Reducing and Manipulating Complex Trace Data*†

Hervé Touati
Alan Jay Smith‡

Abstract

In performance analysis of computer systems, trace-driven simulation techniques have the important advantage of credibility and accuracy. Unfortunately, traces are usually difficult to obtain, and little work has been done to provide efficient tools to help in the process of reducing and manipulating them. In this paper, we present a tool we have developed to help in the data reduction and data analysis phases of trace-driven simulation studies. Our tool has three main advantages: it accepts a variety of common trace formats; it provides a programmable user interface in which many actions can be directly specified; and it is easy to extend. This tool is particularly helpful for reducing and analyzing complex trace data, such as traces of file system or database activity. In this paper, we present our design principles, our implementation techniques, and a few concrete examples of the use of this tool.

1 Introduction

There are two ways to obtain new trace data for a trace-driven simulation study: to instrument a system or to use some existing instrumentation. The first solution requires intimate knowledge of the system, access to the source code, and a very trusting site at which to gather the data. It is sometimes possible to meet these requirements in a research environment (e.g. the work on UNIX reported in [ZDCS85]). Unfortunately we could not use this approach to gather trace data from commercial database installations to investigate buffer management policies and disk caching for large commercial database systems.

To gather trace data on IBM DB2 systems, we had to rely on existing instrumentation. We used the DB2 software monitor [IBM87]. The data format used by the DB2 software monitor is fairly complex. It is intended to be used only by DB2PM, the DB2 performance monitor [IBM86].

---

*December 8, 1989

†The material presented here is based on research supported in part by the National Science Foundation under grant MIP-8713274, by NASA under consortium agreement NCA2-128, by the State of California under the MICRO program, and by the International Business Machines Corporation, Digital Equipment Corporation, Philips Laboratories/Signetics and Apple Computer Corporation.

‡The authors are with the Computer Science Division, EECS Department, University of California, Berkeley, CA 94720, USA

Unfortunately, DB2PM did not provide enough flexibility to satisfy our needs. We had to process the raw trace data ourselves. Similar post processing of data gathered from IBM's GTF (General Trace Facility) and SMF (System Management Facility) [IBM87] was needed for the research reported in [Kur88, Smi85, ZDCS85] and was extremely time consuming and tedious.

The format of these raw traces is fairly complex: it consists of more than 100 record types and over 700 distinct record fields. Just to read the trace data and convert it to an understandable form, we needed to translate a long format description into data structures in some programming language. Doing this by hand would have been a painful and error-prone process. Instead, we wrote a parser which could read a format description file and convert it into an internal data structure. We then integrated the parser into a program to read the trace and display it in textual form. The advantage of this approach is its generality: we were able to use the same program to read the other traces we wanted to use for our study (the IMS traces reported in [KD89], and the UNIX file system traces reported in [ZDCS85]), which were all in different formats. A single program with different format descriptions was able to handle all our traces.

Trace driven simulation benefits enormously from transforming raw traces into a useful common format; the trace analysis and simulation programs thus don't have to suffer from the complexity and overhead of this task. A processed ("reduced") trace is one from which useless information has been deleted, and in which useful information has been encoded in simple ways and in which various references to the same object (e.g. job, file, open) have been tied together by pointers or unique identifiers [Zhou85]. Trace reduction involves simple tasks like extracting record fields, and more complex tasks involving reconstructing useful information scattered throughout the traces. The basic operations and data structures needed to implement these tasks are simple and few: field extraction, associative arrays, assignments, comparisons. More importantly, these operations are the very ones which are needed to perform simple statistical analysis of the trace data. Since we were likely to need several hundred such small programs in the course of our study, we felt it would be more convenient to make them independent programs rather than part of the tool. For that purpose, we implemented an interpreter for a simple command language on top of our tool. Many simple trace reduction or trace analysis tasks may be specified directly as scripts in this command language. With this added capability, it appeared that our tool would be the only program we would ever need to implement our data reduction and analysis tasks: we refer to this program as TRAMP, which stands for *Trace Reduction And Manipulation Program.*

Figure 1 shows the internal organization of our tool. At the beginning of execution, TRAMP reads and parses two files: a file describing the format of the input trace, and a file containing the command language script to be executed. It then processes the input trace, read either from a file or from the output of another program using a UNIX pipe. In particular, TRAMP can operate directly from compressed traces [Sam89], reducing the need for large permanent storage capacities. (The compressed traces are read by a decompression program whose output is then piped to TRAMP.)

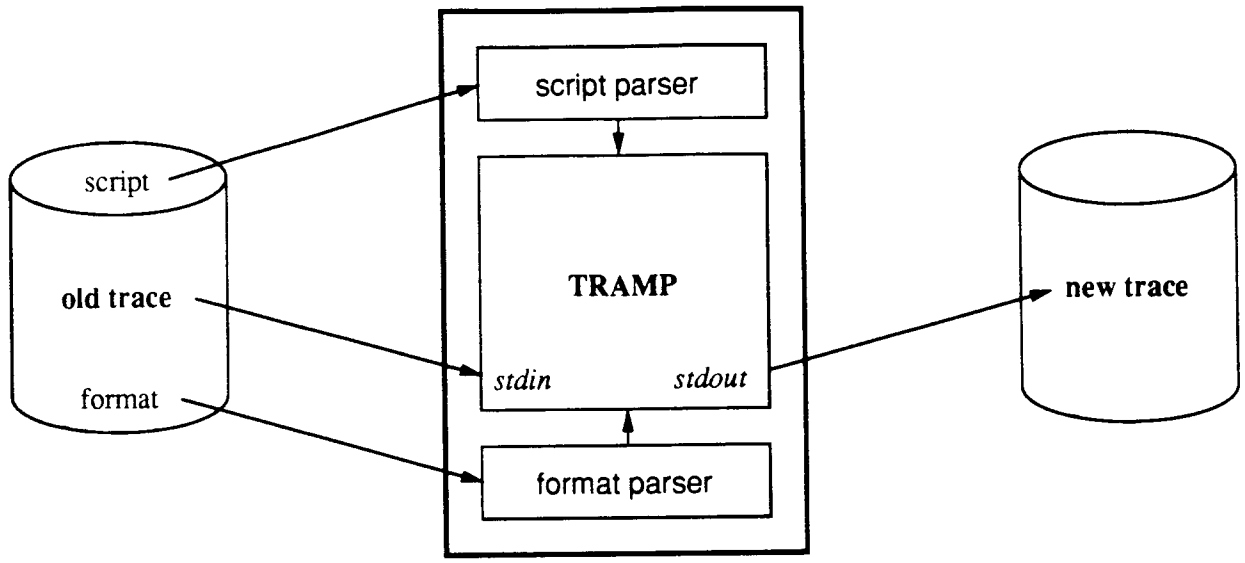

Figure 1: Changing the Format of a Trace

Figure 1 also illustrates how TRAMP can be used to reduce traces from one format (**old trace**) to another (**new trace**). In this case, the command language script specifies how the new trace is to be obtained from the old.

Figure 2 shows that TRAMP can be used in a similar fashion to perform simulations. In this case, the command script specifies the simulation that needs to be performed. To actually implement the simulation algorithm, one has the flexibility to decide which part should be included in TRAMP itself, and which part should be left in the command script. In general, it is preferable to implement basic simulation algorithms in the system and leave specific details and parameters in the command script. As an illustration, we can consider the simulation of management policies of disk caches. Once we have implemented a new cache replacement policy and made it available at the command level, we can use the flexibility provided by the command interpreter to study some of its variants. For example we can compare the performance of one large cache vs. several smaller caches to model the degradation of performance in caching file blocks closer to the disks and further from main memory as was studied in [Smi85]. We can also investigate a variety of prefetching policies directly at the command language level. This approach is simpler, easier to manage and less error-prone than the classical solution which consists of coding in the simulator itself all variants of all policies under investigation.

TRAMP currently consists of approximately 6950 lines of C++ code. Of this, 2450 lines came from the GNU C++ data structure libraries. We have used object-oriented programming techniques

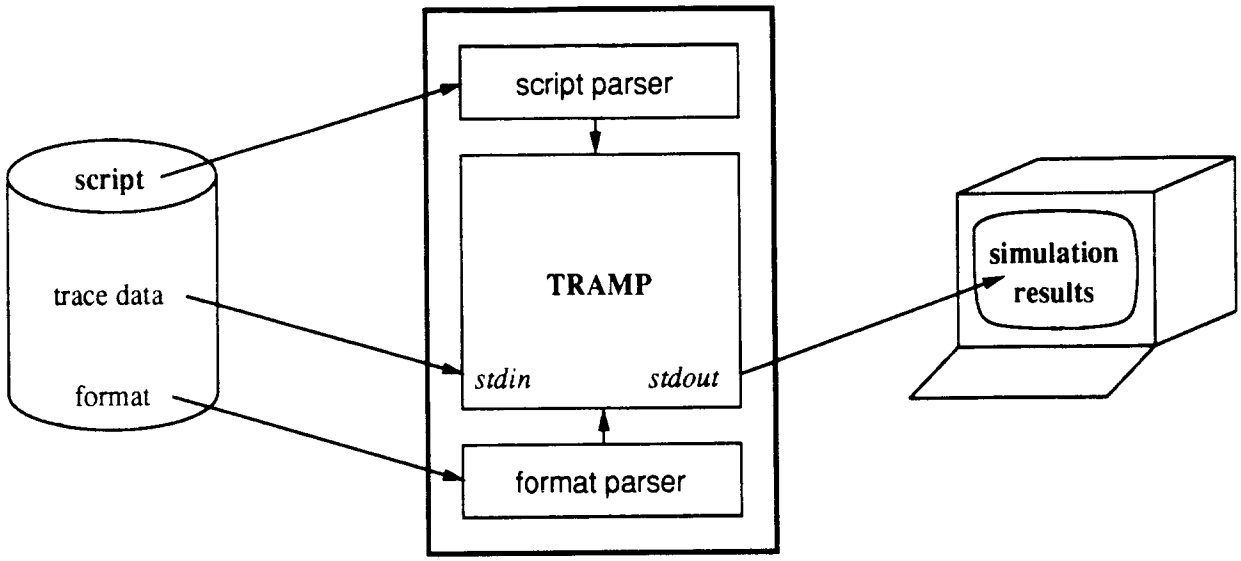

Figure 2: Trace-Driven Simulation

in several parts of our implementation. These techniques are not indispensable, but have been quite useful. We will use a moderate amount of C++ [Str86] terminology throughout this paper.

The remainder of this paper provides more details on TRAMP and compares it with related work. In section 2, we show how trace formats may be specified, and how they are handled by the system. We then describe the design and implementation of the command language. In section 4 we compare TRAMP to similar tools and present work which inspired its development, and in section 5 we present some performance measurements. Finally in section 6 we give four examples of possible use of our system. Section 7 concludes and summarizes the main points of this paper.

2 Specification and Handling of Trace Formats

2.1 An Example

We give in figure 3 an example of specification of trace format. This trace format is composed of two record types: "IO" and "XACT_END". It is a simplified version of the reduced trace format we use for traces of DB2 buffer activity; a "IO" record records a buffer request event, while a "XACT_END" record marks the end of a transaction. The format description language allows the user to specify *records*, record *fields*, and optionally a set of *values* for each field.

- a *record descriptor* starts with the keyword "**record**", followed by the name of the record, a

```
record IO {                          /* *** IO RECORD *** */
  byte       RECORD_TYPE;              /* type of record */
  bytes(4)   XACT_ID;                  /* transaction identifier */
  bytes(4)   FILE;                     /* file identifier */
  bytes(4)   PAGE;                     /* page number */
  time       TIME;                     /* time stamp */
  byte       FUNCTION {                /* read or write */
    0x00  "read",
    0x01  "write",
  };
};
record END_XACT {                    /* *** END_XACT RECORD ****/
  byte       RECORD_TYPE;              /* type of record */
  bytes(4)   XACT_ID;                  /* transaction id */
  time_incr  TIME;                     /* time stamp */
};
```

Figure 3: An Example of Format Specification

list of field descriptors delimited by curled brackets and a semicolumn.

- a *field descriptor* consists of a type specifier, followed by a field name, an optional value list and a semicolumn. A value list is a list of value descriptors delimited by curled braces.
- a *type specifier* consists of a type name, followed by an optional parameter. We will discuss type specifiers in more detail in section 2.2.
- a *value descriptor* consists of a value, a string and a comma. The string is really a comment; it is only used by data output routines.

Format descriptions are syntactically similar to C structures. We have added value descriptors (which borrow the syntax of C structure initializers) and parameters to field types; but we do not support union types nor nested record definitions.

2.2 Field Types

Field types are internally represented by *type descriptors*. Type descriptors are implemented as C structures containing the size of the type they represent and pointers to functions or procedures which specify how fields of that type are to be manipulated. These operations are:

- `get_size(type_descriptor)`: returns the size of the field. For types parameterized by size, this routine is expected to return the size of the particular type instance. For example, the size of a field of type "**bytes**(i)" is i.

- `get_value(type_descriptor, field_pointer)`: this routine returns the value of the field. If the field value does not fit into a 4 byte word, a copy of the value is made and a pointer to the copy is returned. Currently, this is only done for character strings.

- `compare(type_descriptor, value, field_pointer)`: this routine determines whether the field has a given value.

- `print(type_descriptor, field_pointer)`: this routines prints the contents of the field, passed as an argument, in human readable format.

Field types are implemented as an independent module (a C library). All the details of the implementation of field types are hidden inside this module. The field type module exports to the rest of the system only the definition of a type descriptor, the names and types of the generic operations that can be performed on a type descriptor, and the routine: `create_type(type_name, optional_param)`. This routine is used by the rest of the system to create a new type descriptor from the name of the type and an optional parameter.

We implemented type descriptors as C++ classes. The advantage of using C++ or any language with similar facilities is that the initialization routines for type descriptors are generated automatically by the compiler, thus reducing the size of the program and the chances of clerical errors. It also makes the system easier to extend. To add a new field type all one has to do is to implement the four routines listed above, register the name of the new field type in a table, and recompile the library. This modularity makes it simple to add new field types to the system. We give below two examples illustrating the flexibility of our design.

Printing Time Stamps As shown in figure 3, we use the special type "**time**" to specify time stamp fields. Though the field type "**bytes**(4)" could have been used for time stamps, time stamps would have been difficult to read. By using "**time**", we can introduce a specialized `print` routine which knows about the particular unit of time we use in our traces, and can print time stamps in standard time units (hours, minutes, seconds, milliseconds).

Dealing with Byte Ordering Most current computing environments are organized as large *heterogeneous* distributed systems. Due to the rapid evolution of available hardware, it is relatively common for any group of users, especially in the research community, to have machines of different byte ordering sharing access to the same binary trace data files. Because of byte ordering, a correct TRAMP script on one machine may not work properly on another. There are several

possible solutions to this problem, but we think that the simplest and most elegant one is to use the flexibility of the field type system.

The main idea is to consider field types not as a description of the contents of the tape, but as a specification of the operations a given machine should execute to manipulate field values correctly. On a machine with a different byte ordering than the trace, we simply use a different format description file, using the new byte swapping field types "**swap_halfword**" and "**swap_fullword**" to specify 2 and 4 byte quantities. Swapping only occurs when the fields are accessed, which limits the overhead. With this approach, there is no need to keep several copies of the traces with different byte orderings, nor to modify the source program, nor to modify the simulation scripts. Moreover, since TRAMP reads a UNIX environment variable to determine which format file to use, the correct format file may be specified automatically at login time by checking the type of machine being used.

2.3 Limitations and Restrictions

TRAMP does not accept arbitrary format specifications. One implicit assumption we make about records is that fields are represented in the trace as a sequence of consecutive bytes. Moreover, in the current implementation, fields are restricted to be of fixed length. Supporting variable length records is straightforward. All we need to do is to modify the type routine `get_size` so that it accepts a `field_pointer` as the second argument, and the input buffering routines, so that they compute the length of records on the fly rather than reading them from a precomputed table of record lengths. A more sophisticated approach would be to remember which records are not of variable length, and look up their fixed lengths in a table instead of recomputing them. Variable sized records were implemented in a previous version of our tool and were used to reduce raw trace data into a simpler format. During reduction, character strings were replaced by unique identifiers; the original strings have been kept in a separate file. As a result, our reduced traces only have fixed length records, and we have not maintained the code which supported variable length records since then.

A second assumption we made about record formats is that the first field of each record is a single byte field identifying the type of the record. Moreover the value of this byte is expected to be $n - 1$ for the n^{th} record described in the format specification file, that is 0 for the first record, 1 for the second record, and so on. If the original trace has no such record headers, they would have to be added before TRAMP can be used. If the original trace uses byte headers which are not in sequential order, but are small numbers, one can sort the format description file and add dummy records to fill the gaps. A more elegant solution would be to let the user specify explicitly the values of the record headers in the format description file. This extension is straightforward to implement.

TRAMP does not provide support for tightly encoded headers. These headers can be useful to generate very compact traces [Sam89]. Unfortunately they can rely on arbitrarily complex encodings and there is no easy and general way to support them.

3 Command Language Interpreter

The command language interpreter allows the user to specify tasks at a high level, with direct access to the internal data structures of TRAMP, without having to understand or modify the tool itself. The command language is not a general purpose programming language. It is intended to be used in the specific context of record at a time processing of binary trace data. A *program* or *script* written in the command language is composed of a "main" routine, which is executed for each record encountered in the trace. The user can also specify a preprocessing and a postprocessing routine. A simple mechanism to support user-defined routines with no arguments is also provided. To make things more concrete, we present in detail a simple example in the next subsection. In the following subsection, we describe the implementation of the command interpreter.

3.1 An Example

We give below an example of how a simple data analysis task can be directly implemented in the command language. This task consists of computing the number of I/O reads and I/O writes recorded in a trace. The format of the trace is the one given in figure 3. To perform this task, we need to be able to: fetch a record, determine the type of a record, extract the contents of a field, define and manipulate variables, and execute some action before and after the entire tape has been processed. We show below how these actions can be specified in the command language. The script to implement this task is given in figure 4. For clarity of presentation, we use an Algol-like pseudo-code, though the actual syntax of our command language is derived from Lisp. The actual script we used for this example is given in figure 5. The following list presents the constructs we use in the script.

- *fetch the next record:* this action does not need to be specified explicitly in the command language. The main procedure is automatically executed for each record in the trace. The trace is processed sequentially one record at a time.

- *preprocess and postprocess:* the actions to be executed before the trace is processed can be specified in a block of statements preceded by the keyword "**preprocess**". Similarly, there is a keyword "**postprocess**" to mark the block of statements that is to be executed after the trace is processed.

```
preprocess begin
   set read_count = 0;
   set write_count = 0;
end

procedure main begin
   if (RECORD_TYPE == IO) then
      if (IO_FUNCTION == 0x00) then
         set read_count = read_count + 1;
      else
         set write_count = write_count + 1;
      endif
   endif
end

postprocess begin
   echo read_count, write_count;
end
```

Figure 4: Computing the Number of I/O Reads and I/O Writes (pseudo-code)

- *get the type of the current record:* there is a special predefined variable called **RECORD_TYPE** whose contents can be accessed to get the type of the current record.
- *get the contents of a field:* for each field of each record, there is a predefined variable which can be read to get the contents of the field. The name of this variable is obtained by concatenating the name of the record with the name of the field. For example, the variable "IO_TIME" can be used to access the "TIME" field of a record of type "IO".
- *variables:* any syntactically legal identifier can name a variable. Variables do not need to be declared, and are global to the script. They can be assigned a value using the command **set**. For example, "**set** count = 0" initializes the variable "count" to 0, and "**set** count = count + 1" increments the variable "count" by 1.

3.2 Internal Organization of the Command Interpreter

The implementation of the command language interpreter is simple and easy to extend. Two factors have contributed to this simplicity: the use of Lisp syntax for the command language and the use of a hierarchy of *parse nodes* to build a tree representation of the command script. To execute the command script, we directly interpret this tree representation.

```
(defun begin
    (and (setq read_count 0)
         (setq write_count 0)))

(defun main
    (or (and (eq (RECORD_TYPE) IO)
             (or (and (eq (IO_FUNCTION) 0x00)
                      (setq read_count (add read_count 1)))
                      (setq write_count (add write_count 1))))))

(defun end
    (and (echo read_count)
         (echo write_count)
         (nl)))
```

Figure 5: Computing the Number of I/O Reads and I/O Writes (actual script)

Parse nodes are organized similarly to field type descriptors. They are implemented as C structures containing information about the degree of the node, and pointers to child nodes in the tree. If the node is a leaf of the tree, additional information is also associated with the node; for example, if a leaf node represents a variable, the value of the variable is stored in the node. In addition, nodes contain pointers to two functions which implement the following node-dependent operations:

- `execute(record_pointer)`: this routine implements the function associated with the node. It can invoke the function associated to the children nodes recursively. This simple mechanism is sufficient to implement conditional statements, loops, and other basic programming language constructs. A pointer to the current record is passed down the tree as an argument. During preprocessing and postprocessing, this pointer is set to 0.

- `create()`: creates a copy of a node. This function is used only to build the parse tree.

Each construct in the command language corresponds to one type of parse node. For example, there is one type of node for assignment statements, one type of node for conditional statements, one type of node for arithmetic expressions, and one type of node for variables. Each node type has a name. Each name is recorded in a symbol table during the initialization of the program associated with an instance of the node type. Once the symbol table is initialized, the parser can build a parse tree simply by looking up each token in the symbol table and finding which parse node should be generated. Some tokens, like parentheses or variables, are treated specially. Due to the simplicity of Lisp syntax, however, those special cases are few. Adding functionality to the

command language is straightforward with this syntax. One needs only define a new parse node, implement the required operations for it, and register it into the symbol table.

We use C++ classes to implement parse node types. A similar, but less elegant, implementation would be possible in C or any other language with records and function pointers.

4 Related Work

4.1 Sources of Inspiration

AWK [AKW88] has been a major source of inspiration for the design of the command language. A more detailed comparison between AWK and TRAMP is given below. The merits of building software systems in several layers, each in a different programming language, from fast but rigid languages to languages that are slower but easier to modify, is discussed and illustrated by R. Stallman in [Sta84]. C++ [Str86] has been an indirect source of inspiration by providing a good platform for experimentation. Finally, our use of abstract data types to represent field types bears some similarity with their use in relational databases (e.g. see Stonebraker [Sto86]), though our design was developed independently.

4.2 Discrete Event Simulation Tools

We have not found any reference to any tool with capabilities comparable to ours specifically targeted at processing trace data. Work on tools for computer system simulation has concentrated on specialized programming languages, libraries of routines or other programming tools for discrete event simulation, where events are generated internally by the simulator rather than read from a trace. There is an abundant literature in this area and several commercial products are on the market. We refer the interested reader to MacDougall [Mac87] for an overview of this field; [Dah68, Pra75] for a survey of discrete-event simulation languages; and Schwetman [Sch85] for more recent work.

TRAMP is not directly comparable with these tools. Its command interpreter provides support for easy manipulation of trace data and relatively simple trace-driven simulations. Complex simulations have to be coded in the implementation language: the command interpreter is too primitive for such applications. We use C++ as an implementation language. Like Simula [DN66], from which it derives some of its constructs and terminology, C++ provides extended type facilities which make writing simulation programs easier than in other languages. We use a publicly available implementation of C++, developed by the Free Software Foundation, for which libraries of data structures are available. These libraries provide many of the data structures needed in discrete event simulation (priority queues, lists, associative arrays). We have found the use of C++

in conjunction with this general-purpose library convenient enough to limit the need for a more specialized simulation language.

4.3 DB2PM

There is an IBM program product called DB2PM (DB2 Performance Monitor) which can analyze trace data obtained with the DB2 built-in trace facility [IBM86]. DB2PM has several limitations that we tried to overcome with TRAMP: it only accepts trace data in the format generated by the DB2 trace facility, can only be used for a suite of predefined operations, and is not extensible by the user. DB2PM is targeted at IBM customers who want to monitor the activity of their DB2 installations, and is very convenient for that function. It is far from ideal, however, for performing more advanced analyses. It can generate an extensive set of reports, but these reports are all derived from simple statistical analysis of the trace data.

4.4 AWK

AWK is a data formatting and processing language developed by Aho, Weinberger and Kernighan [AKW88]; AWK has been an important source of inspiration in the design of TRAMP, and the systems are similar in several aspects. They both are targeted to sequential processing of formatted data. They both can be programmed in a simple interpreted programming language and provide the same basic programming constructs: variables, conditionals, iterations, associative arrays, user-defined functions, and printing and formatting routines.

However there are several important differences between AWK and TRAMP due to their different intended uses. AWK is targeted at manipulating relatively small text files; TRAMP is targeted at manipulating large trace data files in binary format. TRAMP is faster (we provide some evidence of that in the next section), and provides facilities to generate and manipulate data in binary format. AWK provides a set of powerful functions to manipulate strings of characters, and its user-defined functions can take arguments.

For simple data processing tasks, we found both systems equally easy to use within their respective domain. For complex tasks, we found both systems equally cumbersome to use. Complex tasks are in fact easier to implement in C++, because of its strong typing, its support for data structuring and modularity, and its libraries of predefined data structures. The fact that TRAMP is easy to extend directly in its implementation language is a definite advantage when complex algorithms need to be implemented.

COMMAND		*user time*	*system time*
ORIGIN	FUNCTION	*(seconds)*	*(seconds)*
UNIX	`cat`	0.01	2.52
C PROGRAM	`record read`	5.32	2.48
TRAMP	`record read`	12.18	2.85
TRAMP	`simulation`	248.99	5.14
TRAMP	`skeleton`	126.02	2.87

Table 1: Simple Performance Comparison

`cat:`	UNIX facility to read a file
`record read:`	file read one record at a time with two function calls per record
`record read:`	simplest TRAMP script executed on each record
`simulation:`	LRU cache miss ratio simulation
`skeleton:`	same simulation but with only script and IO overhead

5 Performance

There is often a price to pay for flexibility, and our tool is no exception. The principal source of overhead comes from the fact that command language scripts are interpreted. This overhead could be reduced significantly by relatively simple techniques like byte-code emulation. In practice, however, we have not found the need for this additional complexity.

We ran several experiments to evaluate the performance of our tool. All experiments were run on a VAX 8600 running 4.3 BSD UNIX and repeated 10 times. We only report here average results, since variations between runs were small.

5.1 Command Language Overhead

The experiments reported in table 1 were run using the same file (hereafter referred to as `trace`). The first benchmark we run simply read the file `trace`. We compared the speed of our tool with a UNIX facility, `cat`, and a simple C program. The read overhead reported for our tool can be explained as follows. For each record it processes, our tool calls two functions. The first function executes the command script on the record; the second function determines the size of the next record, makes sure it fits entirely in the buffer and reads more data into the buffer if necessary. In this example, the function executing the command script returns immediately. To provide an element of comparison, the C program was made to perform two function calls which returned immediately every record encountered in the input file `trace`.

To determine the overhead caused by the interpreted execution of a more complex task, we used a more realistic example: the computation of cache miss ratios for a simple replacement

SIMULATION RUN		INPUT FILE		*user time*	*system time*
PROGRAM	FUNCTION	TYPE	SIZE (BYTES)	*(seconds)*	*(seconds)*
AWK	LRU	`ascii`	1,692,789	253.09	2.75
TRAMP	LRU	`binary`	1,000,000	28.51	1.29

Table 2: Comparison between AWK with TRAMP

policy (LRU). The replacement algorithm is implemented in TRAMP itself; the role of the script is to extract the relevant information from the trace and to call TRAMP simulation routines. We measured the execution time to execute the script with and without calling the simulation routines. The results are also reported in table 1. The `skeleton` script refers to a script which performs all the actions of the simulation script except that it does not call TRAMP simulation routines. If A is the total amount of cpu time to perform the simulation, B the total amount of cpu time to execute the `skeleton` script, and C the total amount of cpu time to have TRAMP simply read the file, the overhead in percentage due to the interpreted execution of the script is: $100*(B-C)/A = 44.8\%$. This value was obtained from the data in table 1 by adding user time and system time. This overhead is relatively high but was acceptable for our application. As mentioned earlier, the overhead could be reduced by more efficient implementation techniques; it is not an irremediable consequence of our design.

5.2 Comparison with AWK

We ran a simple experiment to compare the use of our tool with the use of AWK to manipulate trace data. The experiment consisted in performing an LRU simulation on a 50,000 record file. TRAMP used the original binary version of the file, which was 1 million byte long. We then translated the file into `ascii` representation so that it could be processed by AWK. The increase in size to convert binary to `ascii` was 69%, and the increase in simulation time was a factor of 8.6. We used in the simulation the latest version of AWK available from AT&T, which is significantly faster (between 2 to 3 times) than the most widely available original version. The results are reported in table 2.

6 Examples

In this section, we present four examples of the use of our tool. Though these examples have been created for the sake of presentation, they are representative of problems that one encounters in practice. To make things more concrete, we suppose that we want to investigate file system buffer management algorithms for an operating system, and that we are in possession of a trace

of file system activity that has already been reduced to the format shown in figure 3. We show how to use our tool to perform the following tasks: replace process identifiers by unique identifiers (section 6.1); compute the number of references per file (section 6.2); compute the file cache miss ratio for a simple replacement policy (section 6.3); and finally generate an artificial trace with predefined characteristics to check the correctness of the previous example (section 6.4).

These examples only represent a small subset of the possible applications of our tool. We hope however that they will be sufficient to allow the reader to appreciate the capability of our system.

6.1 Replacing Process Identifiers by Unique Identifiers

Operating systems may reuse the process identifiers of terminated processes within the period of observation of a trace. To facilitate the analysis of the trace data, it is useful to replace these process identifiers by identifiers guaranteed to be unique throughout the trace. The algorithm we use to implement this simple task works as follows: we start with an empty identifier table; each time we encounter a record containing a process identifier, we look up in the table to see if there is a unique identifier associated with it. If there is one, we use it. If there is none, we create a new identifier, and store it in the table. We then use this unique identifier to update the trace record. If the trace record was signaling a process termination, we remove the table entry for the corresponding process identifier. The script we use to implement this algorithm follows very closely this high level description. It is shown in figure 6.

This script makes use of several constructs of the command language that we have not encountered so far:

- *associative arrays:* the command language supports one and two dimensional associative arrays. The script uses the following three array operations:
  - "**set** array[key] = value", which associates a value to a key.
  - "key **found in** array", which returns true or false according to whether or not there is an entry for the key in the array.
  - "**delete** key **from** array", which removes a key from an array.
- *user defined procedures:* user-defined procedures are allowed but are restricted in the sense that they cannot take arguments. Parameters can be passed implicitly through global variables.
- *updating records:* it is possible to modify the current record by using the "**update** FIELD_NAME = value" command. The modified record can then be output in binary format with the "**spit**" command.

```
preprocess begin
   set next_id = 0;
end

procedure make_new_id begin
   set id_table[pid] = next_id;
   set next_id = next_id + 1;
end

procedure main begin
   if (RECORD_TYPE == IO) then
      set pid = IO_XACT_ID;
      if (pid not found in id_table) then
         make_new_id();
      endif
      update IO_PID = id_table[pid];
      spit();
   else if (RECORD_TYPE == END_XACT) then
      set pid = END_XACT_XACT_PID;
      if (pid found in id_table) then
         update END_XACT_XACT_PID = id_table[pid];
         delete pid from id_table;
         spit();
      endif
   endif
end
```

Figure 6: Making Process Identifiers Unique

Arrays grow automatically on demand. The space occupied by deleted entries is recovered automatically. Holes are compacted at regular intervals to improve locality of reference.

6.2 Number of References per File

Before starting to design a disk caching or file placement algorithm, it is important to characterize the workload reported in a trace. One statistical distribution of particular interest is the number of read and write references per file. Using TRAMP, it is easy to gather this information from a trace. We just need to create two tables: one for read and one for write references. Each time a file is accessed, the corresponding table entry is incremented by one. After the trace is processed, the contents of the tables are output. It is then often convenient to use or AWK to further reduce

```
procedure main begin
  if (RECORD_TYPE == IO) then
    set file = IO_FILE;
    if (IO_FUNCTION == 0x00) then
      set read_counter[file] = read_counter[file] + 1;
    else
      set write_counter[file] = write_counter[file] + 1;
    endif
  endif
end

postprocess begin
  for (file in write_counter) begin
    if (file not found in read_counter) then
      set read_counter[file] = 0;
    endif
  end
  for (file in read_counter) begin
    echo file, read_counter[file], write_counter[file];
  end
end
```

Figure 7: Counting File References

or format the output.

The script we use to execute this task is given in figure 7. It makes use of one construct we have not encountered so far:

- "**for** (key **in** array)": this is a loop statement which executes the statement that follows it once for each key found in the array.

In the postprocessing routine in the script, it is necessary to add to the "read_counter" array all the keys in the "write_counter" array which are not already present. Otherwise, references to files which are only written within the period of observation of the trace would be lost.

6.3 LRU Miss Ratios

LRU miss ratios are often used in cache analysis studies as a basic point of reference for assessing the efficiency of more complex replacement policies. To compute LRU miss ratios, we do not implement the replacement policy directly in the command language. Though it would be possible

```
preprocess begin
   set buf = make_buffer(LRU, 2048);
end

procedure main begin
   if (RECORD_TYPE == IO) then
      set file = IO_FILE;
      set page = IO_PAGE;
      if (IO_FUNCTION == 0x00) then
         read_buffer(buf, file, page);
      else if (IO_FUNCTION == 0x01) then
         write_buffer(buf, file, page);
   endif
end

postprocess begin
   print_buffer(buf);
end
```

Figure 8: LRU Simulation Script

to do so, it is inefficient and cumbersome. We follow instead a mixed strategy: we code the basic routines needed to implement the replacement policy directly in C++ and make them available at the command language level. We can then write a script to compute the desired miss ratios by calling these routines directly. We show a version of this script in figure 8.

Another advantage of this approach, in addition to providing a good compromise between speed and flexibility, is the fact that it can easily be extended to other replacement policies. For each replacement policy, we just need to implement the following routines (we use one more routine than the ones shown here to study prefetching policies):

- "**make_buffer**(policy, size)": this function returns a pointer to a buffer of the specified size managed by the specified policy. The "policy" argument has to be known to the system.
- "**read_buffer**(buffer, file, page)": when a read record is encountered in the trace, this procedure calls the simulation routine for the policy managing the buffer. The buffer entries are updated by the replacement policy, which also keeps track of buffer hits and misses.
- "**write_buffer**(buffer, file, page)": this procedure is similar to the previous one, but for a write record.

```
preprocess begin
   set io = 0;
   set p0 = 0;          set file0 = 0;
   set p1 = 1;          set file1 = 1;
end

procedure spit_100 begin
   set counter = 0
   while (counter < 100) begin
      spit_char io;
      spit_int p0;                        spit_int file0;
      spit_int counter;                   spit_char 0x00;
      spit_char io;
      spit_int p1;                        spit_int file1;
      spit_int counter modulo 50;         spit_char 0x00;
      set counter = counter + 1;
   end
end

postprocess begin
   spit_100(); spit_100(); spit_100();
end
```

Figure 9: Generating an Artificial Trace

- "**print_buffer**(buffer)": this procedure reports the results of the simulation. It can also be used during the simulation to follow the evolution of the number of buffer misses over time.

6.4 Generating an Artificial Trace

Results obtained from trace-driven simulations do not need to be validated, since they are derived from data obtained by direct experimentation. However they need to be checked. We use a simple testing technique which can be helpful in detecting errors in the implementation of replacement policies. This technique consists of generating artificial traces with well-defined patterns of reference and use them to drive the simulation routines. If the artificial traces are suitable, it is possible to compute analytically the correct miss ratios for these traces and thus to check the results obtained by simulation. We give in figure 9 an example of a simple script which generates a trace which can be used to check the correctness of the implementation of our LRU replacement policy. The trace is composed of the references of two interleaved processes, which have cyclic reference

patterns of different periods. The script uses the following two routines which have not yet been introduced:

- "**spit_int** value": this procedure writes its argument to standard output in binary format. There is also a procedure called "**swap**" which can be called in the case the bytes of the output need to be swapped.
- "**spit_char** value": this procedure writes its argument to standard output in binary format. Only the least significant byte of the argument is considered.

7 Conclusion

We have opened our simulator at both ends: at the low end, by making it independent of trace formats; at the high end, by making it directly programmable by the user. As a result, we were able to enhance the capacity of our simulator far beyond what is usually available in similar tools while reducing the overall development effort. Our simulator is able to perform relatively complex and unrelated tasks, such as trace reduction, trace compression, computation of distributions of various trace dependent parameters, by being programmed directly from its command interpreter.

Because it is format independent, our simulator can be gradually expanded with more sophisticated simulation algorithms with no fear that these algorithms will have to be implemented again. We plan to make our tool publicly available after we complete our investigation. Requests by electronic mail should be addressed to `touati@arpa.Berkeley.EDU`.

8 Acknowledgements

We would like to thank Fred Douglis, Bruce Holmer, Rafael Saavedra-Barrera, Dain Samples, Margo Seltzer, Barbara Tockey, and Peter Van Roy for their helpful comments on earlier drafts which substantially improved the quality of this paper.

References

[AKW88] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[Dah68] O. Dahl. *Programming Languages*, chapter "Discrete Event Simulation Languages", pages 349–395. Academic Press, New York, 1968.

[DN66] O. Dahl and K. Nygaard. "Simula: an Algol-based Simulation Language". *Communications of the ACM*, 9(9):671–678, 1966.

[IBM86] IBM. *Database 2 Performance Monitor: Report Reference*, Order Number SH20-6858-00 edition, July 1986.

[IBM87] IBM. *Database 2 System Planning and Administration Guide*, Order Number SC26-4085-3 edition, May 1987.

[KD89] J. P. Kearns and S. Defazio. "Diversity in Database Reference Behavior". In *International Conference on Measurement and Modeling of Computer Systems*. ACM, May 1989.

[Kur88] O. Kure. *Optimization of File Migration in Distributed Systems*. PhD thesis, UC Berkeley, University of California, Berkeley, CA 94720, 1988.

[Mac87] M. H. MacDougall. *Simulating Computer Systems: Techniques and Tools*. Computer Systems Series. The MIT Press, 1987.

[Pra75] T. W. Pratt. *Programming Languages: Design and Implementation*. Prentice-Hall, 1975.

[Sam89] A. D. Samples. "Mache: No-Loss Trace Compaction". In *ACM Sigmetrics*, pages 89–97. Performance Evaluation Review Vol.17 #1, May 1989.

[Sch85] H. Schwetman. "CSIM: A C-Based, Process-Oriented Simulation Language". Technical Report Technical Report PP 080-85, MCC, 1985.

[Smi85] A. J. Smith. "Disk Cache – Miss Ratio Analysis and Design Considerations". *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[Sta84] R. M. Stallman. *"Interactive Programming Environments"*, chapter 15, pages 300–325. McGraw-Hill, 1984.

[Sto86] M. Stonebraker. "Inclusion of New Types in Relational Data Base Systems". In *Proceedings of the International Conference on Data Engineering*, 1986.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[ZDCS85] S. Zhou, H. Da Costa, and A. J. Smith. "A File System Tracing Package for Berkeley UNIX". Technical Report 85-235, University of California, Berkeley, Computer Science Division, Berkeley, CA 94720, May 1985.