Batch Checking with Applications to Linear Functions

Ronitt Rubinfeld [*]
Department of Computer Science
Princeton University
Princeton, NJ 08544

May 2, 2009

Abstract

We introduce the model of *batch checking*, which allows one to check the result of a program on many inputs at once. We show that one can batch check programs for linear functions with very little overhead in the running time.

1 Introduction

In order to have faith that a program being used is giving the correct answers, a general theory of *checking* programs was introduced by Blum in [1] [3] [5]. This approach is concerned with the task of checking that a program gives the correct answer on a particular given input. In Blum's model, a very important idea is to allow the checker to call the program on other inputs while checking it on a given input (as long as the program is not assumed to be correct on the other inputs either). The checker may then output "PASS"

[*]This research was done while the author was at U.C. Berkeley and supported by NSF grant CCR 88-13632 and an IBM Graduate Fellowship.

if it thinks that the program is correct on the given input, or "FAIL" if it thinks that it has found a bug in the program - i.e. the program is wrong on some input, though not necessarily the given input. The checker may output either "PASS" or "FAIL" if the program is correct on the given input, but is incorrect on other inputs. This additional leniency has been used to construct very simple checkers for a surprising number of problems. However, since the program is being called on other inputs, there is often some overhead in the running time when using a checker.

Though many programmers are willing to spend some time overhead in order to verify that their programs give correct answers, for some applications, where efficiency is crucial, even a constant multiplicative time overhead makes checking undesirable. In this paper, we define a variant model of checking, called *batch checking*: Often greater efficiency can be achieved if the user does not need to know immediately whether the program gives the correct result. In this case, the checker can wait until the program has been called on several inputs and check that the program is correct on all of the inputs at once. Batch checking can allow greater efficiency, and we give examples of functions, such as the class of linear functions, for which batch checking allows one to reduce the overhead of the checking process to the point where it is arbitrarily small.

2 The Batch Checking Model

A program checker as defined by Blum [1] [3] [5] is a checker that checks whether a program is correct on a particular input. A batch checker is a checker that checks whether the program is correct on several inputs at once, and outputs "FAIL" if the program is incorrect on *any* of the inputs:

Definition 2.1 (probabilistic batch program checker) *A probabilistic batch program checker for f is a probabilistic oracle program R_f which is used to verify,* for any program *P that supposedly evaluates f, that P outputs the correct answer on several given inputs in the following sense. On given inputs $x_1, \ldots, x_m$ and confidence parameter β, R_f^P has the following properties:*

1. *If $\exists i$ such that $P(x_i) \neq f(x_i)$ then R_f^P outputs "FAIL" (with probability $\geq 1 - \beta$).*

2. *If P is a correct program for every input then R_f^P outputs "PASS" (with probability $\geq 1 - \beta$).*

The probabilities are with respect to the coin tosses of R_f. R_f is only allowed to access P as a black-box oracle. The batch checker should be simpler or at least different than any program for the function f. See [1][3] for a discussion of how to enforce this in a quantifiable way.

Often a batch checker can be made more efficient than calling the checker for each of the inputs separately. For example, a self-testing/correcting pair [4] can be used to construct a checker: use the self-tester (a program which verifies that a program for f is correctly computing f on most of the inputs) to test the program. If the program fails the test, output "FAIL" and halt, and if the program passes, use the self-corrector (a program which uses a program that correctly computes f on most inputs in order to correctly compute f on all inputs) to compute the correct result for the input being checked with high confidence, and compare the "correct" result to the output of the program on that input. Suppose the self-tester requires total time T, and the self-corrector requires total time S, then the incremental time is $T + S$. To check m inputs, rather than running the checker m times, for a total running time of $m(T + S)$, the tester need only be run once, giving a total running time of $T + mS$. Since T time is usually much larger than S (for example, the self-tester for the mod function makes several hundred calls to the program while the self-corrector makes fewer than 20), this savings can be quite significant.

3 Batch Checking for Linear Functions

The technique given in this section can be used to reduce the multiplicative overhead to arbitrarily close to 1 for any function with the linearity property as defined in [4]. Essentially the linearity property is the property that it is easy to compute $f(x)$ if one is given the values of $f(x_1), f(x_2)$ where $x_1 + x_2 = x$ for free:

Definition 3.1 (Linearity Property) *Suppose f is a function that maps a group G to a group H. We say that f is* linear *if, for* all *x and y in G, $f(x +_G y) = f(x) +_H f(y)$ where $+_G$ and $+_H$ are the group operations over G*

and H, respectively. $+_G$ and $+_H$ are assumed to be much simpler and more efficient to compute than the function f.

Examples of linear functions include integer multiplication, modular multiplication, integer division, the mod function and modular exponentiation.

Our batch checker uses the existence of checkers for any function that has the linearity property [4], but the batch checker works independently of how the checking is done. We present the specific batch checker that results from applying the technique to the mod R function. It is easy to see how to apply the technique to other linear functions.

The technique is based on the idea of Freivalds [7, Freivalds] used in the checker for matrix multiplication. Freivalds' idea was also used in a related manner by [6, Fiat Naor] where many modular exponentiation computations are verified by doing very few modular exponentiation computations.

3.1 Batch Checker for the Mod Function

The following program checks that a program P purporting to compute the function $f(x) = x \bmod R$ for $x \in [0 \ldots 2^n R]$ gives the correct answer on $x_1, \ldots x_m$. $Mod\_Result\_Checker(x, 1/4)$ is a checker that checks that P gives the correct answer on input x, with error probability at most 1/4.

We use $+_q$ and $\sum_q$ to refer to addition mod q.

Program Mod Function Batch_Checker$(n, R, x_1, \ldots, x_m, \beta)$

```
For $i = 1, ..., O(log(1/\beta))$ do:
  Randomly generate $m$-bit 0/1 vector $\alpha = (\alpha_1, \ldots, \alpha_m)$
  $sumin \leftarrow 0$
  $sumout \leftarrow 0$
  For $i = 1, ..., m$ do
    $sumin \leftarrow sumin +_{R2^n} \alpha_i \cdot x_i$
    $sumout \leftarrow sumout +_R \alpha_i \cdot P(x_i)$
  Verify that $sumout = P(sumin)$
  Call $Mod\_Result\_Checker(sumin, 1/4)$
  If verification fails or checker returns "FAIL" then
  output "FAIL" and stop.
Output "PASS".
```

Proof: [of correctness of batch checker] Since the mod R function is linear, $f(\sum_{R2^n} \alpha_i x_i) = \sum_R \alpha_i f(x_i)$. Thus, if P is always correct, the checker outputs "PASS".

We first use Freivald's technique to show that if there is an $l \in [1 \dots m]$ such that $P(x_l) \neq f(x_l)$, then with probability at least 1/2, $\sum_R \alpha_i P(x_i) <> \sum_R \alpha_i f(x_i)$ (and therefore the call to *Mod_Result_Checker* is likely to return 'FAIL'): Let $R = \{0,1\}^n$. Let k be such that $P(x_k) \neq f(x_k)$. Let the good set of α's (those for which the checker is likely to catch a mistake) $G = \{\alpha | \alpha \in R, \sum_R \alpha_i P(x_i) \neq \sum_R \alpha_i f(x_i)\}$ and the bad set $\bar{G} = R - G$. We show a $1-1$ mapping from $\bar{G}$ to G, showing that $|G| \geq |\bar{G}|$, and thus $Pr_\alpha[\sum_R \alpha_i P(x_i) \neq \sum_R \alpha_i f(x_i)] \geq 1/2$. For $\alpha = \alpha_1 \alpha_2 \dots \alpha_m \in \bar{G}$, note that $\bar{\alpha} = \alpha_1 \dots \alpha_{k-1} \bar{\alpha}_k \alpha_{k+1} \dots \alpha_m \in G$ ($\bar{\alpha}$ is just α with the i^{th} bit flipped). The mapping from α to $\bar{\alpha}$ is $1-1$.

Suppose $\sum_R \alpha_i P(x_i) \neq \sum_R \alpha_i f(x_i)$. If the verification that $sumout = P(sumin)$ passes ($\sum_R \alpha_i P(x_i) = P(\sum_{R2^n} \alpha_i x_i)$), we know that $P(\sum_{R2^n} \alpha_i x_i) =\neq \sum_R \alpha_i f(x_i) = f(\sum_{R2^n} \alpha_i x_i)$. Then the call to **Mod_Result_Checker** passes with probability at most 1/4. Thus the batch checker outputs "FAIL" with probability at least 1/8 after one iteration. The fact that $O(\log(1/\beta))$ iterations suffice follows from standard arguments. ■

Let $T(n)$ be the running time of P on inputs of size n. From the self--testing/correcting pair for the mod function, a checker for the mod function can be designed such that **Mod_Result_Checker** requires at most $c \cdot T(n)$ total time for constant c. The total work done by the batch checker is $(c+m)T(n) + O(n \cdot m)$. Since the program is called on all of the m x_i's regardless of whether any checking is done, the multiplicative running time overhead required by batch checking is less than $1 + \frac{c+1}{m}$.

4 Keywords

design of algorithms, program correctness

References

[1] Blum, M., "Designing programs to check their work", Submitted to *CACM*.

[2] Blum, M., Gemmell, P., Kannan, S., Naor, M.,

[3] Blum, M., Kannan, S., "Program correctness checking ... and the design of programs that check their work", *Proc. 21st ACM Symposium on Theory of Computing*, 1989.

[4] Blum, M., Luby, M., Rubinfeld, R., "Self-Testing/Correcting with Applications to Numerical Problems," ICSI Technical Report No. TR-90-041. *Proc. 22th ACM Symposium on Theory of Computing*, 1990.

[5] Blum, M., and Raghavan, P., "Program Correctness: Can One Test for It?", IBM T.J. Watson Research Center Technical Report (1988).

[6] Fiat, A., Naor, M., Personal communication through Moni Naor.

[7] Freivalds, R., "Fast Probabilistic Algorithms", Springer Verlag Lecture Notes in CS No. 74, Mathematical Foundations of CS, 57-69 (1979).