On the Cost of Recomputing:
Tight Bounds on Pebbling with Faults
Preliminary version

Yonatan Aumann1, Judit Bar-Ilan2, Uriel Feige3

1 Department of Applied Math., The Weizmann Institute, Rehovot, Israel.
Current address: MIT Laboratory of Computer Science, 545 Technology Square,
Cambridge, MA 02138, USA. aumann@theory.lcs.mit.edu
2 School of Library, Archive and Information Studies, The Hebrew University
Jerusalem 91904, Israel.
Part of this work was carried out while the author was with the Department of
Applied Math., The Weizmann Institute of Science.
3 Department of Applied Math., The Weizmann Institute, Rehovot, Israel.
feige@wisdom.weizmann.ac.il
Supported by a Koret Foundation fellowship.

Abstract. We introduce a formal framework to study the time and space complexity of computing with faulty memory. For the fault-free case, time and space complexities were studied using the “pebbling game” model. We extend this model to the faulty case, where the content of memory cells may be erased. The model captures notions such as “check points” (keeping multiple copies of intermediate results), and “recovery” (partial recomputing in the case of failure). Using this model, we derive tight bounds on the time and/or space overhead inflicted by faults. As a lower bound, we exhibit cases where \( f \) worst-case faults may necessitate an \( \Omega(f) \) multiplicative overhead in computation resources (time, space, or their product). The lower bound holds regardless of the computing and recomputing strategy employed. A matching upper-bound algorithm establishes that an \( O(f) \) multiplicative overhead always suffices. For the special class of tree computations, we show that faults can be handled with an \( O(f) \) additive factor in memory, and only a constant multiplicative overhead in time.

1 Introduction

We study the time and space complexities of computing in an environment with faulty memory. Consider a large scale computation task. In the course of the computation, many intermediate results are computed and stored in memory for later reference. Memory, however, is not fully secure, and at times the content of a memory unit may be lost (due to hardware failure, power failure, cosmic radiation, etc.). In this case, the computation must recover the corresponding data from elsewhere, possibly by retraceing previous computations. What is the overhead introduced by this recomputing? Is there an alternate order to perform the computation which will accelerate recovery? We may also consider
giving additional space to the program, in order to facilitate quick recovery. For example, essential data may be duplicated in memory, thus decreasing the chances of loosing it. A standard practice is to place periodic “check points” along the computation, down-loading a full copy of the entire memory. Later, the computation need only be recovered from this check point. Note, however, that these duplicate copies or check points may themselves be subject to failures. Whatever the method may be, can additional memory considerably decrease the recomputing time? How much additional space is necessary for these techniques to be effective?

We present a formal framework to study these questions. Previously, time-space complexity of straight-line programs was studied using the “pebbling-game” model [1, 2, 3, 4, 8]. We extend this model to the faulty case, allowing memory units (e.g. registers, disc sectors) to be erased during the computation. The faulty scenario is modeled as a two person game, with a computing player aiming to complete the computation with the space provided to him, and an erasing adversary player slowing him down by erasing memory units. The model assumes that all memory units are alike, and that when the content of a unit is erased, this is evident to the program. The model captures notions such as check-points and recomputing. We do not, however, allow for more sophisticated mechanisms such as error-correcting and alike.

Using this model, we establish tight bounds on the time-space complexity of the computation in the presence of faults. The main result is a negative one: there exist computations for which f faults incur an $\Omega(f)$ multiplicative factor overhead in the resources of the system (time, space, or their product). Thus, given a k factor additional space, the computation may still take $\Omega(1 + f/k)$ times longer. In particular, a computation which without faults completes with s space, when faced with f faults may necessitate as much as $\Omega(fs)$ space in order to keep the computation time linear. The lower bound holds regardless of the computing strategy used for the computation, and even if the program knows in advance the number of faults and is informed of all failures immediately as they occur. We note that the computations which exhibit this worst case performance are of a rather simple structure, and similar computations are frequently carried out in practice (e.g. FFT computations).

Matching to the lower bound, we present an effective algorithm which achieves the same asymptotic performance, for any computation. In essence, the algorithm calls for keeping “snap-shots” of the entire memory at fixed time intervals during the computation. These snap-shots then facilitate recovery in the case of faults.

For some computations, better performance can be obtained. We exemplify this by considering computations which have a tree-like structure. For this class of computations, we provide an algorithm which only necessitates an $O(f)$ additive factor space in order to overcome f faults. We match this with a corresponding lower bound for this class.

As we have previously stated, our main result is a negative one. We show that if we only allow to store duplicates of the intermediate results, and faults are to