SynRGen: An Extensible File Reference Generator

Maria R. Ebling M. Satyanarayanan

February 1994

CMU-CS-94-119

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in
Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Nashville, TN, May 1994

This work has been supported by the Advanced Research Projects Agency (Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio, 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 and Hanscom AFB, Contract F19628-93-C-0193, ARPA Order No. A700), the National Science Foundation (Grant No. ECD 8907068), the IBM Corporation, Digital Equipment Corporation, and Xerox Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, NSF, IBM, DEC, Xerox, or the U.S. Government.

Keywords: workload generation, modeling, debugging, testing, benchmarks, performance evaluation, tracing, extensibility, portability, Coda, file systems

Abstract

SynRGen, a synthetic file reference generator operating at the system call level, is capable of modeling a wide variety of usage environments. It achieves realism through trace-inspired *micromodels* and flexibility by combining these micromodels stochastically. A micromodel is a parameterized piece of code that captures the distinctive signature of an application. We have used SynRGen extensively for stress testing the Coda File System. We have also performed a controlled experiment that demonstrates SynRGen's ability to closely emulate real users – within 20% of many key system variables. In this paper we present the rationale, detailed design, and evaluation of SynRGen, and mention its applicability to broader uses such as performance evaluation.

1 Introduction

Transforming a file system from an initial prototype into a fully deployed system is a process fraught with hazard. Many insidious bugs will only be triggered under heavy loads and extended usage. But fear of serious failures, involving loss of data and lengthy downtime, deters many potential users. How, then, can implementors hope to increase the robustness of their system?

SynRGen is our solution to this problem. Configuration files, describing the behavior of real users and the characteristics of their data, are used by SynRGen to construct programs called *synthetic users*. When executed, a synthetic user generates references emulating the modeled users. By stress testing with a wide variety of synthetic users, an experimental system can be brought to an acceptable level of robustness.

SynRGen's usefulness extends well beyond stress testing. Since synthetic users can be parameterized and since the generated workload is reproducible, they can be used as the basis of a family of benchmarks for performance evaluation. Further, the logistical and privacy problems inherent in exporting reference traces can be avoided by exporting a synthetic user representative of those traces. Most importantly, SynRGen allows a system to be subjected to hypothetical or anticipated usage scenarios. For example, one may wish to study the behavior of a file system when the extent of write-sharing, the degree of locality or the distribution of file sizes differs substantially from that of any currently available environment.

We have used SynRGen extensively in the development of the Coda File System[16]. Our experience confirms that it is an invaluable tool for file system development. We have also shown that the synthetic users produced by SynRGen can closely emulate the impact of real users on system resources. Specifically, our experiments indicate that SynRGen can emulate a group of users in an edit/debug cycle within 20% of key system variables such as network load and server CPU usage.

Our description of SynRGen begins with a discussion of the primary factors that influenced its design. We then describe its architecture and implementation. The results of a controlled experiment exploring SynRGen's ability to emulate real users are presented in Section 4. We conclude the paper with a discussion of potential refinements and a survey of related work.

2 Design Rationale

2.1 Combining Realism with Flexibility

The dominant design consideration for SynRGen was our need to characterize a wide variety of usage environments, including aspects such as the physical characteristics of files, the behavior of users and programs, and the scale of the system. In building SynRGen, we had to carefully balance the degree of realism achieved with the amount of flexibility possible.

Realism can be viewed as the measure of correspondence along a number of dimensions[6]. The dimensions of interest may vary considerably. For example, in one experiment the only variable of interest may be server CPU utilization. In contrast, a more detailed experiment might include many system variables such as cache hit ratio at clients, volume of client-server traffic, and disk traffic at the server. The ultimate degree of realism is to replay an actual file reference trace. Unfortunately, traces can be extrapolated only in limited ways. Parameters such as file

Sources for SynRGen are available via AFS (/afs/cs.cmu.edu/project/coda-synrgen) or via anonymous FTP from ftp.cs.cmu.edu (project/synrgen). For questions or comments related to SynRGen, please send e-mail to synrgen@cs.cmu.edu.

sizes and interarrival times can be scaled with relative ease. But there is no mechanical way to modify more complex aspects such as the degree of write-sharing between users.

SynRGen achieves realism through trace-inspired *micromodels*. File reference traces of applications reveal distinctive patterns or *signatures* in their file access behavior. A micromodel captures the signature of a particular application in a parameterized function. As an example, consider a hypothetical C compiler that opens and reads a ".c" file, opens and reads a number of ".h" files, creates an empty ".o" file, writes to that file, and finally closes it. The specific ".c" file referenced, the number and identity of ".h" files, the sizes of each file, and many other details vary from execution to execution. Yet an examination of traces from many such executions will reveal the general pattern described above. A micromodel for this hypothetical C compiler would capture this distinctive signature, parameterizing the details of interest that vary between executions.

SynRGen achieves flexibility because experimenters can stochastically combine micromodels to capture workloads representative of a particular class of users. They can also specify parameter values at runtime and during configuration. Rather than wiring in the degree of realism, our approach defers this decision to the experimenter.

We expect most experimenters will begin by using existing micromodels, simply setting parameter values appropriately. If they find that no micromodel exists for an important application or if they find that the existing micromodel is not sufficiently accurate, they will either create a new micromodel or improve the existing one. An important aspect of our approach is that it is possible to substantially separate the efforts of the modeler and experimenter – micromodels encapsulate the work of the former.

2.2 File System Independence

A goal in developing SynRGen was to compare the performance of alternative implementations of a particular file system API (application program interface) for identical user communities. For example, one might want to compare AFS[14], NFS[12], Sprite[11] and Coda[16]. This requires that the reference stream generated by SynRGen had to be at the level of abstraction common to these file systems, in this case the Unix file system API.

References generated at the file system API means that semantic constraints at that level must be respected. For example, in the Unix API, one cannot delete a directory unless it is empty; nor can one read from a file until it has been opened. Rather than trying to capture these API-specific constraints in some declarative form (such as a table), we chose to embed them in micromodels written in arbitrary C code. It is then the responsibility of the micromodel's author to ensure that API-specific constraints are met. Further, the mechanism for stochastically combining micromodels is API-independent because the micromodels encapsulate all knowledge of the API.

A consequence of this decision is that SynRGen is not restricted to generating file system references. By writing appropriate micromodels, SynRGen could equally well generate, for example, synthetic SQL database queries or disk I/O references. Although our experience with SynRGen has been limited to file reference generation, we do not foresee any obstacles to its broader use.

2.3 Parameterizing File Locality

The performance of virtually every file system is critically dependent upon design assumptions regarding the degree and nature of locality of file reference. If these assumptions do not hold in a particular environment, the performance of the file system could be significantly different from expectations. We wanted the ability to study the effect of changing the locality of reference substantially.

In order to conduct such a study without recoding every micromodel, one needs the ability to convey interfile locality information between independently-authored micromodels. For example, if a user examines the attributes of a given file, he or she is likely to look at the contents of that file next. Somehow, we must capture this temporal locality of reference. We use *pathname iterators* to meet this requirement. A pathname iterator is simply a procedure that encapsulates locality information. Each call to a pathname iterator yields the name of a file or directory; the stream of names generated by successive calls exhibits the interfile locality modeled by that iterator. To use SynRGen micromodels with different locality patterns, one merely invokes them with different pathname iterators.

In principle, a similar mechanism could provide a choice of intrafile locality models. However, intrafile locality tends to be specific to an application rather than being a function of the usage environment. Hence we expect SynRGen's micromodels to capture intrafile locality internally. For instance, a micromodel for the `more` program would open a file and read all or the initial part of the file sequentially, while a micromodel for a database or a linker/loader would open a file and access portions of the file randomly.

3 Architecture and Implementation

The design rationale presented in the previous section leads directly to the architecture of SynRGen. The core of SynRGen consists of a set of preprocessors that transform configuration files into executable code, linking in the specified micromodels from a library. Synthetic user executables are generated for each type of user specified by the configuration files. Running a synthetic user results in references corresponding to that type of user.

An experiment consists of subjecting a candidate system to a collection of synthetic users. To emulate a timesharing environment, multiple synthetic users are run on the same machine. When emulating a distributed workstation environment, each synthetic user is run on a different client machine. The system-specific instrumentation necessary for monitoring the impact of synthetic users on clients, servers, and the network must be provided externally.

In the following sections, we describe SynRGen at the next level of detail. We first present the abstractions supported by SynRGen, and then discuss how each abstraction is specified. We complete the section by giving the status of our implementation.

3.1 SynRGen Abstractions

Our design is based upon two key abstractions: *volumes* and *user classes*. The volume abstraction provides the basis for modeling the layout and storage characteristics of the file system, while the user class abstraction provides the basis for modeling user activity.

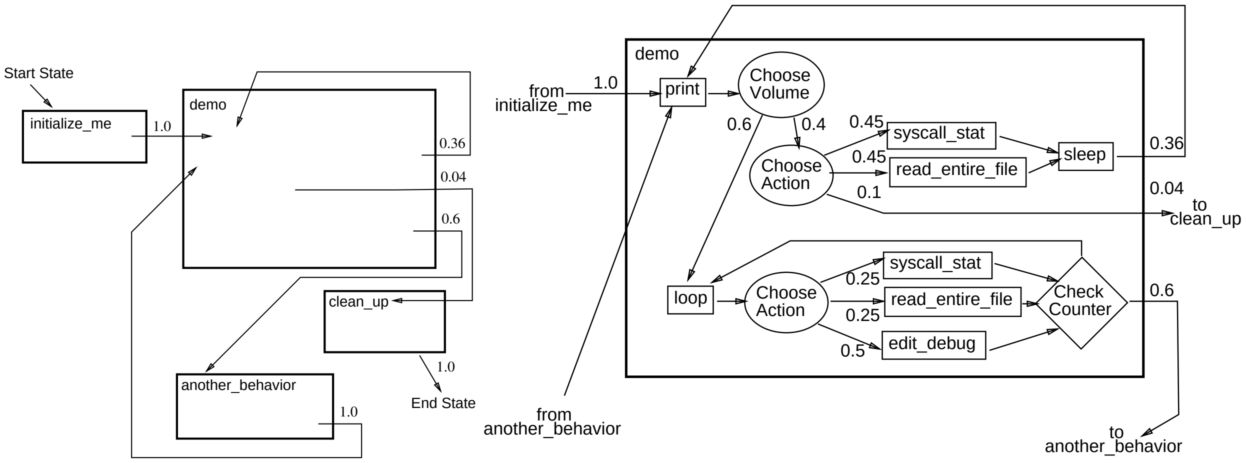
A volume is a subtree of files and directories exhibiting a unique combination of physical characteristics. Each characteristic, such as file size and directory fan-out, is described by a distinct volume-specific stochastic distribution. For simplicity, we assume volumes are mounted only at the root of the file system hierarchy. Table 1 summarizes the characteristics observed in different types of volumes from a representative file system.

A user class corresponds to a stochastic finite state machine. States in the FSM represent distinct user *behaviors*, while transitions represent a user changing from one behavior to another. For example, a behavior might be "programming" or "document processing". Each behavior consists of some set of possibly repetitive *actions*, corresponding to subtasks performed within this behavior. The actions associated with a "programming" behavior might include "searching header files", "editing" and "compiling", while the actions associated with a "document processing" behavior might include "editing" and "formatting". Actions exhibit distinctive file access characteristics, and correspond to micromodels.

Physical Characteristic	Volume Type				
	User	Project	System	BBoard	All
Total Number of Volumes	786	121	72	71	1050
Total Number of Directories	13955	33642	9150	2286	59033
Total Number of Files	152111	313890	113029	144525	723555
Total Size of File Data (MB)	1700	7000	1500	560	11000
Absolute Depth	4.3 (1.3)	6.3 (2.1)	6.0 (1.3)	5.3 (1.0)	5.7 (2.0)
Relative Depth	3.3 (1.3)	5.2 (2.0)	4.0 (1.2)	2.7 (0.8)	4.5 (1.9)
File Size (KB)	10.3 (65.0)	24.0 (145.7)	16.4 (72.6)	2.6 (7.0)	19.1 (118.0)
Directories/Directory	3.6 (13.4)	3.0 (4.5)	3.6 (10.4)	6.8 (19.4)	3.2 (8.3)
Files/Directory	14.6 (30.6)	16.2 (35.6)	15.9 (36.9)	66.9 (142.4)	15.7 (34.5)
Hard Links/Directory	3.7 (12.4)	2.0 (1.5)	4.0 (3.9)	0.0 (0.0)	3.4 (5.7)
Symbolic Links/Directory	4.1 (10.1)	3.4 (7.5)	13.6 (45.3)	6.0 (25.9)	6.3 (24.9)

This table summarizes various physical characteristics of system, user, project, and bulletin board ("bboard") volumes in AFS at Carnegie Mellon University in early 1990. These data were obtained via static analysis. We present only the mean and standard deviation (in parenthesis) here. The full data are represented in cumulative distribution functions. Absolute depth is measured from "/afs/cs.cmu.edu/user" for user volumes, "/afs/cs.cmu.edu/project" for project volumes, and so on. Relative depth is measured from the volume root.

Table 1: Sample Physical Characteristics by Volume Type



(a) Complete Finite State Machine (b) Detail of `demo` Behavior

Figure (a) shows the finite state machine for a hypothetical user class. Each of the states `initialize_me`, `demo`, `clean_up`, and `another_behavior` represent user behaviors. Figure (b) shows the detailed contents of one of these behaviors, `demo`. Each small rectangle in this figure, such as `edit_debug` and `read_entire_file`, corresponds to an action; these actions are implemented in micromodels. In both figures, arcs represent transitions between user behaviors, and the numbers on the arcs indicate transition probabilities.

Figure 1: User Class Finite State Machine

Figure 1(a) shows an example of a user class. This class of users exhibits four behaviors: `initialize_me`, `demo`, `another_behavior` and `clean_up`. The arcs in this figure represent transitions between behaviors; the numbers represent the probability of taking a particular transition. Figure 1(b) details a single behavior (`demo`) of that user class. The actions associated with the `demo` behavior include `syscall_stat`, `read_entire_file` and `edit_debug`.

Our representation of a user class closely resembles a *user behavior graph*, as defined by Ferrari[6]. Each behavior in a SynRGen user class corresponds to a node in the behavior graph, and each transition to an arc.

3.2 Configuration Files

SynRGen's volume and user class abstractions are described in configuration files which are transformed by preprocessors. The `mkclass` preprocessor transforms a user class into a C program representing a synthetic user. The `mkvol` preprocessor transforms a volume description into a C data structure accessed by micromodels through library routines. To simplify the compilation and linking of synthetic users, we provide a third preprocessor, `mksynrgen`, that takes a system configuration file and generates shell scripts.

```
/* Include volume type descriptions */
#include <system>
#include <hacker_project>

/* Volume Instantiations -- name:  volume_type */
sys:  system
codasrc:  hacker_project
synrgensrc:  hacker_project

/* Include user class descriptions */
#include <hacker>

/* User Declarations -- group:  user_class(parameters)  */
codahackers:  hacker()
synrgenhackers:  hacker(project1 = synrgensrc, mean_interarrival = 0.14)
```

Figure 2: Sample System Configuration File

This figure shows a sample system configuration file. Notice that the `synrgenhackers` user declaration redefines the default value of the `$project1` variable to "synrgensrc" and the value of the `mean_interarrival` variable to 0.14.

We describe SynRGen further with a set of examples. Figure 2 shows a typical system configuration file. The first section of this file defines volume descriptions by including volume configuration files, `system.vol` and `hacker_project.vol`. In the next section, we instantiate three volumes: `sys`, `codasrc`, `synrgensrc`. The first volume is of type `system`, and the other two are of type `hacker_project`. The syntax resembles the way in which C programs include header files to obtain `typedef` definitions and then instantiate variables of those types. In a similar manner, the rest of the file obtains definitions of the user class `hacker` and instantiates two different instances of this type of user. The instances differ in that `codahackers` uses the default set of parameters, while `synrgenhackers` redefines certain parameters.

Figure 3 depicts a portion of a volume configuration file. This file can contain up to six sections, each describing a physical characteristic of the file system hierarchy. The physical characteristics are: the file size; the number of files, symbolic links, hard links and directories per directory; and the relative depth. The characteristics are described using the inverse transformation of the corresponding CDF (cumulative distribution function). This information is transformed into a data structure used by the volume information routines (e.g. `get_FileSize`) accessible to micromodels.

FILESIZE:

0.07	0
0.10	100
0.13	200
...	...
...	...
0.99	400000
1.00	10000000

DIRS per DIRECTORY:

0.67	0
0.81	1
0.89	2
0.92	3
0.95	4
...	...
1.00	100

SYMLINKS per DIRECTORY:

0.91	0
0.95	1
0.98	3
0.99	9
1.00	90

... ...

This figure, containing excerpts of the `project` volume configuration file, shows three physical characteristics of this type of volume. This file contains the inverse transformation of the CDF for each characteristic. For example, 13% of all files in project volumes contain no more than 200 bytes of data. The data details the summary presented in Table 1.

Figure 3: A Portion of a Volume Configuration File

Figure 4 depicts the user class configuration file corresponding to Figure 1. The heart of the user class definition consists of the description of individual behaviors such as `initialize_me`, `demo`, `another_behavior` and `clean_up`. Descriptions of behaviors can use either arbitrary C code or our syntactic constructs for commonly encountered control flows. The `demo` behavior, for example, uses our syntactic constructs. After entering the `demo` behavior, the program will stochastically choose a volume in which to operate (either the `$project1` or the `sys` volume). If the project volume is chosen, the program will loop, stochastically choosing one of three actions (`syscall_stat`, `read_entire_file` or `edit_debug`) on each iteration. These actions are micromodels. If the `sys` volume is chosen instead, the program will stochastically choose to perform either a `syscall_stat`, a `read_entire_file` or a transition to the `clean_up` behavior. Notice that each of the micromodels take a pathname iterator, `FractileFallOff()`, as a parameter.

Parameters to the user class can be accessed within the configuration file by prepending a $ to their name, e.g. `$project1`. These variables are bound to a default value when they are declared, but can be rebound either at configuration time or at run time. Run time binding takes precedence over configuration time binding, which in turn takes precedence over the default binding. As shown in Figure 4, the default binding for the `$project1` volume is "codasrc". However, as shown in Figure 2, the system configuration file redefines this parameter to "synrgensrc" for `synrgenhackers`.

Arbitrary C code (surrounded by '{' and '}') also appears in many places within the body of Figure 4. By allowing experimenters to combine specialized syntactic constructs together with arbitrary C code, SynRGen provides a good balance between brevity and open-endedness.

3.3 Implementation Status

SynRGen has been operational since May 1992. The implementation is highly portable and has run under Mach on DEC MIPS workstations, Intel 386/486 machines, and IBM RTs. It has also been ported to run under AIX on IBM RS/6000 machines. The three preprocessors are implemented in C, using `lex` and `yacc`.

We have built up a small library of parameterized micromodels representing a variety of typical applications used in our environment. These include use of an `emacs`-style editor, C compilation, and building programs via `make`. Using micromodels and user classes representative of our environment, we have used SynRGen extensively for stress testing new releases of the Coda File System. Also, we have conducted controlled experiments to evaluate how realistically SynRGen models a true Coda file server workload. The results of our experiments are reported in the next section.

```
/*****************************************************************
 *                    SynRGen USER Configuration File                *
 *****************************************************************/
{
    #include "behavior.h"
    #include "fract_falloff.h"
    #include "volume_info.h"

    Fractile_FallOff_Info system_info, project_info;
    Volume_Info System_Volume, Project_Volume;
}

/* Parameter Declaration and Initialization */
double mean_interarrival = 0.10227;
double ckp_interval = 30.0;
int pause = 10;
int loop_times = 5;
char project1[10] = "codasrc";

initialize_me:
    {
        printf("SynRGen initialized.  Beginning SynRGen demo in %d seconds\n", pause);
        sleep(pause);
        BEGIN(demo);
    }

demo:
    { printf("\n\nSTART: demo behavior\n"); }
    < 0.60 $project1
         loop [$loop_times]
            <0.25 syscall_stat(Fractile_FallOff(), &Project_Volume, &project_info, DIR_OBJ)>
            <0.25 read_entire_file(Fractile_FallOff(), &Project_Volume, &project_info)>
            <0.50 edit_debug(Fractile_FallOff(), &Project_Volume, &project_info, ckp_interval)>
        endloop
        { BEGIN(another_behavior); }
    >

    < 0.40 sys
        <0.45 syscall_stat(Fractile_FallOff(), &System_Volume, &system_info, DIR_OBJ)>
        <0.45 read_entire_file(Fractile_FallOff(), &System_Volume, &system_info)>
        <0.10 { BEGIN(clean_up); } >
    >
    { sleep(Exponential(mean_ interarrival)); }

another_behavior:
    {
        /* This section of the configuration file would model another behavior.  */
        BEGIN(demo);
    }

clean_up:
    {
        printf("\n\nSynRGen run complete!\n");
        exit(0);
    }
```

This figure shows the user class configuration file corresponding to the user class finite state machine shown in Figures 1(a) and (b). The demo behavior exemplifies SynRGen's syntactic constructs which simplify modeling common control flows. In this behavior, we see three distinct actions syscall_stat, read_entire_file, and edit_debug. These actions correspond to micromodels. Arbitrary C code appears in numerous places in the file: at the top of the file, in well-defined points within the demo behavior, and as the entirety of three behaviors (initialize_me, another_behavior, and clean_up). The BEGIN statements that appear throughout this file represent transitions between behaviors. Although the Fractile_FallOff() parameter looks like a function call, the preprocessor translates this into a pointer to a function so that the micromodels can use it as a pathname iterator.

Figure 4: Sample User Class Configuration File

4 Case Study: Users in an Edit/Debug Cycle

An important question is "How well does a SynRGen workload emulate a real workload?" To answer this question, we conducted a *performance-oriented* study [6] comparing real users to a synthetic user. We chose to study the edit/debug cycle because it is a common activity in our environment as well as many others. We wrote micromodels for the most frequent activities in an edit/debug cycle, generated a synthetic user, and then performed a controlled experiment to compare the load generated on Coda servers by the synthetic user to that generated by real users.

4.1 The Synthetic User

Our first task was to build micromodels of the tools most frequently used during edit/debug cycles. To build these micromodels we examined many traces generated by a number of users working in an edit/debug cycle on a variety of machine architectures.

After completing the micromodels, we constructed a synthetic user. The main activity of this synthetic user is an edit/debug cycle that consists of editing some number of files associated with a particular "program", recompiling the modified files, and executing the program after it is built. In addition, the synthetic user occasionally performs a few other related activities such as consulting a manual page, examining a system header file or looking through a source file. Broadly, the user resembles the example shown in Figure 4. The important user class parameters are the mean interarrival time of file system requests, and the mean checkpoint interval time for the `emacs`-style editor.

4.2 The Experiment

For our experiment, we observed five C or C++ programmers as they performed edit/debug activity for one hour. We made no attempt to rigidly constrain these users – our only request was that they confine their activities to a single workstation. Inevitably, some activities not modeled by SynRGen were performed, but we believe that the amount of such activity was minimal.

These users were working on DEC DS-5000/200 machines running Mach 2.6. The data they were accessing resided primarily in triply replicated volumes located on three Coda servers. Each server was also a DS-5000/200 machine running Mach 2.6. The client and server machines communicated via an Ethernet.

The system parameters we measured on the Coda servers include distribution of incoming RPC operations, transactional activity, CPU utilization, disk activity, and Ethernet load. In addition to these server and network statistics, we obtained file reference traces of these users on the client machines.

The mean interarrival time parameter of our synthetic user was set to the mean interarrival time of file system requests observed in the traces of these users (about 0.1 seconds). The mean checkpoint interval parameter to our synthetic user was set using an estimate of the frequency of checkpoints in actual use of our editors (30 seconds). In addition, the pathname iterator to our micromodels was the fractile falloff distribution described by Satyanarayanan [15]. Specifically, 75% of the time, 4% of the files were referenced; 20% of the time, 16% of the files were referenced; and the remaining 5% of the time, the other 80% of the files were referenced.

Using these parameters, we then ran the synthetic user on one of the client workstations in a subtree of the Coda file system. The physical characteristics of files in this subtree were consistent with the data presented in Table 1 for project volumes.

Characteristic	Real Users							Synthetic User	Relative Error	Stdzd Error
	User 1	User 2	User 3	User 4	User 5	Mean (StDev)	COV			
Total Time (sec)	3642	3379	3603	3602	3597	3565 (105)	0.0	3602	1%	0.4
Total Operations	3431	2177	2898	2085	2601	2638 (551)	0.2	2590	-2%	0.1
SERVEROPS:										
FetchData	32	30	63	4	28	31 (21)	0.7	11	-65%	1.0
FetchStatus	1219	1061	1462	1161	1278	1236 (149)	0.1	1078	-13%	1.1
StoreData	269	29	70	72	35	95 (99)	1.0	167	76%	0.7
StoreStatus	263	48	49	59	21	88 (99)	1.1	115	31%	0.3
Create	117	26	21	22	7	39 (44)	1.2	33	-15%	0.1
RVM:										
Transactions	2158	437	422	403	394	763 (780)	1.0	686	-10%	0.1
CPU TIME:										
User (sec)	81	90	81	63	68	77 (11)	0.1	63	-18%	1.3
System (sec)	163	169	244	132	133	168 (46)	0.3	134	-20%	0.7
DISK:										
Transfers	7804	2434	11735	2535	3518	5605 (4070)	0.7	3968	-29%	0.4
KBs Transferred	57112	15891	86862	15046	22359	39454 (31633)	0.8	22420	-43%	0.5
Busy Time (sec)	137	45	196	46	65	98 (67)	0.7	69	-30%	0.4
ETHERNET:										
Packets In	123800	129365	163207	117353	107830	128311 (21088)	0.2	114245	-11%	0.7
Packets Out	27485	34737	38847	25867	24888	30365 (6115)	0.2	25205	-17%	0.8

This table presents the results of a controlled experiment comparing the workload generated on Coda servers by five real users to that generated by a synthetic user. For each "user", we present the mean value observed at the three servers. In addition to the load generated by each real user, we present the mean, standard deviation and coefficient of variation for these users. The relative error is defined as the ratio of the difference between the synthetic user and the mean of the real users to the mean of the real users. A relative error greater than zero implies that the synthetic load overestimated the actual load, while an error less than zero implies the synthetic load underestimated. The standardized error is defined as the ratio of the absolute value of the difference between the synthetic user and the mean of the real users to the standard deviation of the real users.

Table 2: Comparison of Real and Synthetic Users

4.3 Results

Table 2 compares the values of the system variables obtained from our real users to those obtained from our synthetic user. The last column of this table calculates the difference between the synthetic user and the mean of the real users in units of the standard deviation of the real users. This column shows that all but two system variables (the number of status fetches and the CPU usage in user mode) fall within one standard deviation of the average for the real users. For most system variables including the two above, the synthetic user comes within 20% of the mean value for the real users. With the exception of the number of data fetch and store requests, all the other system variables of interest lie within about 40% of their observed values for the real users. In all cases where the results from the synthetic user diverge substantially from the mean value for the real users, the result from the synthetic user still falls within the range observed for real users.

A final observation is that SynRGen consistently underestimates the observed values from real users (i.e., relative error is negative for most system variables). This suggests that we might be able to get the synthetic user to better match real users by applying a correction factor to the run time parameters. Experiments not reported here confirm that this is indeed the case.

It is important to note that the bulk of our effort in this experiment was in building the micromodels. The actual construction of our synthetic user was relatively simple. This confirms the underlying hypothesis of our approach: that it is possible to substantially separate the efforts of the modeler and experimenter, and to encapsulate the work of the former in micromodels.

Our results confirm that SynRGen is able to realistically model users in at least one domain. Further validation of our approach would require similar controlled experiments spanning a broader class of activities, applications, and environments.

4.4 Extending the Case Study

In the above sections we used SynRGen to model a single user in an edit-debug cycle. Suppose, however, that we need to model an entire user community. What would the necessary changes be? The first step is creating validated SynRGen models for each class of user in the community under study. Once available, these synthetic users can be run simultaneously to generate a workload representative of the entire community.

Almost certainly, we will want to model data sharing between users. Sharing can be read-read, read-write or write-write. For simplicity, we refer to read-read sharing simply as "read-sharing" and to both read-write and write-write sharing as "write-sharing".

We model read- and write-sharing of files and read-sharing of directories by having synthetic users perform activities in shared volumes. For example, to model read-sharing of manual pages or header files, all members of a community might occasionally examine the contents of files and directories in shared system volumes. Similarly, when modeling members of a project, one would concentrate their activities in one or more project volumes. Increasing the time spent performing activities in the shared volume will increase the probability of sharing.

Modeling write-sharing of directories is more challenging. Synthetic users create, remove and rename objects as dictated by their micromodels. When synthetic users write-share directories, they may experience interference caused by these directory updates. For instance, one synthetic user might remove an object that another synthetic user later attempts to edit. The micromodels in Section 4.1 do not update internal data structures upon discovering new, missing or renamed objects. In order to support write-sharing of directories, we must modify those micromodels to use failed system calls as hints to trigger updates to internal data structures. One of SynRGen's strengths is that these improvements require modifications only to the micromodels and not to SynRGen itself.

5 Open Issues and Future Work

5.1 Automatic Micromodeling

Because writing micromodels is a labor intensive task, automating this process in some fashion would make modeling new activities easier. One approach would be to "invert" a file reference trace, producing a command script. This command script, when executed, would produce a trace isomorphic to the original trace. We have experience with an *untrace facility* that does just that [17]. A difficulty with such an approach is determining how to parameterize the generated micromodels, and how to specify locality.

Another approach would be to use the strategy proposed by Thekkath et al. [20]. This approach recognizes that random samples selected from a representative trace of the workload being modeled will result in workloads that exhibit the same statistical characteristics as the original. These random samples could be used as micromodels. Parameterizing these micromodels must still be done manually.

Both of these techniques require an accurate trace of the environment being modeled. This requirement limits their applicability to environments that already exist – one cannot use either of these techniques for constructing a micromodel for a hypothetical or anticipated application.

5.2 Incorporating Locality Phases

SynRGen provides a means of modeling locality of reference in a synthetic workload. In Bunt et al.'s terminology [2, 3], SynRGen offers flexibility in modeling any degree of *concentration* of locality; that is, SynRGen allows any number of files to be in the active set. However, Majumdar and Bunt have shown that file reference locality exhibits phases, or *bounded locality intervals* [8]. Although we do not foresee any difficulty in incorporating locality phases, SynRGen currently has no notion of these phases. The consequence of not supporting bounded locality intervals is that SynRGen can be expected to display a higher degree of locality than actual workloads. Adding the notion of locality phases to SynRGen would improve the accuracy of modeling.

As seen in Figure 2, the synthetic user greatly underestimated the number of fetches seen by the servers. One possible explanation of this behavior is that the client cache was able to service fetch requests for the synthetic user more successfully than for the real users. In other words, the synthetic user may be displaying a higher degree of locality than real users.

5.3 Quantifying Accuracy of Micromodels

The overall realism of an experiment is limited by the accuracy of individual micromodels. This raises the question of how to quantify the accuracy of a micromodel. One approach would be to define some metrics and use these metrics to compare traces generated by the synthetic user to a set of reference traces generated in a real system. These metrics might include the fraction of files referenced and the locality of files referenced. While such metrics give some indication of the accuracy of the micromodel, they completely ignore the order in which events happen. In modeling locality of reference, it is important to capture the order in which system calls occur as well as the order in which files and directories are accessed.

One metric that respects ordering is the *longest common subsequence* (LCS) . The LCS is a well-known measure of closeness between two strings and has been used in a variety of contexts such as DNA sequencing and speech processing[13]. We explored the possibility of using this metric to quantify the accuracy of micromodels. Unfortunately, we found this approach to be intractable due to the large storage requirements of the algorithm and the voluminous size of realistic traces.

Quantifying accuracy thus remains an open problem. We now believe that such quantification must be based either on metrics other than the LCS, or on a more efficient approximation to the LCS.

6 Related work

Synthetic file reference generation has received considerable attention in the recent past. Much of this attention has been focused on measuring NFS performance. For example, NHFSStone [9] facilitates comparisons of competing NFS implementations. A fundamental distinction between this work and ours is the dependence of this benchmark on the NFS interface. Further, this benchmark is considerably less realistic and less flexible than SynRGen.

Bodnarchuk and Bunt [1] significantly improve upon the above benchmark in both flexibility and realism. Their algorithm involves sampling discrete distributions to choose an NFS operation, a file system in which to operate (uniformly choosing a specific file from a "representative set" of this file system), a data transfer size, and an interarrival time. SynRGen differs from this work in that it raises the generator up to the file system call level, models interfile locality in addition to intrafile locality, and allows measurement of client effectiveness as well as server and network effectiveness.

In contrast to NFS benchmarks, SynRGen operates at the file system call level. This makes performance comparison of file system implementations such as AFS and NFS possible. It also broadens the range of phenomena that can be modeled.

Another widely used benchmark, the Andrew Benchmark[7], operates at the Unix system call level and attains a respectable degree of realism. Because this benchmark is restricted to specific activities, it cannot be used to model a variety of workloads. In contrast, SynRGen is considerably more flexible and allows a wide range of scenarios to be modeled.

Viewing SynRGen in a broader context, it is clear that the idea of generating a synthetic workload is not new. In fact, there are a wide variety of synthetic workload generators including the SPEC benchmark suite[19], the TPC benchmarks[18], IOBENCH[21], tcplib[5], and many others. What differentiates SynRGen from other workload generators is its flexibility in accommodating new workloads while preserving realism. It achieves this flexibility by providing not a single workload generator, but a common framework on which workload generators can be built. The separation of micromodels from their stochastic combination is, to the best of our knowledge, not duplicated in any other synthetic workload generator.

7 Conclusion

SynRGen is a tool born of necessity. Early in the evolution of the Coda File System, it became clear to us that we needed some way of stressing our system without burdening real users. In retrospect, SynRGen has indeed proven to be an invaluable tool for this purpose. Coda is now used daily by over 30 users as their primary data repository. New releases of Coda are exposed to synthetic users for an extended period of time before they are installed on our production servers. Furthermore, SynRGen proved invaluable in debugging our current backup system.

We are confident that SynRGen's unique ability to combine realism with flexibility will make it attractive to other file system developers. As discussed earlier in the paper, we foresee it being useful in performance evaluation. We also envision it allowing researchers to subject their systems to a broader range of workloads than has historically been possible. This will enhance the credibility of research systems, whose generality has often been questioned because of their bias toward academic workloads.

While already useful in its present form, SynRGen is not a finished piece of work. There is clearly work to be done in building up a rich library of micromodels, representing a wide range of applications. A related piece of work is to assemble a collection of configuration files capable of representing a variety of usage environments. Finally, as Section 5 explained, SynRGen itself can be refined in a number of ways. We believe that these efforts will result in an important asset to the file systems community.

8 Acknowledgements

We wish to thank Lily Mummert for her indispensable file reference tracing facility. In addition, we would also like to thank our "real" users and those who provided comments on early versions of this paper, including Puneet Kumar, Qi Lu, Scott Nettles, Brian Noble, Chris Okasaki, Josh Raiff, David Steere and Matt Zekauskas. Finally, we would like to thank the referees for their many valuable comments.

References

[1] BODNARCHUK, R. R., AND BUNT, R. B. A Synthetic Workload Model for a Distributed System File Server. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (San Diego, CA, May 1991).

[2] BUNT, R. B., AND MURPHY, J. M. The Measurement of Locality and the Behavior of Programs. *The Computer Journal 27*, 3 (August 1984).

[3] BUNT, R. B., MURPHY, J. M., AND MAJUMDAR, S. A Measure of Program Locality and Its Application. In *Proceedings of the 1984 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Cambridge, MA, August 1984).

[4] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[5] DANZIG, P. B., JAMIN, S., CÁCERES, R., MITZEL, D. J., AND ESTRIN, D. An Empirical Workload Model for Driving Wide-area TCP/IP Network Simulations. *Internetworking: Research and Experience 3*, 1 (March 1992).

[6] FERRARI, D. On the Foundation of Artificial Workload Design. In *Proceedings of the 1984 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Cambridge, MA, August 1984).

[7] HOWARD, J. H., KAZAR, J. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., AND SIDEBOTHAM, R. N. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[8] MAJUMDAR, S., AND BUNT, R. B. Measurement and Analysis of Locality Phases in File Referencing Behavior. In *Proceedings of Performance 86 and ACM Sigmetrics 86, Joint Conference on Computer Performance Modelling, Measurement and Evaluation* (Raleigh, NC, May 1986).

[9] MOLLOY, M. K. Anatomy of the NHFSSTONES Benchmarks. *ACM SIGMETRICS Performance Evaluation Review 19*, 4 (May 1992).

[10] MUMMERT, L. B., AND SATYANARAYANAN, M. Efficient and Portable File Reference Tracing in a Distributed Workstation Environment. Carnegie Mellon University, manuscript in preparation.

[11] NELSON, M., WELCH, B., AND OUSTERHOUT, J. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[12] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX Summer Conference* (Summer 1985).

[13] SANKOFF, D., AND KRUSKAL, J. B. *Time Warps, Strings Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1983.

[14] SATYANARAYANAN, M. The ITC Distributed File System: Principles and Design. In *Proceedings Tenth ACM Symposium on Operating Systems Principles* (December 1985).

[15] SATYANARAYANAN, M. *Modelling Storage Systems*. UMI Research Press, Ann Arbor, MI, 1986.

[16] SATYANARAYANAN, M., KISTLER, J. J., KUMAR, P., OKASAKI, M. E., SIEGEL, E. H., AND STEERE, D. C. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers 39*, 4 (April 1990).

[17] SATYANARAYANAN, M., KISTLER, J. J., MUMMERT, L. B., EBLING, M. R., KUMAR, P., AND LU, Q. Experience with Disconnected Operation in a Mobile Computing Environment. In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing* (Cambridge, MA, August 1993).

[18] SERLIN, O. The History of DebitCredit and the TPC. In *The Benchmark Handbook*, J. Gray, Ed. Morgan Kaufman, 1991.

[19] SYSTEM PERFORMANCE EVALUATION COOPERATIVE. *SPEC Benchmark Suite Release 1.0*, October 1989.

[20] THEKKATH, C. A., WILKES, J., AND LAZOWSKA, E. D. Techniques for File System Simulation. Technical Report 92-09-08, University of Washington, Department of Computer Science and Engineering, 1992.

[21] WOLMAN, B. L., AND OLSON, T. M. IOBENCH: A System Independent IO Benchmark. *Computer Architecture News 17*, 5 (September 1989).