BUILDING A BETTER GPSS:

A 3:1 PERFORMANCE ENHANCEMENT

James O. Henriksen
CACI, Inc.

INTRODUCTION

This paper describes the development of GPSS/H, a new GPSS for the IBM 360/370 and Amdahl 470 computer systems. Preliminary results indicate that the new system runs well over three times as fast as its IBM counterpart, GPSS/360.

The first section of this paper describes a detailed analysis made of the run-time performance of GPSS/360. The techniques used and the shortcomings identified are described in detail. Next, the design goals of GPSS/H are presented, followed by a brief overview of the new implementation. The new implementation departs radically from the traditional interpretive approach found in many simulators and constitutes a frontal attack on the shortcomings identified by performance measurement of GPSS/360. Finally, performance comparisons are made between GPSS/H and GPSS/360.

The efforts described herein were made during the period 1968-1974, with a good share of the effort predating the existence of GPSS/V. In addition, the author never had at his disposal a copy of GPSS/V for analysis and modification. Since the emphasis in the development of GPSS/V was language extension, rather than performance enhancement, many of the deficiencies identified in GPSS/360 still exist in GPSS/V. The analysis presented herein is, therefore, largely applicable to GPSS/V.

The version of GPSS/360 analyzed was Version I, Modification Level 4, as modified to run under the control of The University of Michigan Terminal System, MTS (1). All the computer runs described herein were made at the University of Michigan Computing Center, where the hardware configuration at the time was a dual processor IBM 360/67. Accordingly, all CPU times cited are for the 360/67.

PERFORMANCE ANALYSIS OF GPSS/360

Performance Measurement Techniques Used

Three basic techniques of performance measurement were used in analyzing GPSS/360. In order of increasing complexity, they were (1) simple instrumentation achieved by making minor insertions in the code, (2) the use of a software probe, and (3) instruction counts and timings made by manually counting the instructions required to perform certain functions and looking up the corresponding instruction execution times.

Instrumentation

Instrumentation was inserted into GPSS/360 to provide CPU times for the assembly, input, execution, and output phases of a model, to count the total number of blocks executed, and to calculate the average number of microseconds of CPU time per block executed. These measurements were rather easy to provide and, while technically unsophisticated, provided some very useful insight. "Typical" models were found to require around 450 microseconds per block. This figure may be used as a guideline for assessing the efficiency of the implementation of a model. A model which requires more than 500 microseconds per block, for example, might be suspected as inefficient. In the absence of heavy computational requirements, e.g., function and variable evaluations, the model should be highly suspected of inefficient implementation. Such a model might be a candidate for the introduction of User Chains to increase efficiency, as discussed by Schriber (2).

Further use was made of the instrumentation facilities to measure the average execution times of a number of different GPSS block types. The results of these measurement runs are shown in figure 1. In each case, a model consisting of 1000 replications of the given block(s) contained in a loop was executed. The resultant timings for block pairs shown are artificially lower than they would be in a "real" model, because of the fact that no simulated time elapses, thereby rendering the updating of utilization integrals trivial.

The information provided by instrumentation was important in two respects. First, it provided some initial insight into the overall performance of GPSS/360. Second, it provided the impetus for more detailed analysis: because the execution times recorded seemed inordinately large, a compelling urge arose to determine in greater detail exactly where the CPU time was consumed.

Block(s) Timed		Avg. CPU Time (Microsec.)
SAVEVALUE	1,0	186
ASSIGN	1,0	215
SEIZE	1)	
RELEASE	1)	258
QUEUE	1)	
DEPART	1)	230
ENTER	1)	
LEAVE	1)	259
ADVANCE	10	534
ADVANCE	10,5	733
ADVANCE	100,FN$EXPO	954

Figure 1 - GPSS/360 Average Block Execution Times

Use of a Software Probe

In order to make a more refined timing analysis, numerous runs were made of GPSS/360 under the control of the *TIMETALLY software probe of MTS. By scheduling periodic timer interrupts and noting their subsequent points of occurrence, *TIMETALLY produces histograms which show where CPU time is consumed in the address space of a program. Using this technique, a number of bottlenecks were discovered, including the DCOD, PREV, and UPBLK subroutines. The DCOD subroutine decodes the operands of a block and calls a subroutine to execute the block itself. Its execution is highly interpretive, and significant overhead is involved, even in the case of blocks which have no operands. The PREV subroutine is called every time a non-GENERATE block is entered, in order to perform two tests. First, if tracing is enabled, the output phase is invoked to produce the trace information. Second, a test is made to see whether the entering transaction is leaving a GENERATE block. If so, the successor arrival into the GENERATE block is scheduled if the limit count has not been reached. If tracing is disabled, for all blocks which do not immediately follow GENERATE blocks, the call to PREV accomplishes absolutely nothing. The UPBLK subroutine updates the overhead information, e.g., block counts, associated with the successful completion of a block. The DCOD, PREV, and UPBLK subroutines were found to account for CPU times ranging from 120 microseconds in trivial models to nearly 200 microseconds in one very complex model. In view of the 450 microsecond figure identified above, these routines consume a significant portion of the CPU time in a GPSS/360 run.

Instruction Counts and Timings

Further analysis was carried out by examining the assembly code listings of GPSS/360, beginning with bottlenecks identified in *TIMETALLY runs. The instruction counts and timings determined for representative functions are presented below, where direct comparisons are made with the corresponding figures for GPSS/H. The further discussion of quantitative results is deferred to that more meaningful context. We now proceed with a discussion of the quantitative conclusions that were reached.

Shortcomings in the Implementation of GPSS/360

Poor Run-time Representation - The largest shortcoming of GPSS/360 is the poorly designed run-time representation of models. As a consequence of the poor representation, the addressing of GPSS entities, the maintenance of overhead information, and the computation of utilization integrals are particularly costly.

In GPSS/360, nearly all entities are referenced internally by integer indices. Thus, to access the data associated with a particular entity, its index must be multiplied by the number of bytes per entity (A shift is used if this number is a power of 2.) and the resultant product must be added to the starting address for entities of the type in question. "Load-multiply-add" instruction sequences are therefore prevalent where single "load" instructions could be used if entity references were maintained internally as actual machine addresses.

Because block information is accessed in the same way as other entities, the maintenance of overhead information (current and total block counts, current block number, and previous block number) is quite costly. Instruction count and timing information presented below demonstrates that typically at least 11% of the CPU time in a GPSS run is consumed in this activity.

The utilization integrals for queues, user chains, and storages are maintained as double precision floating point values. The values upon which these integrals are based, elapsed time and current contents, are integer variables, implying the need for integer-to-floating-point conversion. The use of (software-implemented) double precision integer integrals enables accumulation of the same statistics in one sixth the time, as evidenced below.

Inconsistent Register Usage - The GPSS/360 simulator is comprised of a large number of relatively small subroutines. The execution of a block might typically involve execution of a half-dozen or more subroutines. The fact that machine register usage in these routines is somewhat inconsistent is indicated by the prevalent practice of saving and restoring register contents at subroutine entry and exit, respectively. The PREV subroutine provides a classic illustration of the cost of this approach. The nine instructions executed in a typical call of PREV consume 23 microseconds of CPU time on the 360/67. Of these nine instructions, the "store multiple" and "load multiple" instructions, which are used to save and restore registers, consume approximately 13 microseconds, more than half the execution time of the whole subroutine. Although this in part reflects the peculiar architecture of the 360/67, in which "multiple" instructions are costly, the overstatement is only slight. The frequent shuffling of register contents is expensive.

Redundant Error Checking - One of the fundamental rules of simulation modelling is that complete error checking must be done at execution time. Because of the non-deterministic nature of most simulations, logical errors might go undetected in the absence of complete error checking. In GPSS/360, an overly conservative error checking philosophy was adopted. Nearly every subroutine, including those at the lowest levels, checks the validity of its arguments. An alternative approach might be to have the calling routine validate only those arguments which could possibly be invalid. If this is too dangerous, another strategy, widely used in GPSS/H, is to provide primary and secondary subroutine entry points, which perform and omit, respectively, error checking. If a user codes a "SEIZE JOE" block in GPSS/360, the number assigned to the symbol JOE by the assembly phase is repeatedly checked for validity each time the SEIZE block subroutine is called. It is particularly vexing to see any program repeatedly revalidate a constant value which it, in fact, generated.

Interpretive Operation - Interpretive operation implies generality: an interpretor must be able to interpretively perform all operations allowable at a given point. A second hallmark of interpretive operation is that decision logic in the program being interpreted is represented in the form of data, rather than in the form of executable machine instructions. An interpretor, therefore, requires generalized code for converting decision logic into actual branches within the interpretor. It cannot be nearly as efficient as machine instructions generated to carry out the same logic directly. In GPSS/360, the decoding of block operands is highly interpretive. In "typical" models, this activity consumes 20-25% of the CPU time and, in complex models, may exceed 30%. In GPSS/H, block operands are evaluated by compiler-generated object code. The superiority of this approach is evident in the results presented below.

DESIGN GOALS OF GPSS/H

The analysis of GPSS/360 gave rise to the desire to design a new implementation of GPSS which would eliminate the inefficiencies identified. The goals of this redesign effort will now be briefly presented.

Upward compatibility and Language Enhancements

In order to be genuinely useful to 360/370/470 users, any new version of GPSS, it was felt, had to be upward compatible with GPSS/V. This meant that models acceptable to GPSS/V could be processed without changes to the GPSS source code. The results would not be identical, because (1) different random number generators would be used, and (2) bugs identified in GPSS/360 would not, of course, be duplicated. Since the new system would have a totally different internal structure, models employing the HELP block would probably require modification.

While the principal goal of the redesign was performance enhancement, provision was also made for inclusion of a number of language extensions. Most significant among these was the capability for use of arbitrarily complex expressions as block operands. Since an expression compiler had to be built anyway, it was actually easier to include this capability than to force exact compatibility with GPSS/V. Another language extension provided for was the inclusion of standard parenthesis notation for specification of entity indices.

Implementation as a compiler

GPSS/H was designed to be a true compiler, capable of generating machine instructions for evaluation of arbitrarily complex expressions. The design goal was to generate object code in all cases where it was practical and to generate calls to run-time support routines in cases where generation of object code was impractical.

3:1 Performance Enhancement

The performance analysis of GPSS/360 indicated that a 3:1 enhancement could surely be achieved, and this figure was adopted as a design goal. More emphasis was to be placed on speed than on size. If significant speed increases could be obtained at the cost of modest increases in size, then the decision was to be made in favor of speed.

Debugging Aids and Instrumentation

To facilitate debugging and analysis of the new system, an internal trace feature and extensive instrumentation were provided for in the design. These features were to be implemented in such a way that the "production" system could easily be assembled with these features omitted, eliminating their relatively high execution cost.

Coding Standards

GPSS/H was designed to be implemented in assembly language. Coding standards to be employed included (1) a comment density of at least one comment per machine instruction, (2) liberal use of macros, (3) use of symbols for all register and flag bit references, (4) modularized code, with typical routines being one or two listing pages in length, and (5) uniform register usage conventions. Wherever possible, instruction sequences were to be ordered to take advantage of parallelism available in high-performance CPUs, such as the IBM 360/195.

IMPLEMENTATION OF GPSS/H

Overall System Architecture

The overall architecture of GPSS/H is shown in figure 2. The functions of each of the major modules will now be briefly discussed. The CONTROL module oversees the entire compilation and execution of a run and keeps track of the CPU time used by other modules. PASS I is the first pass of compilation. It reads and lists the source program and translates it into an internal representation, which is a combination infix and Polish-postfix notation. The compiler employs an extremely fast recursive descent parser, which was hand-coded from a formal grammar developed for GPSS expressions. Excellent compile-time diagnostics are given. Interested readers should refer to Gries (3) for a complete presentation of these topics. The PASS I INTERLUDE assigns values to symbols and prints out a complete cross-reference listing. For explicit references to entities, e.g., "SEIZE FRED" or "ENTER 17", the maximum number of the appropriate entity class is automatically adjusted upward, if required, relieving the user of the burden of having to REALLOCATE the entity class. PASS II reads the internal representation of the program, produced by PASS I, and generates object code to perform the required functions. The PASS II INTERLUDE loads the object code and relocates relocatable addresses in preparation for execution. As of this writing, the compiler does not produce object module output, but this capability could easily be added to the PASS II INTERLUDE. The SIMULATOR is a collection of subroutines which controls the execution of a model and provides those services for which compilation of in-line object code is impractical. The OUTPUT module produces all run-time output. The SYSTEM INTERFACE module contains all operating system-dependent functions, in order to facilitate transportability to different systems.

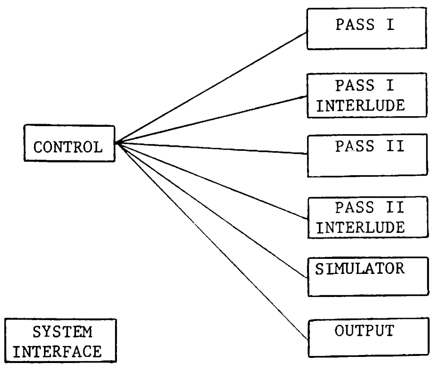

Figure 2 - Overall System Architecture

Architecture of the SIMULATOR

Interaction with Compiled Code - The interaction between compiled code and SIMULATOR is depicted in figure 3. For non-trivial blocks, e.g., SEIZE, in-line code is generated by the compiler for evaluation of the block's operands, and a call is constructed to carry out the actual block operation. The block subroutine may, in turn, call on simulator subroutines to perform its tasks. For trivial blocks, e.g., SAVEVALUE, in-line code is generated to perform the entire block operation. Similarly, trivial SNAs are evaluated in-line, and complex ones are evaluated by subroutine call.

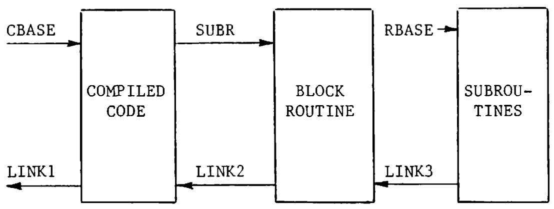

Figure 3 - Interaction of Compiled Code with the Simulator

General Register Usage - A great deal of the high performance of GPSS/H is due to carefully designed conventions for general register usage. These conventions, summarized in figure 4, are based on two important underlying principles: (1) frequently referenced items must be kept in registers at all times; and (2) subroutine calls, as depicted in figure 3, must, because of their high frequency of occurrence, be made with a minimum of overhead. Note that twelve of the registers have dedicated functions, leaving only four registers available for evaluation of block operands. (In the absence of SNA subroutine calls, registers SUBR and PARM may be used, making a total of six.) Since block operands tend (Remember, GPSS/H allows expressions as block operands.) to be relatively simple, this is sufficient. When a transaction is dispatched during a scan of the current events chain, registers OLDBLK, CURBLK, LINK1, and CBASE are loaded, in a single instruction, and this is all that is necessary to establish the transaction's current environment. The LINK1 register is dedicated to (1) scheduling successor arrivals into GENERATE blocks, (2) the TRANSFER-ALL and TRANSFER-BOTH blocks, and (3) the EXECUTE block. In all three cases, the register holds a special, non-sequential return address that overrides the flow of control that would otherwise normally take place.

No.	Name	Usage
0	ONE	Always contains a 1
1	LINK2	Primary link register
2	XBASE	Points to current xact
3	OLDBLK	Points to previous block
4	CURBLK	Points to current block
5	LINK1	Special link register
6	CBASE	Compiled code base reg
7	LINK3	Secondary link register
8	WORK1	
9	WORK2	
10	WORK3	
11	WORK4	Work registers
12	RBASE	Simulator base reg
13	RDATA	Data base reg
14	SUBR	Subroutine base reg
15	PARM	Parameter register

Figure 4 - Register Usage Conventions

COMPARATIVE PERFORMANCE OF GPSS/360 AND GPSS/H

In the paragraphs which follow, results are presented for four cases in which performance comparisons were made. The first three comparisons are made on the basis of instruction counts and timings, while the fourth is made on the basis of actual run times. The first case, an "ENTER 1" block, was chosen as a representative non-trivial block basic to the language. The second case, a "SAVEVALUE 1,0" block, was chosen as a representative trivial block in the language and dramatically illustrates the high cost of overhead in GPSS/360. The third case compares miscellaneous primitive operations vital to carrying out a simulation. The final case presents the results of benchmark runs of a one-line, single-server queuing model. A prototype version of GPSS/H was used which had the necessary subset of the language implemented in both compilation and execution phases of the system.

Results for the ENTER Block

The instruction counts and timings for an "ENTER 1" block are shown in figure 5. The functional division of the activities performed to execute the block is made on the basis of GPSS/360 subroutines executed. Although these subroutines do not explicitly exist in GPSS/H, the same functional divisions of machine instructions were used, in order to facilitate comparison. The first line of figure 5 shows the effort required to carry out the logic of the ENTER block proper. Since the skeletal logic of the block is the same in both cases, the instruction counts are fairly close. The difference in times is attributable to a superior representation in GPSS/H, in which important values are kept in registers, reducing the number of (slower) storage references. The effort required to decode the ENTER block arguments is much greater for GPSS/360 because of the inefficiency of the highly interpretive DCOD subroutine, described above. The four instructions employed in GPSS/H, to satisfy the curious reader, are a load of the address of storage 1, a load of a (default B-operand) value of 1, a load of the address of the ENTER subroutine, and a branch-and-link instruction to call it. Because the validation of "1" as a storage number is done in the PASS I INTERLUDE of GPSS/H, and calculation of its address is done in the PASS II INTERLUDE, both of which precede execution, GPSS/H requires no run-time effort for these operations, while GPSS/360 consumes 18 microseconds. The inefficiencies of the PREV and UPBLK subroutines and the inefficient manner of updating storage utilization integrals were described in detail above. When a storage becomes non-empty, any transactions waiting on a "SNE" delay chain must be reactivated. In GPSS/360, this is accomplished by a generalized subroutine, ECHNGE, while in GPSS/H in-line code within the ENTER block subroutine is employed. The results presented assume no transactions were reactivated.

The most important comparison between the two systems is, of course, the "bottom line," for which the ratio is approximately 5.5 : 1. We rest our case for the ENTER block.

	COUNTS		MICROSECONDS	
FUNCTION	360	H	360	H
ENTER Block Proper	26	20	35	22
Decode Args, Call Subr	24	4	44	6
Calculate Storage Address	12	0	18	0
PREV	9	0	23	0
Update Integral	56	11	115	18
"SNE" Reactivation	8	3	20	3
UPBLK	23	6	51	6
	158	44	306	55

Figure 5 - Comparative Performance of the ENTER Block

Results for the SAVEVALUE Block

The instruction counts and timings for a "SAVEVALUE 1,0" block are shown in figure 6. In GPSS/H, this block is executed completely in-line. The basic sequence of instructions is a load of the address of savevalue 1, a load of the constant zero, and a store into memory. The basic sequence is followed by six instructions required to perform the overhead accounting associated with block completion. The GPSS/360 implementation of this block requires no explanation at this point: the criticisms offered in the foregoing discourse are, almost painfully, applicable.

	COUNTS		MICROSECONDS	
FUNCTION	360	H	360	H
SAVEVALUE Block Proper	12	0	18	0
Decode Args, Call Subr	44	2	68	3
In-line Store	0	1	0	1
Calculate Savex Address	6	0	7	0
PREV	9	1	23	1
UPBLK	23	5	51	5
	94	9	167	10

Figure 6 - Comparative Performance of the SAVEVALUE Block

Results for Miscellaneous Primitive Functions

Figure 7 displays instruction counts and timings for additions/deletions to/from the current events chain, for additions/deletions to/from the future events chain, for dispatching a scan-active transaction encountered on the current events chain, and for scanning the current events chain. For the most part, the results show a decided superiority of GPSS/H over GPSS/360; however, there are two cases, ADDFUT and SCAN CEC, which merit further discussion. Both of these cases involve loops in which a chain of transactions is examined; hence, results are depicted in the form of fixed cost plus a coefficient times the number of transactions examined. In both cases, the ratios between GPSS/360 and GPSS/H coefficients are less than 3:1. This implies that simulations intensive in either of these two activities could fail to meet the 3:1 design goal. In the case of future events chain additions, the algorithm employed in both systems is a linear search, in descending time order, of a linked list. There exist alternative algorithms (See (4) for a discussion.) which could be provided, perhaps upon request of a control card, to achieve improved performance. This will be done when time permits. In the second case, nothing can be done to improve the implementation, given the architecture of the GPSS current events chain. The "load-test-conditional-branch" sequence employed in GPSS/H is optimal. The important point here is that models shouldn't be intensive in the scan of the current events chain. The coefficient for the loop is small enough that many iterations would be required to reach dominance by this loop in total run time. If the current events chain contains such large numbers of scan-inactive transactions, the modeller should, for his own sake, do something about it, such as incorporating user chains into the model.

Results for the One-Line Single-Server Queuing Model

Benchmark runs of the one-line, single-server queuing model shown in figure 8 demonstrated a 4.4 : 1 superiority of GPSS/H over GPSS/360. The times for several runs were averaged to derive the overall ratio. Because of slight difference in the way timing is done in the two systems, the 4.4 figure is conservative. Certain functions performed in the INPUT phase of GPSS/360 are performed at run time in GPSS/H, for example.

	COUNTS		MICROSECONDS	
FUNCTION	360	H	360	H
ADDCUR*	30	7	54	9
SUBCUR*	35	5	75	7
ADDFUT	18+7n	7+3n	36+9n	9+4n
SUBFUT	19	5	53	7
Dispatch Active Xact	19	5	37	11
SCAN CEC	3+9n	2+3n	4+12n	3+5n

* Non-empty priority class

Figure 7 - Comparative Performance for Selected Primitives

```
      SIMULATE
*
*     BARBER SHOP SEGMENT
*
      GENERATE    18,6      ARRIVALS 18 +- 6 MINUTES
      QUEUE       JOEQ      ENQUEUE FOR BARBER
      SEIZE       JOE       ENGAGE BARBER
      DEPART      JOEQ      EXIT THE QUEUE
      ADVANCE     15,3      HAIRCUT 15 +- 3 MINUTES
      RELEASE     JOE       FREE UP THE BARBER
      TERMINATE             DEPART THE SHOP
*
*     TIMER SEGMENT
*
      GENERATE    48000     800 HOURS IN MINUTES
      TERMINATE   1         SHUT DOWN THE SHOP
      START       1
      END
```

Figure 8 - One-Line Single server Queuing Model

CONCLUSIONS

Four-fold enhancements have been achieved in somewhat limited testing of a prototype version of GPSS/H. This demonstration, along with the foregoing analysis, indicates that the design goal of 3:1 is attainable. The ultimate performance in "real" modelling situations is a matter for speculation until the system is completed. Inclusion of such features as secondary storage of entities, a la GPSS/V, will undoubtedly slow down GPSS/H. Remember, however, the comparisons presented herein have all been with GPSS/360, which is presumably somewhat faster than GPSS/V.

As of this writing, availability of the new system is, optimistically, scheduled for September, 1976.

REFERENCES

1. MTS Manual, Volumes 1-5 and 9-13, University of Michigan Computing Center, Ann Arbor, Michigan

2. Thomas J. Schriber, Simulation Using GPSS, John Wiley & Sons, New York, 1974, pp. 393-426

3. David Gries, Compiler Construction for Digital Computers, John Wiley & Sons, New York, 1971

4. Jean G. Vaucher and Pierre Duval, "A Comparison of Simulation Event List Algorithms," Communications of the ACM, April, 1975