remember a "debt" to be paid back to the exponent. We assume, in fact, that calculating the product's exponent is easy: this principally involves adding the exponents of A and B. Handling the sign bits is also easy. Thus, we assume the difficult part of the computation to be the following: *given two 53-bit numbers, each of form* $1.b_1b_2 \ldots b_{52}$*, find their 106-bit exact* *product.*[8] Once this is done, we round off to 53-bit normalized form, calculate the proper exponent, and, if necessary, convert back to ultra-small format.[9]

Let's consider, then, the problem of multiplying n-bit significands A and B—we can ignore the binary point and think of them as large integers—to get their $2n$-bit product. Sophisticated algorithms exist for solving this problem in $O(n \log n)$ time. However, these algorithms are generally considered impractical for microprocessors, which instead use a straightforward multiplication algorithm: they multiply A by the high-order bit of B, shift the result left, add in A multiplied by the next bit of B, shift the running sum left again, and so on. Implemented serially, this algorithm takes $O(n^2)$-time. Implemented on a chip, it requires n steps, at each of which we must employ an addition circuit at least n bits wide.

Then consider the following simple checker for multiplication.[10] Inputs are n-bit integers A, B and $2n$-bit C; we must decide whether or not $AB = C$. Pick a small random integer r; r should be about $\log n$ bits long. Calculate $(A \bmod r)$ and $(B \bmod r)$; multiply the resulting small numbers together, and again take the result $\bmod\ r$. As our test of whether or not C is correct, compare this residue to $(C \bmod r)$.

We will first illustrate this method via a simple example. Consider a processor with 5-digit decimal representations. Asked to multiply 2.5736 by 3.6239, this processor claims that the exact result is 9.32646904 (and so will give 9.3265 as its rounded output). To check this multiplication, we pick $r = 73$, and compare residues as shown in Fig. 4. The matching results suggest that the multiplication is indeed correct.

Why does this work? If $AB = C$, then $((A \bmod r)(B \bmod r) \bmod r) = (AB \bmod r)$ must equal $(C \bmod r)$ for any value of r. On the other hand, if $AB \neq C$, it *is* possible

[8] Indeed, it is in dealing with this most arduous stage of the computation that microprocessors use time-saving techniques such as lookups in tables of precomputed values. Such time-savers may prove complicated and potentially buggy; and it is our understanding that this is what happened in the Pentium.

[9] This formulation may be something of a cheat, in that we assume the exact 106-bit product to be calculated. In fact, a processor might try to dodge some of this work, since only the high-order half of the result will actually be needed. If we then have only the high-order half of the product available for our check, we face the more difficult problem of checking *approximate* multiplication rather than *exact* multiplication. As far as we know, this problem remains open.

[10] This is an application of a standard result-checking trick [10]. Moreover, it is a modern version of "casting out nines," a traditional check for arithmetic computations.

$$\begin{array}{ccc} 25736 & \times & 36239 \\ \bmod 73\Downarrow & & \Downarrow\bmod 73 \\ 40 & \times & 31 \\ & \downarrow & \\ & 1240 & \\ & \downarrow\bmod 73 & \\ & 72 & \\ & & \\ & 72 & \\ & \Uparrow\bmod 73 & \\ & 932646904 & \end{array}$$

Fig. 4. Checking multiplication, by a comparison of residues mod 73. Thick arrows correspond to the most arduous parts of the computation.

to pick an unfortunate value of r for which the residues will still match. However, it is readily proven that the probability (over the random choice of r) that this will happen is small.[11] Thus, if we repeat this check a few times, we can readily make the chance of error as small as desired.

Is this check computationally quicker and simpler than the original multiplication of A by B? Yes, for the check only requires that a pair of *small* numbers ($O(\log n)$ bits long rather than n) be multiplied, and that four numbers be divided through by a *small* number r. Such divisions take only $O(n \log n)$ time sequentially, or, when done on a chip, n steps, at each of which a small, quick, $O(\log n)$-wide circuit suffices. Thus, there is indeed substantial ground to regard this check as simpler—and hence, hopefully, quicker, less expensive in silicon, and more reliable—than the computation which it checks.

A final issue is the need for generation of a random number r, which could be inconvenient in a high-speed microprocessor. Pragmatically speaking, it might be best to record several

[11] Some details have been omitted. In particular, it is best that r be a random *prime.*

random primes on the chip and then use these repeatedly. While this approach is not as rigorous as the use of true randomness, it is unlikely that our hard-wired values of r will happen to correlate with the chip's bugs.

B. A Corrector for Multiplication

When our simple checker detects a multiplication error, the microprocessor might then circumvent this error by means of a complex corrector. We employ a standard method [4], one closely related to the matrix-multiplication corrector from Section I-F; however, we must here attend to details of legal input range in a context of limited-accuracy real numbers.

We assume that the essential task to be corrected is the multiplication of n-bit normalized significands A, B. We also assume that our chip is internally capable of multiplying two $(n+1)$-bit normalized significands and getting the exact, $2(n+1)$-bit product; this is reasonable, as arithmetic processors often use more bits of precision internally than they provide in their external representations. And finally, we assume that the chip returns the right $2(n+1)$-bit product for *most* inputs: say, for all but at most a one in a billion fraction of possible $(n+1)$-bit input pairs. It is easy to acquire this final assurance via a stage of random testing.

Now, at run-time, say that our chip fails on n-bit inputs A, B. Then pick R_1, R_2 randomly from the set of all possible n-bit normalized significands.[12] Calculate values $\frac{A+R_1}{2}$ and $\frac{B+R_2}{2}$; note that our extra bit of precision allows these numbers to be represented exactly. Also note that these numbers are each the average of two significands on legal range $[1,2)$, and so are also significands on this legal range. Then, as our corrected value for AB, calculate

$$4\left(\frac{A+R_1}{2}\right)\left(\frac{B+R_2}{2}\right) - 2R_1\left(\frac{B+R_2}{2}\right) - 2R_2\left(\frac{A+R_1}{2}\right) + R_1R_2.$$

What exactly is going on with this expression? First, note that

$$AB \;=\; \left[2\left(\frac{A+R_1}{2}\right) - R_1\right]\left[2\left(\frac{B+R_2}{2}\right) - R_2\right]$$

[12] Again, as in Section II-A, pragmatically speaking we might store a list of values for R_1 and R_2, rather than requiring fresh random bits each time.

R_1	$(A + R_1)/2$
1.0000	1.00101
1.0001	1.00110
1.0010	1.00111
$\vdots$	$\vdots$
1.1110	1.10011
1.1111	1.10100

Fig. 5. For $A = 1.0101$, values of $(A + R_1)/2$ as R_1 varies over all 5-bit normalized significands.

$$= \quad 4\left(\frac{A+R_1}{2}\right)\left(\frac{B+R_2}{2}\right) - 2R_1\left(\frac{B+R_2}{2}\right) - 2R_2\left(\frac{A+R_1}{2}\right) + R_1R_2.$$

Thus, our expression is indeed mathematically equivalent to AB.[13]

Second, how much work is needed to complete this calculation? The multiplications by $\frac{1}{2}$ or 2 or 4 are easy on a binary computer, and we assume addition and subtraction to be quick and reliable compared to multiplication. Thus, the computation reduces to doing four multiplications. This is a significant cost, but, as usual, we assume that it will only be needed in the rare case that the multiplier's original output is incorrect.

Finally, how does this calculation serve as a reliable self-corrector for multiplication? Note that, rather than calculating AB directly, we instead carry out four multiplications, each on inputs which are *essentially random* $(n + 1)$-bit significands. To see this, first recall that the inputs to the four multiplications are each one of R_1, R_2, $\frac{A+R_1}{2}$, $\frac{B+R_2}{2}$. R_1 and R_2 are known to be random, except that they are only n bits long; hence they are chosen randomly from exactly half of all possible $(n + 1)$-bit significands. Next observe that, for *any* A, as R_1 varies over all possible n-bit significands, $\frac{A+R_1}{2}$ varies over exactly half of all possible $(n + 1)$-bit significands. For example, if $n = 5$ and $A = 1.0101$, then, as illustrated in Fig. 5, $\frac{A+R_1}{2}$ varies uniformly from 1.00101 to 1.10100, which represents half of legal range $[1, 2)$.

Thus, consider any one of the four multiplications. We have shown that the multiplication is on a pair of significands each chosen uniform-randomly from half of all possible

[13] We are assuming sufficient bits of precision to calculate all of this arithmetic exactly, so there are no problems with round-off errors.

$(n+1)$-bit significands. Moreover, the two significands in the pair are statistically independent of each other: R_1 is independent of R_2, $\frac{A+R_1}{2}$ independent of $\frac{B+R_2}{2}$, and so on. Thus, the pair of significands is chosen uniform-randomly from a quarter of all possible input pairs. But we know from testing that our multiplier fails on at most a one in a billion fraction of all possible input pairs. Thus, for each of our input pairs, the quarter of the total domain from which the input pair is chosen can have density of "bad inputs" at most four in a billion. Hence the chance that *any* of our four multiplications will be erroneous is at most sixteen in a billion. And so our corrector will indeed return the correct value for AB with high probability.

To illustrate the use of our corrector, let us consider a processor which is required to multiply 5-digit normalized decimal significands on range $[1, 10)$; internally, however, it can multiply 6-digit significands to get their exact 12-digit product. Say that the processor fails to calculate the correct product of inputs $A = 2.5736$, $B = 3.6239$. Then we pick random $R_1 = 9.1257$, $R_2 = 5.0022$, and calculate as shown in Fig. 6. Note that the critical step is the calculation of four products, corresponding to the figure's four thick arrows. Each of these multiplications is on a pair of essentially random numbers. Thus, with high probability, all four multiplications will be done correctly, and our corrector will return 9.32646904, which is in fact the exact value of $2.5736 \cdot 3.6239$. We could finally verify this with our simple checker, then output rounded product 9.3265.

C. A Simple Checker for Division

We turn now to division. A dividing circuit, given n-bit numerator N and denominator D, may be expected to return Q, the quotient up to n bits, and a small remainder R. This output is correct if R is sufficiently small and

$$N = D \cdot Q + R$$

or, equivalently,

$$N - R = D \cdot Q.$$

But then checking division reduces to the same task as checking multiplication.

For example (Fig. 7), dividing 123.84 by 26.958 yields quotient 4.5938 and remainder 0.00033960. This output is correct if $123.84 - 0.00033960 = 123.8396604$ equals the product

$$
\begin{array}{ccc}
2.5736 & \times & 3.6239 \\
\downarrow & & \downarrow \\
\left[2\left(\frac{2.5736+9.1257}{2}\right) - 9.1257\right] & \times & \left[2\left(\frac{3.6239+5.0022}{2}\right) - 5.0022\right] \\
\downarrow & & \downarrow \\
[2(5.84965) - 9.1257] & \times & [2(4.31305) - 5.0022] \\
 & \downarrow &
\end{array}
$$

$$
\begin{array}{cccccccc}
4(5.84965\times4.31305) & - & 2(9.1257\times4.31305) & - & 2(5.0022\times5.84965) & + & (9.1257\times5.0022) \\
\Downarrow & & \Downarrow & & \Downarrow & & \Downarrow \\
4(25.2298329325) & - & 2(39.3596003850) & - & 2(29.2611192300) & + & 45.6485765400 \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
100.91933173 & - & 78.71920077 & - & 58.52223846 & + & 45.6485765400 \\
 & & & \downarrow & & & \\
 & & & 9.32646904 & & &
\end{array}
$$

Fig. 6. Correcting multiplication, by a transformation of AB to $4\left(\frac{A+R_1}{2}\right)\left(\frac{B+R_2}{2}\right) - 2R_1\left(\frac{B+R_2}{2}\right) - 2R_2\left(\frac{A+R_1}{2}\right) + R_1R_2$. Thick arrows correspond to those parts of the computation which are most arduous and whose correctness is most in question.

of 26.958 and 4.5938, which, by the method of Section II-A, we may verify to be true.

D. A Corrector for Division

We assume the arduous, unreliable part of division to be the calculation of an n-bit value for $N \div D$, where N and D are n-bit, normalized significands—i.e., on range $[1, 2)$. We also assume that, through a phase of random testing, we have assured that our chip calculates $1 \div X$ correctly for all but at most a one in a billion fraction of n-bit normalized significands X.

Our correcting procedure for $N \div D$ is then as follows: we generate R, a random n-bit normalized significand. We then return, as our corrected value for $N \div D$,

$$N \cdot R \cdot (1 \div (R \cdot D)).$$

Why does this work? First note that, if the three multiplications and one division all return correct results, this expression is indeed mathematically equal to $N \div D$—barring

	(123.84 − 0.00033960 =)	
123.84	123.8396604	4.5938
$\div$	$\div$	$\times$
26.958	26.958	26.958
$\|?$	$\|?$	$\|?$
4.5938 + remainder = 0.00033960	4.5938	123.8396604

Fig. 7. Transforming a problem of checking division (left column) to an equivalent problem of checking multiplication (right column).

problems of round-off error, which we shall discuss below. The three multiplications may be checked, as described in Section II-A, and, if necessary, corrected, as described in Section II-B. This leaves only the division $(1 \div (R \cdot D))$, which may be checked as described in Section II-C. If it is found to be incorrect, we can repeat the process with a new value of R. But we shall argue that the division will be correct with very high probability over the choice of R.

To see this, fix D. Then, as R varies over all n-bit normalized significands, the significand of $R \cdot D$ also varies quite broadly over the n-bit normalized significands. For example, in the case $n = 4$, $D = 1.101$, Fig. 8 illustrates the eight possibilities.

The reader may verify that, generalizing the pattern of Fig. 8, the significands of $R \cdot D$ must behave as follows when R ticks through all possible values from 1 to 2: the significands are at first strictly increasing; then, when $R \cdot D$ passes 2, the significands fall back to 1; their values are thereafter increasing again, with no more than two in a row equal; and we end on a significand no greater than the one on which we started. It readily follows that, for any particular value of the $R \cdot D$ significand, there are at most two distinct values of R which map to that significand.

But we know from testing that at most one in a billion significands X result in a bad calculation of $1 \div X$. Thus, for any D, at most two in a billion values of R result in a

R	$R \cdot D$
1.000	1.101
1.001	1.111
1.010	$1.000 \cdot 2^1$
1.011	$1.001 \cdot 2^1$
1.100	$1.010 \cdot 2^1$
1.101	$1.011 \cdot 2^1$
1.110	$1.011 \cdot 2^1$
1.111	$1.100 \cdot 2^1$

Fig. 8. For $D = 1.101$, 4-bit values of $R \cdot D$ as R varies over all 4-bit normalized significands.

bad calculation of $1 \div (R \cdot D)$. But this means that, each time we pick a value of R and run our self-corrector, its chance of failure is at most two in a billion, independent of the values of N and D.

A problem with the above corrector is that relative arithmetic error will build up over the three multiplications and one division required. However, a limit may readily be placed on this error: it is not difficult to see that the overall relative error is limited to approximately four times the relative error that the processor may experience in a single arithmetic operation. Our corrector's precision problem may thus be solved if we assume that the processor, while only required to provide n-bit arithmetic externally, uses several more bits of precision in its internal calculations. Then approximately $(n + 2)$ bits of precision within the corrector will suffice to keep the final output accurate to the required n bits.[14]

As a simple example, consider a processor with 5-digit decimal inputs and outputs, normalized significands on range $[1, 10)$, and 6-digit internal arithmetic. Then, to correct $N \div D$, where $N = 6.3109$ and $D = 2.9832$, we generate random $R = 4.41377$, transform $N \div D$ to $N \cdot R \cdot (1 \div (R \cdot D))$, and evaluate this expression as illustrated in Fig. 9. The three multiplications required may be tested and corrected; we have seen that, with

[14] Correspondingly, R should then be chosen randomly from the set of $(n+2)$-bit normalized significands, and our testing phase must assure that $1 \div X$ is calculated correctly for all but at most a one in a billion fraction of $(n+2)$-bit normalized significands X.

$$
\begin{array}{c}
6.3109 \div 2.9832 \\
\downarrow \\
6.3109 \times 4.41377 \times \frac{1}{4.41377 \times 2.9832} \\
\downarrow \\
6.3109 \times 4.41377 \times \frac{1}{13.1672} \\
\Downarrow \\
6.3109 \times 4.41377 \times 0.0759463 \\
\downarrow \\
6.3109 \times 0.335210 \\
\downarrow \\
2.1155
\end{array}
$$

Fig. 9. Correcting division, by a transformation of $N \div D$ to $N \cdot R \cdot (1 \div (R \cdot D))$. The thick arrow corresponds to that part of the computation whose correctness is most in question.

probability of error at most two in a billion, the division will be correct; and our use of 6-digit intermediate results assures that the final output will be accurate to 5 digits.

In closing, we observe that the above checkers and correctors must execute *within* the microprocessor, where they may isolate critical stages of a computation and may take advantage of the chip's additional bits of internal arithmetic precision. This suggests that hardware checker/correctors are indeed best suited to be embedded in the microprocessor itself, rather than being implemented as external software patches.

III. Conclusion

The life of a computer user is a perilous one. Imagine that you are using software to calculate matrix determinants. You type in a matrix; your workstation outputs a number. You hope this is indeed the determinant of your matrix—and if you have carefully tested your program, your confidence will be greater—but, in the final analysis, who knows? Perhaps something went wrong this time: a degenerate process may have taken place rather than the sophisticated computation you intended. The program's behavior at run-time remains a mystery. You must accept its correctness on faith; and that is contrary to scientific method.

Our hope is that result-checking will obviate this problem. Imagine now that you are using software supported by a powerful suite of checkers. You type in a matrix; the workstation outputs a number. The computer's behavior is identical to that above, and yet, to you, it may well seem different. For now you know that your program has completed its run without generating any checker warnings. Thus, a degenerate process could not have taken place: the silence of the checkers provides evidence that the correct computation did in fact take place beneath the mute facade of the machine. In place of black-box mystery, your experience is one of specific *assurances.*

Where are we in terms of the real-world testing of these theories? We argue that effective run-time checks may readily be added to software packages, and could prove an advantage in software testing and maintenance. This direction has been sporadically anticipated: indeed, simple run-time checks—e.g., checks that variables are in legal range—are sometimes employed in modern software development. But checking has never been given a full and sufficient trial. In our ongoing research ([6], [7]), we are taking first steps toward such a trial.

Checking simple hardware functionalities is, arguably, more immediately practicable. Despite the premium placed on optimizing microprocessor performance, it seems likely that, as hardware becomes ever faster and more complex, it will make sense for a commercial chip to embed a substantial suite of checkers and correctors. Indeed, as microprocessors grow in complexity, traditional testing may prove too arduous to be practical, so that checking and correcting will become a necessity.

For both hardware and software, we believe that checking is the way of the future.

References

[1] S. Ar, M. Blum, B. Codenotti, and P. Gemmell, "Checking approximate computations over the reals," *Proc. 25th Symp. Theory of Computing,* pp. 786–795, 1993.

[2] M. Blum, "Designing programs to check their work," *Int'l Computer Science Institute Tech. Report TR-88-009,* Dec. 1988.

[3] M. Blum and S. Kannan, "Designing programs that check their work," *Proc. 21st Symp. Theory of Computing,* pp. 86–97, 1989.

[4] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems," *J. Computer and System Sciences,* vol. 47, pp. 549–95, Dec. 1993.

[5] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," *Algorithmica,* vol. 12, pp. 225–244, Aug.–Sept. 1994.

[6] M. Blum and H. Wasserman, "Software reliability via run-time result-checking," http://http.cs.berkeley.edu/~blum/. Submitted to *Journal of the ACM.* Preliminary version: "Program result-checking: a theory of testing meets a test of theory," *Proc. 35th Symp. Foundations of Computer Science,* pp. 382–392, 1994.

[7] C. Boettcher and D. J. Mellema, "Program checkers: practical applications to real-time software," *Test Facility Working Group Conf.,* 1995.

[8] R. Butler and G. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Trans. Software Engineering,* vol. 19, pp. 3–12, Jan. 1993.

[9] R. Freivald, "Fast probabilistic algorithms," *Springer Verlag Lecture Notes in Computer Science #74: Mathematical Foundations of Computer Science,* pp. 57–69, 1979.

[10] R. Freivald, "Fast probabilistic verification of number multiplication," *Automatic Control and Computer Sciences,* vol. 13, no. 1, pp. 37–39, 1979.

[11] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson, "Self-testing/correcting for polynomials and for approximate functions," *Proc. 23rd Symp. Theory of Computing,* pp. 32–42, 1991.

[12] R. Rubinfeld, "On the robustness of functional equations," *Proc. 35th Symp. Foundations of Computer Science,* pp. 288–299, 1994.

[13] F. Vainstein, "Error detection and correction in numerical computations by algebraic methods," *Proc. 9th Int'l Symp. Applied Algebra, Algebraic Algorithms and Error-Detecting Codes,* pp. 456–464, 1991.

Manuel Blum studied at MIT (BS, 1959; MS, 1960; PhD, 1964), and holds the Arthur J. Chick Chair in EECS at the University of California, Berkeley. He joined the Berkeley faculty in 1968, and was Associate Chair for Computer Science from 1977 to 1980.

His publications have contributed to the study of fundamental algorithms, complexity theory, automata theory, inductive inference, cryptography, interactive proof, and result-checking.

Professor Blum is a Fellow of the IEEE Computer Society, the American Association for the Advancement of Science, and the American Academy of Arts and Sciences. He is the recipient of the 1995 ACM Turing Award.

Hal Wasserman has studied at Harvard University (BA, 1986), the University of Southern California (MA, 1993), and the University of California, Berkeley (MS, 1995). He is currently a PhD candidate at the University of California, Berkeley.

His interests are various and syncretic. His collaboration with Manuel Blum on result-checking includes consulting for Hughes Aircraft as well as "Software reliability via run-time result-checking" [6], which was presented in a plenary session of the 1994 IEEE Symposium on Foundations of Computer Science.

Affiliation of Authors

Manuel Blum holds the Arthur J. Chick Chair in EECS at the University of California, Berkeley, CA, e-mail: blum@cs.berkeley.edu, homepage: http://http.cs.berkeley.edu/~blum/. This work was supported in part by NSF grant CCR92-01092 and in part by a MICRO grant from Hughes Aircraft Corporation and the State of California.

Hal Wasserman is with the Computer Science Division, University of California, Berkeley, CA, e-mail: halw@cs.berkeley.edu, homepage: http://http.cs.berkeley.edu/~halw/. Supported by NDSEG Fellowship DAAH04-93-G-0267.

A preliminary version of the current paper appears in *Proc. 8th Int'l Software Quality Week,* 1995.