

# An Empirical Study of Tracing Techniques from a Failure Analysis Perspective

Satya Kanduri and Sebastian Elbaum  
Dept. of Computer Science and Engineering  
University of Nebraska - Lincoln  
{skanduri, elbaum}@cse.unl.edu

## Abstract

*Tracing is a dynamic analysis technique to continuously capture events of interest on a running program. The occurrence of a statement, the invocation of a function, and the trigger of a signal are examples of traced events. Software engineers employ traces to accomplish various tasks, ranging from performance monitoring to failure analysis. Despite its capabilities, tracing can negatively impact the performance and general behavior of an application. In order to minimize that impact, traces are normally buffered and transferred to (slower) permanent storage at specific intervals. This scenario presents a delicate balance. Increased buffering can minimize the impact on the target program, but it increases the risk of losing valuable collected data in the event of a failure. Frequent disk transfers can ensure traced data integrity, but it risks a high impact on the target program. We conducted an experiment involving six tracing schemes and various buffer sizes to address these trade-offs. Our results highlight opportunities for tailored tracing schemes that would benefit failure analysis.*

## 1. Introduction

Program tracing is a dynamic analysis technique used to understand the behavior of a running program. A trace is essentially a footprint of specific software events that reflect program behavior. Software engineers trace their programs to perform various tasks, ranging from performance analysis to debugging.

As evident from Section 2.2, there is a large body of work describing how to efficiently instrument a program to obtain traces, while minimizing the overhead by optimizing the instrumentation and monitoring strategies. However, it is harder to find studies quantifying the impact of different tracing strategies to assist in the event of a failure.

Understanding the impact and associated trade-offs of a

tracing technique on failure analysis is extremely important given the considerable time spent by software engineers reproducing failures and finding the faults leading to those failures. Since traces are one of the key instrument available to make those tasks more effective and efficient, we need to pay attention on how traces are utilized in this context of failure analysis.

Tracing techniques have unique goals when considered in the context of failure analysis. More specifically, the integrity of the trace data becomes crucial in order to accelerate (or even make feasible) the failure reproduction or the fault finding process. However, a tracing technique always needs to minimize the possibilities of perturbing the traced program, avoiding the introduction of new behavior that could be considered faulty.

These considerations raise new questions. First, as evident from Table 1 and the slower response time of non-volatile devices, buffering as much of the trace as possible would always seem beneficial to minimize overhead on the target program. However, in the event of a failure, maintaining large amount of valuable traced data in a buffer could be detrimental for failure analysis activities due to the likelihood of traced data corruption (or complete loss). That is why buffering needs to work in combination with algorithms that trigger the transfer of the buffered data at certain times. How much trace data should be buffered in order to minimize overhead, and when should the data be transferred to (slower) permanent storage to minimize the risk of losing valuable information in the event of a failure constitute a set of challenging questions.

In this paper we start addressing these questions by studying how different combinations of trigger transfer algorithms and buffering strategies affect the information collected through a trace to perform failure analysis. We begin this process by evaluating a set of tracing techniques under different failure scenarios, specifically focusing on the amount of trace data loss, and their associated costs. Our efforts are then directed at finding tracing techniques that better fit the context of failure analysis, and also at making

Buffer Size	Clock time (in seconds)
1	423.4
8	54.0
64	7.0
512	1.1
4096	0.4
32768	0.3

**Table 1. Timings for reading a 1500 Kbyte file with different buffer sizes (adapted from [24]).**

engineers aware of the impact that tracing techniques could have on their daily activities.

In the next section we formally define the problem and present the related work. Section 3 introduces the trigger algorithms. Section 4 presents the experiment we designed and implemented to address our research questions. Section 5 and 6 present our findings. Finally, Section 7 summarizes this effort and introduces the future work.

## 2. The Tracing Problem

### 2.1. Definition

A trace is simply a sequence of events generated by an executing program that is being monitored. More formally, given program  $P$ , instrumented to capture the events  $e$  of type  $E$ , a new element is added to the trace  $tr_p$  every time  $P$  performs event  $e$ .

Tracing techniques normally use a buffer  $B$  [9, 11, 15, 19] to temporarily store the occurring events, and they also employ trigger algorithms to determine when to transfer the data to permanent storage. Buffering allows us to minimize the frequency of disk transfers,  $trF$ , which are generally  $10^5$  times [8] more expensive than memory accesses.

However, minimizing disk transfer overhead through larger buffers increases the risk of losing information if a failure occurs, which limits the effectiveness of the trace. Furthermore, environments with memory constraints might require a trace implementation that employs a fixed size circular buffer, which might lead to buffer overwrites with its associated loss of events.

Hence, choosing the technique that better combines buffer size and trigger algorithm can have a critical impact in the performance of the target application and in the integrity and usefulness of the collected trace to assist in the analysis of a software failure. Our work focuses on tracing techniques presenting several combinations of buffer size and trigger algorithm to adjust transfer frequency.

More formally, this problem can be defined as follows. Given  $BS$ , a set of buffer sizes,  $TA$ , a set of trigger algorithms, we define the cartesian product of  $BS$  and  $TA$

that results into a set of tracing techniques, as  $BT$ . Assuming a function  $g$  from  $BT$  to the real numbers, the *Tracing Problem* can be defined as finding a combination  $C$  in  $BT$  such that  $(\forall C') (C' \in BT) (C \neq C') \rightarrow g(C) \geq g(C')$ . For simplicity we assumed that  $g$  assigns a higher value to the combination that maximizes just one criterion.

### 2.2. Related Work

*Tracing* is widely used for capturing and replaying execution [18, 26], for program analysis and testing [2, 7, 20], for monitoring distributed systems [10, 16, 21], and for measuring performance [12]. With such a diverse and large application domain, traces have received significant attention, especially on how to generate more efficient techniques to trace a program execution. For example, specialized hardware monitors have been developed that could be used for program tracing [5]. These devices can transparently monitor a program, which diminishes the drawbacks of employing software traces. However, their lack of flexibility constitutes an enormous limitation. For example, they cannot be used to monitor a system that is going to be migrated to several platforms. Furthermore, they have trouble keeping up with the current processors due to their speed and built-in cache mechanisms.

That is why software instrumentation is still prevalent today, mainly due to its flexibility and low cost. Researchers have taken several approaches to optimize the generation of traces through software instrumentation. The first approach is to minimize the amount of instrumentation required to collect trace information by performing selective profiling and tracing [1, 3, 4]. The idea is to monitor a minimal number of events, which makes the tracing process more efficient. Some efforts go even further by dynamically adjusting the events to collect and when to store them [18] or even removing the instrumentation as the program executes [20]. A second approach is through the use of buffering strategies [11], which are often included in commercial software [9, 19]. The goal of this approach is to minimize disk transfers by collecting as much information as possible in a temporary buffer. Within buffering strategies there have been also efforts to encode or compress the trace data to maximize the buffer space [22].

A third approach consisted in the optimization of transfer schedules to minimize the slow down produced by tracing. These studies were mainly performed within the context of checkpointing techniques for fault-tolerance and recovery [21, 26]. The problems analyzed under these efforts have similar goals to ours. However, there are significant differences. First, traces attempt to provide a continuous stream of events, while checkpoint provides a snapshot of the system at a given time. This leads to different trade-offs. For example, although traces and checkpointing techniques at-

tempt to minimize performance impact, their considerations are different because snapshots require the transfer of large images in a few attempts, while the stream of trace events usually require small and frequent transfers. Second, our evaluation will be focused on the impact of the different techniques in the trace capability to assist failure analysis activities.

### 3. Trigger algorithms

In this section we present the trigger algorithms that force the transfer of traced data to permanent storage. We devised six trigger algorithms that expose different trade-offs in the presence of various buffer sizes. The algorithms include an optimal algorithm that represent the best we can perform in terms of data loss, a control algorithm that represent the common practice, and four algorithms that combine adaptive techniques and the use of thresholds.

#### 3.1. Optimal

The optimal algorithm generates no data loss by performing the transfers at optimal intervals, without consideration for cost. In order to do that, the algorithm employs knowledge about the exact failure occurrence. Although this technique is not practically feasible, it serves as a lower bound for data loss.

#### 3.2. Buffer Full

The buffer full algorithm constitutes the control treatment, the one most commonly found in tracing techniques. This simple algorithm transfers data whenever a buffer is full, which makes it very predictable.

#### 3.3. Non-core event adaptive

We define adaptive algorithms as the ones that change their transfer frequency behavior based on the observed events.<sup>1</sup> In particular, event adaptive algorithms perform a transfer every time a new (not encountered earlier) event is found in the trace. For example, every time an uncovered function is invoked. Then, this type of algorithm will likely decrease the number of transfers over time as the software runs, unless a new event occurs. Since the likelihood of failure is often associated with new events, this algorithm has the potential of losing less traced data in the event of a failure.

---

<sup>1</sup>Note that our definition of adaptive is different from [18] in that we adapt the transfer frequency instead of the events to capture or the window of observation.

We refined the event adaptive algorithm by incorporating the concept of core events. Core events are the ones occurring upon the invocation and immediate termination of a program. For instance, the events occurred when a word processor is opened with a blank document and immediately terminated without any other operations are the core events of that word processor. All the events that are not core events are non-core events. This refinement minimizes the initial number of transfers due to the occurrence of core events.

#### 3.4. Non-core event adaptive with thresholds

This algorithm is similar to the non-core event adaptive algorithm with the addition of thresholds. The incorporation of thresholds let us diminish the frequency of transfers in the event of a spurt of new events, and it also limits the amount of traced data collected without a transfer.

We define two thresholds: a *min* threshold which is *factor* times smaller than the buffer size, and a *max* threshold which is *factor* times larger than the buffer size. Upon the occurrence of a new non-core event, this algorithm waits for *min* events to be accumulated in the buffer (if they are not already there) before triggering a transfer. This is analogous to having a "lazy" transfer. In addition, a disk transfer is forced if *max* events are accumulated and not yet transferred, which can minimize the amount of data loss in the event of a failure.

#### 3.5. Least frequent events

This algorithm is devised based on the premise that the infrequent events are more likely to be associated with a failure than the relatively frequent ones. Under this algorithm, a transfer is executed when the less common events occur. As a result, the least frequent event algorithm is expected to minimize the number of transfers, but also have a small data loss in the event of a failure. This algorithm uses an event list probability (generated prior to the program execution) to determine the set of least frequent events. Without loss of generality, we arbitrarily determine that an event in the lower quartile of the probability list was an infrequent event.

#### 3.6. Least frequent events with thresholds

The least frequent events with thresholds constitutes an extension of the previous algorithm. For details about the threshold implementation observe the non-core event adaptive with thresholds algorithm.

## 4. Experiment

We start the experimental section by defining the variables we observe, and the metrics we employ to quantify them. Then, we proceed to refine the goal of the experiment by specifying the research questions in terms of the metrics we collect. Next, we introduce the materials used and the experimental design we employ to address our research questions. The last part of this section presents the threats to the validity of our experiment and the tasks we performed to control those threats.

### 4.1. Variables and Metrics

#### 4.1.1. Independent Variables

The independent variables in this experiment are trigger algorithms and buffer size. The six trigger algorithms we employ were presented in Section 3. We define buffer size,  $max_{bs}$ , as the maximum number of events that can be retained in memory at a given instance. Given that we are working with fixed size buffers, measuring buffer size is a simple task conducted before program execution. The experiments will be performed using six buffer sizes.

#### 4.1.2. Dependent Variables

We are interested in two types of dependent variables. First, we would like to know how much information is lost due to the selected buffer size and trigger algorithm. This aspect quantifies the effectiveness of the tracing technique. Second, we are interested in measuring the cost to determine the efficiency of the technique. In order to capture loss and cost we introduce the following three measures. These metrics use the notion of a failure scenario which will be described in this section under Experimental Setup.

- **Overwrite Induced Loss (OIL):** This is the total number of events lost due to buffer overwrite, a common situation when traces are implemented through circular buffers. A buffer overwrite will occur when the number of traced events between disk transfers exceeds the  $max_{bs}$  (buffer size).

Given  $trF$ , the total number of disk transfers,  $dt_j$ , a particular disk transfer (among  $trF$  transfers), and  $E_{dt_j}$ , the number of events that occurred between disk transfers  $dt_{j-1}$  and  $dt_j$ , we compute OIL in a particular disk transfer as:

$$OIL_{dt_j} = \begin{cases} \text{if } E_{dt_j} > max_{bs} \text{ then } E_{dt_j} - max_{bs} \\ \text{else } 0 \end{cases} \quad (1)$$

Given a particular failure scenario  $FS_i$ , its total OIL is:

$$OIL_{FS_i} = \sum_{i=1}^{trF} (OIL_{dt_i}) \quad (2)$$

For the optimal and buffer full algorithms, OIL will always be 0 because they prevent overwrite. However, the rest of the algorithms are likely to incur in overwrites as they minimize disk transfers. For example, the non-core event adaptive algorithm could only have an OIL of 0 if a new non-core event is observed before the buffer becomes full (e.g, every  $max_{bs}$  events).

- **Failure Induced Loss (FIL):**

This is the number of events elapsed between the last disk transfer and the occurrence of a failure. From a purely failure analysis effectiveness perspective, FIL is perhaps the most important metric we are collecting. It represents the risk of losing essential events in the vicinity of failure, which are likely to be the most valuable in failure reproduction and fault finding. Given  $TE$ , the total number of events before a failure,  $dt_j$ , a particular disk transfer,  $E_{dt_j}$ , the number of events that have occurred between the disk transfers  $dt_{j-1}$  and  $dt_j$ ; the expression for FIL of a particular failure scenario  $FS_i$  is given by:

$$FIL_{FS_i} = TE_i - \sum_{j=1}^{trF} (E_{dt_j}) \quad (3)$$

Per design, the optimal algorithm has a best-case and worst-case values of 0 for FIL. The other algorithms could range from 0 when the failure immediately follows the latest disk transfer, to  $TE_i$ , when the algorithm fails to trigger the transfer at all. For example, under the least frequent event algorithm, FIL is likely to increase if the sequence of events leading to the failure were relatively common events.

Note that, as defined, FIL may account for some events that are overwritten in the buffer. However, since those events would have been lost even in the presence of an infinite buffer (OIL would have been 0), it corresponds to include them within the FIL metric.

- **Cost:**

To quantify the impact of our tracing techniques, we measured the cost in function of the number of disk transfers and the amount of traced data transferred. Although there are other types of costs, we focus just on the transfers because they are the most significant. We

identified two type of transfer costs: fixed and variable. Each transfer has a fixed cost that includes from moving the head of the drive to the proper location, to finding the proper inode to get a file pointer that can be manipulated. A block of transferred data, on the other hand, has a variable cost that depends on the block size.

Let us define  $C$ , the fixed cost per disk transfer (cost in terms of the excess time taken to write to disk than to write to a buffer), and  $c$ , the variable cost per event block transfer (cost to actually write an event to disk). Given a disk transfer  $dt_j$ , and the  $E_{dt_j}$  events that have occurred between the disk transfers  $dt_{j-1}$  and  $dt_j$ , the equation for  $Cost$  for failure scenario  $FS_i$  can be stated as follows.

$$C_{FS_i} = C * trF + \sum_{j=1}^{trF} (c * Events_{dt_j}) \quad (4)$$

where

$$Events_{dt_j} = \begin{cases} \text{if } E_{dt_j} < max_{bs} \text{ then } E_{dt_j} \\ \text{else } max_{bs} \end{cases} \quad (5)$$

The worst case cost scenario consists of singly transferring every traced event to disk. That would be the case in the absence of buffering. Given that  $C$  is approximately  $10^5$  times larger than  $c$  [8], trigger algorithms performing less frequent but larger transfers are preferable.

## 4.2. Research Questions

Now that the metrics and definitions are in place, we refine the goal of the experiment by restating it in the form of research questions.

**RQ1:** What is the effect of buffer size and transfer frequency on the amount of data loss for different failure scenarios as measured by OIL and FIL?

**RQ2:** How does cost vary among tracing techniques?

**RQ3:** Under what type of scenario does each technique excel? and under what scenarios does a technique behave poorly?

## 4.3. Experiment Setup

In order to evaluate the combination of trigger algorithms and buffer sizes, we selected a target program to be traced and created a set of sample failure scenarios.

The program we selected for the experiment is Vim. Vim, short for Vi IMproved, is an open source editor based on the popular UNIX Vi editor. Apart from inheriting all the features of the standard Vi, it incorporates additional functionality such as unlimited undo, syntax coloring, and split windows. We selected Vim because its source code is easily accessible for experimentation, its size is representative of many products in the market (over 80KLoc), and it can be run indefinitely which let us obtain large traces. The events we are observing are function invocations. In order to capture those type of events, we instrumented Vim using our software analysis tool (CLIC [6]). A total of 1402 possible distinct function invocation events can be generated by Vim.

To generate the traces we took the following considerations. First, we wanted to be able to repeat the same execution to generate the same trace, while just varying the tracing technique being evaluated. Second, we wanted to deal with large enough traces that approximate long running programs to minimize the threats to external validity. Third, we wanted to select a variety of behaviors leading to different failures scenarios so that the evaluation was not biased toward specific circumstances. Then, employing the test suite that accompanied Vim as a starting point was a reasonable choice.

The test suite was composed of 32 coarse test cases. Each test case generates a trace that ranges from 20,000 to 580,000 events approximately. We then proceeded to create failure scenarios by forcing a software failure in each one of those test cases generating a trace. We simulated the existence of a fault in certain functions to generate the failures. More specifically, we created a list of functions executed by at least one test case, but less than 5% of all the test cases. This list had 133 functions that we then considered faulty. Next, as the program was executing and the trace was being generated, we forced the program to fail when one of those functions was executed. Since we are focusing on post-release failure analysis, the assumption is that if a fault was present in the code, it was more likely to be located in the functions that were not extensively executed by the test suite.<sup>2</sup> Since the functions in the faulty list are not executed by 11 of the 32 test cases, we ended up with 21 viable failure scenarios for experimentation.

## 4.4. Design

To address our research questions we have designed an experiment with various combinations of buffer sizes, trigger algorithms and failure scenarios.

<sup>2</sup>We are aware that this is a simplification of how a fault becomes a failure, and that execution is one of the elements required for a fault to be exposed [25]. Details about the impact of this implementation is presented in the threats section.

We have implemented six trigger algorithms (Section 3), used six distinct buffer sizes, and 21 failure scenarios. For techniques using thresholds, we established the threshold factor to 10. We also had to carefully choose the buffer sizes as they have a significant impact on the results. A small buffer size may bias the results towards a high OIL, while a large buffer size may bias the results towards a high FIL. Moreover, extremely large buffer sizes may introduce noise into our results due to operating system dependencies like paging. Last, we intended for the results to be applicable to other programs so we had to set buffer sizes relative to program size. Hence, we decided to use buffer sizes of 25%, 50%, 100%, 200%, 500% and 1000% of the number of distinct events in the program (1402 in this case).

Our experiment adheres to a full factorial design consisting of 3 factors(trigger algorithm, buffer size, failure scenario). We employed six levels for trigger algorithm and buffer size. We decided to consider failure scenario as a random factor (instead of fixed like the other two) since failure scenarios are likely to change if this experiment is replicated on other programs. This experimental design results in 756 data points. The analysis of these data points in relation to each one of the three dependent variables is presented in Section 5.

## 4.5. Threats to Validity

### 4.5.1. Internal Validity

Threats to internal validity are influences that can affect the independent variables with respect to causality, without the researcher's knowledge [27]. The size of the traces generated by Vim test cases may have biased the results because our trigger algorithms assume to be working with long-running programs. We controlled this threat by ensuring that all traces at least cover the maximum buffer size. Also, the results obtained may have been affected by personal bias given that a single person conducted the entire experiment including failure simulation.

Another threat to internal validity is the failure simulation process we employed. This process was practical and convenient to perform various long running experiments. However, the placement of faults on the functions rarely executed by the test suite can be questionable because executing a function does not mean that the faulty statement was executed, and even if it was, the fault might not have been exposed. This procedure, although it does not affect the optimal and buffer full algorithms, can benefit the non-core event adaptive and least frequent events algorithms. We partially controlled this threat by determining the rarely executing functions based on the probability generated by the whole test suite, while the techniques only looked at each particular scenario (a test case) which constitutes a subset of that universe.

Last, the infrastructure required to execute this study involves many tools. Although some of the tools were used and tested in previous experiments, some new tools were written to gather, filter and process trace information. We manually verified results and ran the study repeatedly to obtain confidence in the new tools' correctness.

### 4.5.2. Construct Validity

Threats to construct validity are significant when the measurement instruments do not capture the concepts they are supposed to capture [27]. Since we are interested in the impact of traces on failure analysis activities, the FIL metric is probably the most likely to capture that aspect of the tracing techniques. The approach is based on the notion that techniques that provide the most complete trace are more likely to help a software engineer performing any type of failure analysis. When combined with the Cost measure we can also analyze the trade-offs between effectiveness and efficiency. Still, there other possible ways and metrics to quantify the impact of the independent variables. Our metrics are just a subset of those possible measures.

### 4.5.3. External Validity

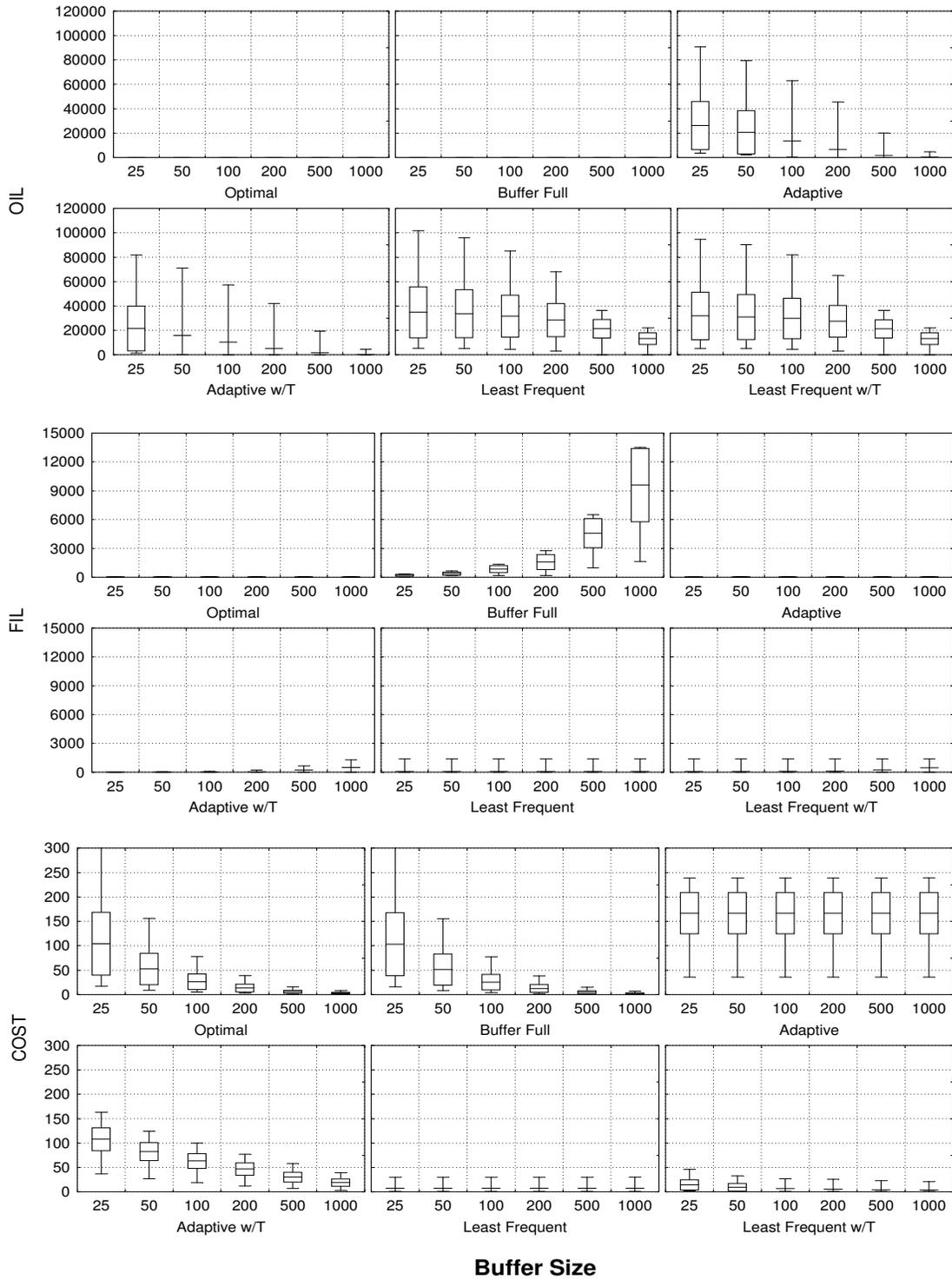
Threats to external validity are conditions that limit researcher's ability to generalize the results of his/her experiment to industrial practice [27]. The program chosen may not be enough to represent the general population. This poses a threat to external validity as the data obtained may be a characteristic of the program or failure scenario chosen and hence cannot be generalized.

The failure simulation scheme that we utilized has several limitations. Failures are simulated in infrequent events under the assumption that they are more likely to fail than relatively frequent ones. We also made assumptions about the context in which these techniques are employed. We assumed that main memory is finite and storage is infinite which may not be true in some situations. We have also analyzed our tracing techniques in isolation, independent of the operating system's services (e.g., virtual memory, paging). Last, our trigger algorithms work with fixed size buffers. All these factors limit the generalization of our results.

### 4.5.4. Conclusion Validity

Last, we encountered threats to conclusion validity, which in our case would be to find a relationship significant when it was not. To avoid this possibility we established the hypotheses related to our research questions up-front that clearly defined what we were targeting, we used as many failure scenarios as possible, and we controlled the assumptions for the statistical techniques we employed.

**Figure 1. Box plots for OIL, FIL, and Cost for the six trigger algorithms under six buffer sizes. Due to space constraints we have used shorter names for the algorithms: Adaptive is non-core event adaptive, Least Frequent is least frequent events, and w/T means with thresholds. The Cost scale represents millions.**



## 5. Results and Analysis

To provide an overview of all the collected data we generated the box plots included in Figure 1. The box plots provide a concise display of a distribution across the different trigger algorithms presented in Section 3, under varying buffer sizes (as described in Section 4), for each one of the metrics we defined. The central line in each box marks the mean value. The edges of the box mark the standard deviation. The whiskers mark the minimum and maximum.

The first 2 rows of graphs depict the effects of trigger algorithm and buffer size on OIL. As expected per definition, the values for the optimal and the buffer full algorithms were 0, independently of buffer size. For the rest of the trigger algorithms, the box plots suggest that OIL decreases with an increase in buffer size. It can also be perceived that their variance also decreases steadily with an increase in buffer size. The adaptive algorithms seem to lose less data due to overwrite than the least frequent event techniques, and the addition of thresholds seemed to be beneficial for both sets of algorithms.

The third and fourth rows of Figure 1 present the plots for our main measure of impact on failure analysis: FIL. Upon observing these plots, we note that the buffer full algorithm (one of the most efficient in terms OIL) is not as effective in terms of FIL, and it gets worse as the buffer size increases. Non-core event adaptive and least frequent events algorithms do not seem to depend on buffer size and consistently have lower FIL than our control algorithm. This fact is evident from their corresponding box plots. In fact, non-core event adaptive algorithm is comparable to optimal in terms of FIL. The addition of thresholds seems to slightly increase FIL for larger buffer sizes.

Last, we observe the cost box plots in the lower part of Figure 1. We only accounted for variable cost to minimize nuisance introduced by the use of different equipment and configuration. Non-core event adaptive and least frequent events algorithms have a steady cost. However, the non-core event adaptive algorithm which had a low FIL, costs more than the optimal algorithm which denotes the existence of many unnecessary transfers. The mean cost and variance of the other algorithms generally decrease as buffer size increases. The plots suggest that least frequent events algorithm has a steady low cost, and the addition of thresholds is likely to result in even lower costs for large buffer size (thresholds were computed as a percentage of buffer size).

We now proceed to formally analyze the collected observations through an analysis of variance (ANOVA) to determine if there are significant differences between the means of our trigger algorithms and the buffer sizes we selected [17]. We performed an ANOVA for each one of the three metrics. The summary of these analyses is presented in Ta-

bles 2, 3, and 4. Each one of these tables has seven columns. The first column represents the effect being investigated. We are investigating 5 effects, one per row, including the main effects, the interaction between trigger algorithm and buffer size, and the error term in the last row. The second column in the tables reminds us whether the effect is fixed or random (the only random effect we defined was failure scenario). The third to sixth columns present the sum of squares, the degrees of freedom, the means squared, and the F values as reported by our statistical tool. The last column presents the p values, which indicate if the differences are statistically significant (we set alpha to 0.05).

Table 2 confirms our observations from the box plots (note that since optimal and buffer full did not present any variation we did not include them in this ANOVA and that is why the degrees of freedom for algorithms indicates three). That is, the results indicate that there is enough statistical evidence to reject the null hypothesis stating that the means were the same. This was the case for each one of the effects' means. In other words, all the means are significantly different at the 0.05 level. More specifically, the trigger algorithm and the buffer size have a significant influence on OIL. We also found that there is significant interaction between these effects. That is, the behavior of the trigger algorithms varied considerably among buffer sizes. Last, the failure scenario also had a significant impact on OIL indicating that the appropriateness of certain algorithms and buffer sizes would vary depending on the failure scenario (when does the failure occur). Tables 3 and 4 also present significant differences for all the effects and due to space constraints we will save their detailed explanation. Though the distribution of OIL, FIL and Cost is non-normal, we performed ANOVA on them as the F statistic is robust to non-normality [13, 14, 23].

Since the ANOVAs showed significance for the main effects of interest, we now would like to know which algorithm or buffer size contributed the most to that difference, or better yet, how the algorithms and buffer sizes are different from each other. For example, for FIL, the ANOVA showed a significant difference due to trigger algorithm. However, the only algorithm that seemed quite different in the plots was buffer full. We would like to formally confirm that type of observation. In order to do that we performed a Bonferroni procedure which is a post-hoc test that allows us to perform this type of analysis [17].

The Bonferroni results for each one of the metrics (ranked by their mean) across trigger algorithms are presented in Tables 5, 7, and 9. The Bonferroni results discriminating based on buffer size are presented in Tables 6, 8, and 10. The last column of these tables has letters indicating the grouping among different levels of an effect so that, if they are not significantly different, they will have the same letter.

**Table 2. ANOVA on OIL**

Effect	(F/R)	SS	d.f	MS	F	p
Trigger Algorithm	Fixed	33413067E3	3	11137689E3	218.57	0.00
Buffer Size	Fixed	29029329E3	5	58058657E2	113.93	0.00
Failure Scenario	Random	69281831E3	20	34640916E2	67.98	0.00
Trigger Algorithm * Buffer Size	Fixed	27281684E2	15	181877894	3.56	0.00
Error	Fixed	23439824E3	460	50956141		

**Table 3. ANOVA on FIL**

Effect	(F/R)	SS	d.f	MS	F	p
Trigger Algorithm	Fixed	772477097	4	193119274	313.05	0.00
Buffer Size	Fixed	337444463	5	67488893	109.40	0.00
Failure Scenario	Random	27433145	20	1371657	2.22	0.00
Trigger Algorithm * Buffer Size	Fixed	10789562E2	20	53947810	87.45	0.00
Error	Fixed	357791641	580	616882		

**Table 4. ANOVA on Cost**

Effect	(F/R)	SS	d.f	MS	F	p
Trigger Algorithm	Fixed	2255766	5	451153	1331.70	0.00
Buffer Size	Fixed	217801	5	43560	128.58	0.00
Failure Scenario	Random	226289	20	11314	338.78	0.00
Trigger Algorithm * Buffer Size	Fixed	198483	25	7939	338.78	0.00
Error	Fixed	236807	699	339		

**Table 5. Bonferroni test - Trigger Alg. on OIL**

Trigger Algorithm	Mean	Group
Non-core adaptive with thresholds	9097	A
Non-core adaptive	11506	B
Least frequent events with thresholds	25753	C
Least frequent events	27178	C

**Table 8. Bonferroni test - Buffer Size on FIL**

Buffer Size	Mean	Group
25	68	A
50	101	A
100	199	A
200	355	A
500	1017	B
1000	2116	C

**Table 6. Bonferroni test - Buffer size on OIL**

Buffer Size	Mean	Group
1000	6794	A
500	11440	B
200	16866	C
100	21315	D
50	25268	E
25	28619	F

**Table 7. Bonferroni test - Trigger Alg. on FIL**

Trigger Algorithm	Mean	Group
Non-core adaptive	0	A
Least frequent events	66	A
Non-core adaptive with thresholds	130	A
Least frequent events with thresholds	166	A
Buffer Full	2855	B

**Table 9. Bonferroni test - Trigger Alg. on COST**

Trigger Algorithm	Mean	Group
Least frequent events with thresholds	7	A
Least frequent events	7	A
Buffer Full	33	B
Optimal	33	B
Non-core adaptive with thresholds	58	C
Non-core adaptive	166	D

**Table 10. Bonferroni test - Buf. Size on COST**

Buffer Size	Mean	Group
1000	34	A
500	36	A B
200	42	B
100	49	C
50	62	D
25	83	E

Table 5 shows that non-core event adaptive with thresholds significantly outperforms the other algorithms, non-core adaptive ranks second, and the least frequent event ranked last and are not significantly different. Table 6 shows that the means of OIL when discriminated by buffer sizes are significantly different, with larger buffer sizes generating a lower loss due to overwrite. Table 7 shows that buffer full is significantly worse than the rest in terms of FIL. Table 8 shows that buffer sizes of 500% or more tend to have a significantly larger FIL. Last, we see in Table 9 that the least frequent event algorithms are significantly cheaper than the rest and that the adaptive algorithm without thresholds is the most expensive. And, in Table 10 we note that as we move to larger buffers the cost often decreases significantly.

## 6. Discussion

Section 5 presented and analyzed the effects of trigger algorithm and buffer size on our dependent variables. The analysis indicated that the independent variables had a significant effect on OIL, FIL, and cost. However, the analysis did not provide any information about the trade-offs associated with the selection of a trigger algorithm. As it was evident in Section 5, no single algorithm is effective in all circumstances. Hence, the motivation behind this discussion is to elucidate some specific trade-offs that need to be considered when selecting a trigger algorithm.

Figure 2 presents a scatter plot depicting the relationship between cost and FIL for the six algorithms with a buffer size fixed at 100%. We decided to employ just FIL because it is the metric that is most likely to reflect the ability of the trace to assist in failure analysis activities. Then, each observation in the figure represents the result of running a tracing algorithm on a given failure scenario with the fixed buffer size. The figure seems to indicate several trends. First, if FIL is the primary concern for a practitioner, any algorithm other than buffer full would generate relatively low values. Buffer full not only presents a large range of values, but even its minimum FIL value is close to 200. This means that the last 200 events before the failure will not be available to the software engineer, making the trace much less

valuable.

On the other hand we note in Figure 2 that non-core event adaptive always approximates to 0. However, this comes at a cost, with values starting at approximately 150K. If cost is then a major consideration, then this algorithm may not be a viable choice. In that case, least frequent events algorithm is recommendable due to its steady low cost (across all buffer sizes) and still comparable FIL. It is also interesting to observe the impact of adding thresholds to the base algorithms. The addition of thresholds tend to increase FIL slightly, but also diminishes the cost associated with the based algorithms. Furthermore, the addition of thresholds make the algorithm performance bounded which might be an important consideration for their usage.

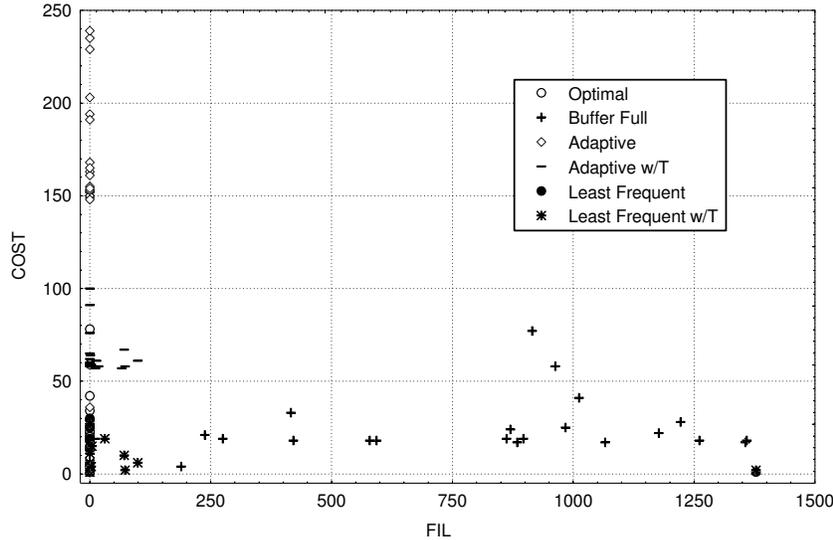
## 7. Conclusions and Future Work

We have presented an empirical study exposing the trade-offs that emerge from the choice of different trigger algorithms and buffer sizes. More specifically, we focused on the impact of these factors on the effectiveness and efficiency of tracing techniques from the perspective of the failure analysis activities regularly performed by software engineers. As we have discussed, these type of empirical studies, like any other, have several limitations to their validity. Keeping these limitations in mind, we draw several observations from this work.

First, our results indicate that the adaptive and least frequent algorithms are significantly superior (as measured by FIL) than the commonly used buffer full algorithm. This implies that these relatively simple algorithms are likely to be more effective for failure analysis activities. Second, we found that, depending on the buffer size, the performance of the algorithms can vary considerably. But overall, the results indicate that simple algorithms (such as the least frequent event algorithm) have more potential to help in failure analysis tasks as indicated by the FIL values, while incurring in lower costs than the control and the optimal algorithm.

Independently of what trigger algorithm or what buffer size to select, the results put in evidence the importance of these factors in the effectiveness and efficiency of tracing. Engineers unaware of the relationship between buffer size and trigger algorithms can make poor choices regarding tracing techniques, that may negatively impact the failure analysis activities. These choices may include: (1) allocating excessive space for buffering which increases the risk of data loss in the event of a failure, (2) performing unnecessary disk transfers which increases the risk of perturbing the traced program, (3) assume that the default setup of a given trace facility automatically fits the particular tracing requirements. Engineers aware of these factors could make better choices, and hopefully perform failure analysis more

Figure 2. Scatter Plot of FIL Vs. COST



effectively.

Our results suggest several avenues for future work. First, to address questions of whether these results generalize, further studies are necessary. We plan to extend these studies with more failure scenarios, using naturally occurring faults, and analyzing additional systems. Second, we will evaluate additional tracing techniques that combine various types of events, changing buffer sizes, and different instrumentation strategies, which entails a challenging integration task. Last, we need to start transitioning from the experiments to case studies performed with software engineers to assess the accuracy and precision of our measures to evaluate the tracing techniques.

## Acknowledgements

This work was supported in part by the NSF Information Technology Research program under Awards CCR-0080898 to University of Nebraska, Lincoln.

## References

- [1] T. Ball. Efficiently counting program events with support for on-line queries. *ACM Transactions on Programming Languages and Systems*, pages 1399–1410, September 1994.
- [2] T. Ball. The concept of dynamic analysis. In *Foundation of Software Engineering*, pages 216–234, 1999.
- [3] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of MICRO-29*, 1996.
- [4] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(7):1319–1360, 1994.
- [5] J. Casmira, J. Fraser, D. Kaeli, and W. Meleis. Operating system impact on trace-driven simulation. In *Proceedings of the 31st Annual Simulation Symposium, IEEE Computer Society*, pages 76–82, April 1998.
- [6] S. Elbaum, J. Munson, and M. Harrison. CLIC: A tool for the measurement of software system dynamics. In *SETL Technical Report - TR-98-04.*, 04 1998.
- [7] M. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. *Software Testing, Verification, and Reliability*, 10(3), September 2000.
- [8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1996.
- [9] E. Iacobucci. *OS/2 Programmer's Guide*. Osborne McGraw-Hill, 1988.
- [10] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Trans. Computer Systems*, 5(2):121–150, May 1987.
- [11] D. R. Keppel, E. J. Koldinger, S. J. Eggers, and H. M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 18(1), May 1990. Performance Evaluation Review.
- [12] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software Practice and Experience*, 20(2):197–218, February 1994.
- [13] D. M. Levine, P. P. Ramsey, and R. K. Smidt. *Applied Statistics*. Prentice Hall, 2001.
- [14] H. R. Lindman. *Analysis of Variance in Experimental Design*. Springer-Verlag, 1992.
- [15] F. Meilinger. Os/2 problem determination and analysis (pda). In *A workshop about OS/2 PDA*.

- [16] J. Moe and D. A. Carr. Understanding distributed systems via execution trace data. In *The Proceedings of the Ninth International Workshop on Program Comprehension*, 2001.
- [17] D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley and Sons, New York, fourth edition, 1997.
- [18] R. H. B. Netzer. Optimal tracing and incremental reexecution for debugging long-running programs. In *Proceedings of the ACM SIGPLAN '94 conference on Programming Language design and implementation*, pages 313–325, 1994.
- [19] OCSystems. Aprobe: a tracing probe. [http://www.ocsystems.com/scrn\\_shot\\_tracing.html](http://www.ocsystems.com/scrn_shot_tracing.html).
- [20] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *International Conference on Software Engineering*, pages 277–284, 1999.
- [21] J. S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, July 1997.
- [22] S. Reiss and M. Renieris. Encoding program executions. In *International Conference on Software Engineering*, pages 221–232, 2002.
- [23] J. Rupert G. Miller. *Beyond ANOVA, Basics of Applied Statistics*. John Wiley & Sons, 1986.
- [24] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1995.
- [25] J. M. Voas. Pie: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.
- [26] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *Proc. IEEE Fault-Tolerant Computing Symp*, pages 22–31, June 1995.
- [27] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering, An Introduction*. Kluwer Academic Publishers, 2000.