2

UCID-16711

This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the laboratory.

LAWRENCE LIVERMORE LABORATORY

University of California/Livermore, California

THE SUITABILITY OF SOFTWARE PHYSICS
AS A CHARGING BASIS FOR THE FLOE SYSTEM

Jeffrey C. Huskamp

March 10, 1975

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Energy Research and Development Administration, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

MASTER

Prepared for U.S. Atomic Energy Commission under contract no. W-7405-Eng-48

DISTRIBUTION OF THIS DOCUMENT UNLIMITED

EB

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Abstract

The concept of software physics is investigated as a possible charging basis for the CDC 7600 FLCE operating system. The charging equations for cpu usage and disk usage are shown to be inadequate for expressing program charging as a function of program efficiency. To remedy this, alternate equations are proposed. The charging equations for tape usage, however, are shown to be sufficient. The last section of this paper contrasts the relative efficiencies of the CHAT optimized, the PUTT, and the CHAT unoptimized compilers, which are available on the FLCE system, on a sample of fourteen program kernels.

Introduction

The Concept of Software Physics

The concept of software physics was developed by Ken Kolence as the basis for a unified theory for computer resource charging. Software physics consists of a set of measures derived from the definition of software work. From the definition of work, corresponding definitions of software power (work/second) and software intensity (work/byte-second) can be formalized. The major advantage of software physics is that job charges are repeatable, that any program performance improvement will yield a lower job cost, and that the parameters of the charging equation are easily obtainable. To date, these ideas have been applied almost exclusively to business data processing installations which use highly structured operating systems like OS/360 or OS/370. One question which needs to be answered is whether the same concepts can be applied to a relatively unstructured operating system (such as FLOE) and to a different machine organization (such as the CDC 7600 at Lawrence Livermore Laboratory).

The definition of software work given by Kolence in [4] and [5] is:

"A processor performs one unit of software work on some storage media when one byte of that media is altered."

Work is the unit in which resource capacity usage is expressed. Capacity usage for a resource is the sum of the direct work performed (i.e., number of bytes altered) plus the usage of other resources involved (such as number of words in a disk file or the length of a memory buffer). When the actual value of work performed by a program on a resource cannot be determined, such as for the cpu, an average value of work is computed by multiplying the average power ($\bar{P}$) of the resource (in units of work/second) times the number of seconds the

program used the resource. In other cases, such as for disk files whose usage is measured in units of (number of bytes occupied)x(number of seconds occupied), the work equivalent is determined by multiplying the value of intensity (I) for the resource (work/byte-second or power/byte) times the usage. Once the work equivalent of all the resources used by a program is computed, the job cost (not including overhead costs) is determined by:

$$\text{Cost} = \sum_{\substack{\text{resources} \\ \text{used}}} (\text{Capacity usage of the resource}) \times (\text{Rate for the resource})$$

In the following discussions, only capacity usage will be considered since the rate determination is beyond the scope of this report.

This report will consider the capacity usage equations for the central processor, disk I/O and tape I/O since these are the usual devices used by programs at Lawrence Livermore Laboratory (LLL). The capacity equations (where C is the capacity usage) derived by Kolence for these devices are:

$$\begin{aligned} C(\text{cpu}) = & \bar{P}(\text{cpu}) \times (\# \text{ cpu seconds}) \\ & + \frac{\bar{P}(\text{cpu})}{\text{total } \# \text{ bytes in main storage}} \times (\# \text{ cpu seconds}) \\ & \times (\# \text{ main storage words used}) \end{aligned}$$

$$\begin{aligned} C(\text{disk}) = & (\# \text{ disk accesses}) \times (\text{average block length}) \\ & + \bar{P}(\text{disk}) \times (\# \text{ disk seconds}) \times \frac{(\# \text{ bytes in disk file})}{(\# \text{ bytes on entire disk})} \\ & + \bar{P}(\text{channel}) \times (\# \text{ channel seconds}) \times \frac{(\text{average block length})}{(\text{total } \# \text{ bytes main storage})} \end{aligned}$$

$$\begin{aligned}
 C(\text{tape}) = & (\# \text{ tape accesses}) \times (\text{average block length}) \\
 & + \bar{P}(\text{tape}) \times \frac{(\text{total tape used})}{\bar{P}(\text{tape})_{\text{rated}} \times (\# \text{ tape seconds})} \times (\# \text{ tape seconds}) \\
 & + \bar{P}(\text{channel}) \times (\# \text{ channel seconds}) \times \frac{(\text{average block length})}{(\text{total } \# \text{ bytes in main storage})}
 \end{aligned}$$

where $\bar{P}(\text{disk}) = (\text{average block length}) / (\text{average time to read a block})$

$\bar{P}(\text{channel}) = (\text{average } \# \text{ bytes transmitted on channel}) / (\text{average execution time of channel})$

$\bar{P}(\text{tape}) = (\text{average actual observed bytes transmitted to or from the tape drive per unit of execution time})$

$\bar{P}(\text{tape})_{\text{rated}} = \text{manufacturers rated speed for the tape drive.}$

Clarification of the above formulas can be found in [4].

Architecture of the CDC 7600

A simplified model of the CDC 7600 is shown in Figure 1. Like the CDC 6000 series of machines, all storage references are requested by loading the address of the location to be fetched or stored into the appropriate A register and the correspondingly numbered X register will either receive the memory contents or contains the word to be stored. The registers A1 through A5 (and hence X1 to X5) are used for fetching and A6 through A7 (and hence X6 through X7) are used for storing. The B registers are used for indexing.

The main differences between the CDC 7600 and the CDC 6000 series are the increased size of the instruction stack and the two level memory hierarchy of small, fast core memory (SCM) and large, slow core memory (LCM). The instruction stack is capable of holding a ten word loop so no memory references are required for instruction fetching for small loops. In addition, loop termi-

nating branch instructions take two cycles instead of eleven for branches to words that are not in the stack.

The speed of the central processing unit is a result of overlapping instruction execution times and of the instruction look-ahead stack. The overlapping instruction execution is also available on the CDC 6600 but not on the CDC 6400 or CDC 6500. On the CDC 7600, instructions may be issued at a maximum rate of one per cycle (one cycle = 27.5 nanoseconds). Instruction issuance may be delayed if the register(s) which contain input for the instruction is busy - i.e., the register containing an input operand for the current instruction is the output register for a previous instruction which has not yet been completed. In Figure 2B, the IX5 instruction which is issued at cycle 10, could not be issued at cycles 3 through 9 because registers X1 and X2 contain input operands and are not free. Instruction execution time is dependent on whether the target address is in the instruction stack for branch instructions and on the completion time of prior instructions (i.e., two instructions cannot deliver results to two X registers, two A registers or two B registers during the same cycle since only one input bus exists for each set of registers). For example, in Figure 2A, the IX5 instruction issued at cycle 10 normally executes in two cycles, but in this instance requires three since a previously issued instruction (SA4 issued at cycle 4) is using the X register input bus at cycle 11. More detailed instruction execution rules may be found in Appendix A of [1].

Description of the FLOE System

The FLOE system is an internally written time-sharing operating system for the CDC 7600. FLOE is a basic system which provides resource sharing and inter-

process communication. From this basic system the users may implement any interface required to meet their specific application with little overhead incurred from unwanted operating system features. For a complete description of the FLCE system, the set of Livermore Time-Sharing System (LTSS) manuals may be requested from the Technical Information Department at LLL.

A user process involved at a terminal enters the system at a priority determined by the user. When the user's process is selected to run, the entire program is loaded into LCM and waits for SCM space to become available (instructions can only be executed from SCM). When SCM space is available, the SCM portion of the program is copied into SCM and the program is executed. Therefore, during execution, the SCM portion of the program resides in both SCM and LCM while the LCM portion only resides in LCM. This fact is used when capacity usage is calculated.

Goals of this Report

This report is divided into two sections. The first section explores the applicability of Kolence's charging equations to the FLCE system on the CDC 7600. In particular, efficiency considerations peculiar to the FLCE system, which should affect the program charges are studied and alternate equations will be advanced when needed. This section will attempt to determine whether software physics can be successfully applied to a scientific installation with a relatively unstructured operating system.

The second part of this report will be a comparison of compilers in use at the Laboratory. The CHAT compiler implements the LRLTRAN languages, an extension

of FORTRAN [6]. For this study, both the optimized and unoptimized object code of this compiler were analyzed. The PUTT compiler is the standard CDC FORTRAN compiler for the CDC 7600 which was modified to run under the FLCE system. The analysis done on each of these compilers includes a comparison of the number of instructions executed, the number of cycles used, the number of floating point operations produced per second and the charge for execution (based on cpu seconds) of each of fourteen production kernels.

Program Kernel Evaluation Method

Obtaining the Program Kernels

In order to obtain a sample of the calculations performed by the largest users of the computer resources at LLL, fourteen program kernels were identified as representative of the various types of production programs currently in use. These kernels appeared in a recently compiled invitation for bids for the new CTR computer center to be located at LLL. The kernels are shown in Tables 1 through 14.

Data Collection Procedure and Validation

All fourteen kernels were compiled by each of the compilers used in the test, and instruction timing diagrams, like the one in Figure 3, were constructed. Using the diagrams, the calculated number of cycles used, the number of non-PASS instructions executed, the number of PASS instructions executed, the number of 60-bit instruction words in the loop, the number of data words used and the number of fetches and stores from memory (for work calculation) could be mechanically determined. Using the above numbers, the number of floating point operations per second was computed for each loop. This measure of efficiency is widely used at the Laboratory since floating point operations

indicate results are being produced while fixed point operations are linked to program indexing which represent values that are not results important to the users of the code. In order to validate the calculated figures, each loop was executed and timed to obtain the number of floating point operations per second using the internal microsecond clock.

Related efficiency measures that can be derived from the figures produced are the instruction issuance efficiency (number of non-PASS instructions issued/ number of cycles used), Kolence's approximation to the cpu capacity usage and the actual cpu capacity usage. Kolence's approximation to the cpu capacity usage was calculated by the formula:

$$C = \bar{P} \times (\text{cpu seconds}) + \left[\frac{\bar{P}}{(\text{total SCM storage})} \times (\text{cpu seconds}) \times (\text{program length}) \right] + \left[\frac{\bar{P}}{(\text{total LCM storage})} \times (\text{cpu seconds}) \times (\text{program length}) \right]$$

where $\bar{P}$ = the average work of the entire workload. This was calculated by the equation:

$$\bar{P} = \frac{\text{work}}{\text{cpu seconds}} = \frac{(\text{fetches+stores})}{\text{cpu seconds}}$$

where the summation was over the fourteen kernels compiled by each of the three compilers (42 values were summed). The value calculated was $P = 4.922 \times 10^6$.

total SCM storage = the total size of SCM storage. The value 65,536 was chosen to represent the CDC 7600 configuration for a large scientific installation.

total LCM storage = the total size of LCM storage. The value 400,000 words was chosen to represent the CDC 7600 configuration for a large scientific installation.

program length = the number of instruction words in the loop plus the number of data words used.

The actual cpu capacity usage was calculated from the formula:

$$C = W + \left[\frac{\bar{P}}{\text{total SCM storage}} \times (\text{cpu seconds}) \times (\text{program length}) \right] + \left[\frac{\bar{P}}{\text{total LCM storage}} \times (\text{cpu seconds}) \times (\text{program length}) \right]$$

where W = the actual cpu work performed (i.e. the number of fetches and stores executed).

$\bar{P}$, total SCM storage, total LCM storage, program length = as defined above.

Note that for the capacity equation, when the size of main storage becomes large, the third term becomes negligible. For the 42 cases calculated, the third term never exceeded 2.5% of the total and was usually less than one percent.

The listing of the kernels and the calculated values for each of the categories listed above appear in Tables 1 through 14. The validation of the figures consisted of the comparison of the calculated number of floating point operations per second with the measured value. Most of the figures compare within 5%. This discrepancy can be explained by the measured value including loop

set-up and a small error associated with issuing system call to obtain the clock reading. Some of the discrepancies are on the order of 10-15% (for example, the CHAT unoptimized results in Table 1). These discrepancies were traced to memory bank conflicts which are not accounted for in the timing diagrams. For example, in the CHAT unoptimized example in Table 1, seven cycles are lost per loop execution due to a memory bank conflict. When the lost cycles are added to the calculated number of cycles used, the calculated floating point operations per second is lowered to 2.635×10^6 . The percentage difference is then 2.23%. It is felt that memory conflicts should be neglected in the compiler comparison since placement of the variables in memory is usually dependent on the arrangement of the declaration statements and the variables within the declaration statements in the source code, and does not reflect on the efficiency of the object code produced. Therefore, the calculated value of the number cycles consumed was used in lieu of the measured value in all calculations.

Modifications to the Software Work Charging Algorithm for the CDC 7600 FLCE System

The objective of an equitable charging algorithm is to provide lower charges for programs which run more efficiently. This objective was a main concern to Kolence when he drafted his charging formulas. However, the definition of efficiency may not be the same for highly structured operating systems (such as CS/370) and for more basic operating systems (such as FLOE). This is particularly true in the area of input/output where the peripheral devices are masked by accessing routines in CS/370 (BDAM, ISAM, etc.) unlike in FLOE. For a more complete idea of the capabilities of the FLOE system, the reader is referred

to [2]. The following sections will analyze the capacity equations for the cpu, disk and tape, and additions are suggested for the FLOE system in order to charge more efficient programs less.

CPU Capacity Usage

In the CDC 7600, cpu program efficiency is a function of the following five attributes:

- (1) Instruction ordering
- (2) Stackloops - i.e., whether a program loop fits entirely in the instruction stack so no memory references are needed to fetch instruction words.
- (3) Instruction packing - i.e., the number of PASS instructions executed.
- (4) SCM memory usage
- (5) LCM memory usage

Improvements in each of these areas can be directly measured in terms of the number of machine cycles used.

To demonstrate the effect of instruction ordering on execution time, the two program segments in Figures 2A and 2B were written to implement the FORTRAN assignment statement $A = B + C + D + E$ using the same eight instructions. The only difference in the two programs is the ordering of the instructions. The more efficient program (Figure 2A) executes in 17 cycles while the less efficient segment executes in 25 cycles (a 47% slower loop).

Stackloops speed up loop execution by executing the loop terminating branch instructions in 3 cycles rather than 11 cycles for a branch out of the instruction stack. The instruction stack is capable of holding loops less than or equal to 10 instruction words in length. As an example of the effect of the instruction stack, suppose the PUTT compiler object code for kernel 3 shown in Table 3 was 11 instruction words long (non-stackloop) rather than 10 (stackloop). The 11 instruction work loop would execute 37.5% slower than the smaller loop, due to the increased execution time of the loop terminating branch instruction.

Instruction packing is a measure of the potential number of cycles during which an instruction could not be issued because a 30-bit instruction must be executed next when only 15 bits remain in the present instruction word. The remaining 15 bits are filled with a PASS instruction and the thirty bit instruction starts in the next instruction word. The PASS instruction executes in one cycle.

SCM and LCM efficiency is related to the type of processing to be done and on the placement of operands to minimize bank conflicts. For example, SCM is best suited for random accessing (8 cycles/fetch) while LCM is best suited for sequential accessing (3 or 17 cycles/fetch). A complete discussion of the role of SCM and LCM is considered to be beyond the scope of this report. A more complete explanation can be found in [3].

Each of the first three efficiency measures listed at the beginning of this section is directly related to the amount of cpu time used. The more cpu time

consumed, the less efficient the program for implementation of the same problem. In Kolence's charging formulas, the theoretical capacity for the CDC 7600 is computed by:

$$\text{Capacity} = \text{Work} + \text{SCM usage} + \text{LCM usage}$$

where Work is the number of fetches and stores performed by the cpu. Kolence's final charging formula approximates the amount of work done by a program by multiplying the average power of the processor by the number of cpu seconds used. From the above observations on cpu efficiency, the amount of actual work performed by a program (number of fetches and stores) should not necessarily be related to program efficiency. For the kernels investigated in this report (Tables 1 - 14), kernels 1, 4, 5, 6, 7, 8 and 14 charged more for implementation that took fewer cpu cycles. In the remaining kernels, the ordering of the charges was correct, but the charge was not always directly related to the number of cpu cycles used. For example, the program loop which had the correct charge ordering in Table 2, the PUTT compiler implementation of the program loop uses only 2.8% more cpu cycles than the CHAT optimized implementation, but is charged 19.4% more based on the amount of real work performed. Thus the concept of work and power, for charging purposes at least, does not reflect real resource usage. For this reason, the use of cpu seconds as a charging basis for the central processor in the CDC 7600 is recommended.

Disk Capacity Usage

The major contribution to the disk capacity usage is the disk drive work (= number of disk accesses X average block length) since the other two terms are divided by large numbers (number of words in a disk pack in one case and

the total number of words in main storage in the other). On the FLCE system, the disk usage efficiency is related to the I/C strategy used by the user program rather than to the number of disk accesses performed. Once a disk file has been opened, all I/C is requested by specifying the first word address on the disk to be read/written and the number of words to be transferred. A central memory buffer is also specified to be the source or sink for the words to be transferred.

For the CDC 817 and CDC 844 disks in use on the CDC 7600, each track is divided into blocks of 512 words and 64 words, respectively, which are called sectors. The first word disk address of sectors in a disk file are 0, ..., $n*512$ (or $n*64$). In the following discussion, let S be the number of words in a sector and n be a positive integer. Since only full sectors can be read or written by the hardware, the most important measure of disk usage efficiency is the number of passes over the data needed to complete a user's disk request. There are four possible cases. For the first three cases (a disk read starting on a sector boundary with $n*S$ words to be read, a disk read starting anywhere for an arbitrary number of words, and a disk write starting on a sector boundary with $n*S$ words to be written), only one pass over the data on disk is needed to complete the disk request. However, for the fourth case of a disk write request which does not start on a sector boundary, either two or three passes over the data are needed. This is due to the initial and/or final sector containing the data to be replaced having to be read, the data patched in, then written back to the disk. If just the first sector or just the last sector needs to have data patched in, two passes over the data are needed. If both the first and the last sector need

to have data patched in, three passes over the data are necessary (provided the first sector is not the same as the last sector in which case only two passes are necessary).

Tables 15 and 16 contain experimental timings for both the CDC 817 (slow access, fast transmission) and the CDC 844 (fast access, slow transmission) disks. A disk file of 100,000 words was written using disk writes of length 1000, 2000, 4000 and 10,000 words. The timings in both tables are the average of ten trials, where each timing trial is given by the expression $0.5 \times (\text{actual number of channel seconds needed to complete the request})$. The trials are not repeatable since disk seek time is included in the timings. The data can be used to show trends rather than absolute numbers. As discussed above, the sector boundary reads and the non-sector boundary reads have timings which are very close to each other. The largest difference is between the timings for the sector boundary write and the non-sector boundary write. The non-sector boundary write used in the timings required both the first and last sectors to be patched with new data. Therefore the disk timings for non-sector boundary writes should require about three times the amount of channel time needed for sector boundary writes.

From the above discussion, the disk charging algorithm should take into account the number of passes over the data on disk. The number of passes can be easily computed by the following steps:

- 1) If a disk read is requested, set the number of passes to one and skip the remaining steps.
- 2) Set the number of passes to one.

- 3) If the first word disk address is not on a sector boundary, add one to the number of passes.
- 4) If the last word to be written is not on a sector boundary, add one to the number of passes.
- 5) If the number of passes is three and the first word disk address and the last word to be written are in the same sector, set the number of passes to two.

From the above discussion, it is suggested that the charging algorithm should be based on the number of passes over the data on disk and the number of words transferred. A proposed capacity usage equation for disk drive and channel work is:

$$C(\text{disk}) = k_1 \times (\# \text{ passes over the data}) + k_2 \times (\# \text{ of words transferred})$$

The above capacity equation is repeatable and the constants can be adjusted to provide the proper penalty for inefficient disk usage. In addition, the value of $\bar{P}(\text{disk})$ does not need to be determined. The pack storage usage charge provided by Kolence may be added to the equation to charge for disk storage usage. If a more highly structured operating system was studied (like CS/370) where the actual disk handling is covered by many layers of the operating system, the capacity equation derived by Kolence would probably be used rather than the one suggested here.

Tape Capacity Usage

The only special feature available on the FLCE system for tape processing is the ability to read or write multiple records with one request. In

Table 17, the charge time and real time for writing 200,000₈ words on a tape in 1000₈ word records are shown. The tape was written requesting one, two, four and eight records to be written with one request. The charge time shown in Table 17 is computed from the expression $0.5x(\text{actual channel time used to read or write the tape})$. The number of cycles used is shown to be independent of the number of records read/written with one request. However, the real time needed to process the tape requests decreases as the number of records read/written with one request increases. Therefore the only advantage of reading/writing multiple records is for speeding up real time tape bound jobs.

Since the amount of channel busy time is the same regardless of the tape I/O strategy (hence the capacity usage is the same), the tape usage is proportional only to the number of records read/written. Therefore Kolence's equations for capacity usage are directly related to the tape drive efficiency on the FLCE system. No further modifications appear to be necessary for computing capacity usage.

Comparison of Three Production Compilers

Efficiency Considerations

As was stated in the Introduction of this report, the three compilers analyzed were the CHAT optimized, the CHAT unoptimized and the PUTT compilers which are available to all users at LLL. None of the compilers was able to equal the efficiency of a hand-coded program for the reasons given later in this section. For example, the timing diagram of the first kernel for the

most efficient compilation (CHAT optimized) is displayed in Figure 3. This should be contrasted with the hand-coded version displayed in Figure 4 which was coded in 20 minutes by the author. Both programs fit in the instruction stack and execute with no instruction fetch delays. The CHAT compiled version executes 76.7% slower than the hand-coded version. This example provides a rough measure of possible improvement for the kernels measured. An estimate based on experience would be that improvements up to a factor of two are possible by hand-coding.

Most of this advantage comes from the fact that a programmer can carefully plan the loop execution in order to keep in registers the variables fetched or overstored in previous loop executions, to aid in the present execution of the loop. For example, in Table 5, the values of $Z(I)$, $Y(I)$, $X(I-1)$, $Z(I+1)$ and $Y(I+1)$ should be saved to aid in calculation of future passes through the loop. With this method, only $Z(I+2)$ and $Y(I+2)$ need to be fetched each pass through the loop. The CHAT optimized compiler makes seven fetches per loop execution, PUTT makes seven fetches, and CHAT unoptimized ten fetches (including one fetch for the loop index, I).

The CHAT optimized and PUTT compilers perform extensive prefetching of constants and indexed variables that are used in the loop while the CHAT unoptimized does not. For example, in Table 1, the CHAT optimized and PUTT compilers prefetched K , I , R , T and Q and thus only needed three operand fetches per loop execution. The CHAT unoptimized compiler, however, fetched K , I , R , T , Q and the three operands fetched by the other two compilers every loop execution. This is the major reason that the CHAT unoptimized object code executed more slowly.

Another difference is that the CHAT optimized compiler does most of the fetches needed for the loop execution at the beginning of the loop. Both the CHAT optimized and the CHAT unoptimized compilers appear to do operand fetches in blocks rather than singly. This has the effect of minimizing lost cycles waiting for operands from memory. For example, instead of issuing three memory fetches at widely separated times when the fetches will take 8 cycles apiece to complete, three instruction fetches are issued in three successive cycles so all three operands will be available after 10 cycles. The PUTT compiler performs operand fetching more on an 'as needed' basis. This type of fetching occurred in Table 5 where the CHAT optimized compiler performed two blocks of fetches, one three fetches long and the other four fetches long, and caused 13 cycles to be wasted per loop execution. The PUTT compiler, on the other hand, issued six fetches (only two of which were in adjacent cycles) and caused 39 cycles to be wasted per loop execution. Note that a cycle is wasted if an instruction can not be issued during the cycle because the cpu is waiting for operands to be fetched from memory.

The above paragraphs describe the major areas of inefficiencies found in the compiled kernels. The following are miscellaneous observations on the compilers:

- 1) None of the three compilers made extensive use of the B registers in the CDC 7600. The major reason for this is that the B registers are only 18 bits long. Common values, such as some LCM addresses, are too large for this register. The compiler must know whether the value to be placed in the register will ever overflow before a B register can be allocated.

- 2) Most of the cycles used in the kernels are waiting for input - either operands from memory or register results. For example in Table 5 for the CHAT optimized compiler, 33.9% of the cycles were used to initiate instructions and 63.1% of the cycles were waiting for input. Therefore, the kernels all appear to be X register limited.
- 3) The PUTT compiler performed some unnecessary register copying when a previously computed variable could be used in a subsequent computation (common subexpression matching).
- 4) The CHAT unoptimized compiler used the 'jump on X register positive' and 'jump on X register zero' instructions to terminate loops rather than the one instruction 'jump on register B₁ greater than or equal to B₂.'

This section could not be concluded without mentioning three areas of efficient object coding that were covered by all three compilers. The compilers recognized common subexpressions, performed at least some instruction scheduling and performed some form of instruction packing (minimized the number of PASS instructions executed). The above three areas are very important for the CDC 7600.

Compiler Comparison Summary

The total number of cycles used for all kernels, the total number of non-PASS instructions executed, the total number of PASS instructions executed and the total number of fetches and stores executed are tabulated for each compiler in Table 18. From this summary and the above discussion on compiler efficiency, it is concluded that the ordering of the compilers for the fourteen kernel

sample analyzed is (from best to worst) CHAT optimized, PUTT and CHAT unoptimized.

Conclusion

The first part of this report investigated efficiency considerations which lead to improved system performance but were not represented in Kolence's charging equations. In particular, cpu efficiency was shown to be related to the number of cpu cycles used rather than the concept of work. In fact, a savings could be realized by not having to determine the value of the average power for the cpu from a hardware monitor. For direct access devices (disk), efficiency was shown to be related to the number of passes over the data needed to complete the I/O request and was not represented in the charging formula. A formula was advanced which made the disk and channel work components of the capacity usage a function of the number of passes over the data and the number of words transmitted. The tape capacity usage equation was shown to be directly related to efficiency considerations on the FLCE system and no modification was needed. Thus from the above results it is concluded that Kolence's ideas of a charging algorithm related to machine efficiency is machine and operating system dependent. In the last section of this report, the object code of the three production compilers was contrasted and the order of the compilers (from best to worst) was found to be the CHAT optimized, the PUTT, and the CHAT unoptimized compiler.

FIGURE 1
Architecture of the CDC 7600

for the Equation $A = B + C + D + E$

(B)

Cycle	Instruction	A-registers								X-registers								B-registers							
		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
21									X																
22									X																
23	FX2 X5+X0											X													
24												X													
25												X													
26												X													
27	NX5 B0,X2														X										
28	SA2 B2+B7				X							X			X										
29					X							X			X										
30												X													
31												X													
32												X													
33												X													
34												X													
35												X													
36	FX0 X2*X5								X																
37									X																
38									X																
39									X																
40									X																
41	FX2 X1+X0											X													
42												X													

Cycle	Instruction	A-registers								X-registers								B-registers							
		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
43											X														
44											X														
45	NX6 B0,X2															X									
46	PASS															X									
47																X									
48	SA6 B2,X							X								X									
49	SB2 B2+B3							X								X				X					
50	PASS																			X					
51	JBGE B1,B2,L																								
52	(JUMP)																								
53	(JUMP)																								

FIGURE 4
Hand-Coded Loop for Kernel 1

Cycle	Instruction	A-registers								X-registers								B-registers							
		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	FX6 X3*X2															X									
2	SX3 X4											X				X									
3	FX7 X4*X1											X				X X									
4	SB2 B2+B1															X X						X			
5	SA4 A4+B1						X							X	X X							X			
6						X								X	X										
7														X	X										
8	FX7 X7+X6													X	X										
9														X	X										
10														X	X										
11														X	X										
12	NX7 X7,B0													X	X										
13																X									
14																X									
15	FX7 X7*X5															X									
16																X									
17	SA5 A5+B1						X								X	X									
18							X								X	X									
19															X	X									
20	FX7 X7+X0														X	X									
21															X	X									

Cycle	Instruction	A-registers								X-registers								B-registers							
		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
22															X		X								
23															X		X								
24	NX7 X7,B0														X		X								
25																	X								
26																	X								
27	SA7 A7+B1							X																	
28	LT B2,B3,LOCP							X																	
29	(JUMP)																								
30	(JUMP)																								

TABLE 1
Hydro Code Excerpt (Kernel 1)

```

      Q = 0.0
      I = 10
      DO 1 K= 1,400
1      X(K) = Q+Y(K)*(R*Z(K+1)+T*Z(K+1+1))

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	21199	5600	400	16	1206	1600
FUTT	21199	7200	400	17	1206	2800
CHAT unoptimized	24803	10000	400	13	1206	4000
	Calculated number of floating point operations per second (x10 ⁶)	Measured number of floating point operations per second (x10 ⁶)	Instruction issuance efficiency (%)	Actual capacity (x10 ³)	Kolence capacity (x10 ³)	Calculated power (x10 ⁶)
CHAT optimized	3.431	3.334	26.4	1.663	2.932	2.745
FUTT	3.306	3.214	32.7	2.863	2.932	4.628
CHAT unoptimized	2.932	2.576	40.3	4.072	3.423	5.864

TABLE 2
MLR Inner Product (Kernel 2)

```

Q = 0.0
DO 2 K=1,996,5
2   Q=Q +Z(K  )*X(K  ) +Z(K+1)*X(K+1)
.     +Z(K+2)*X(K+2) +Z(K+3)*X(K+3)
.     +Z(K+4)*X(K+4)

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	14199	5400	200	17	2002	2000
FUTT	14599	5800	200	18	2002	2400
CHAT unoptimized	19595	7000	1400	17	2002	2800
	Calculated number of floating point operations per second (x10 ⁶)	Measured number of floating point operations per second (x10 ⁶)	Instruction issuance efficiency (%)	Actual capacity (x10 ³)	Kolence capacity (x10 ³)	Calculated power (x10 ⁶)
CHAT optimized	5.122	5.060	38.0	2.069	1.991	5.122
FUTT	4.982	4.864	39.7	2.471	2.047	5.978
CHAT unoptimized	3.712	3.380	35.7	2.895	2.748	5.196

TABLE 3
Inner Product (Kernel 3)

Q = 0.0
DO 3 K=1,1000
3 Q=Q +Z(K)*X(K)

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	21999	7000	0	9	2002	2000
PUTT	23999	7000	0	10	2002	2000
CHAT unoptimized	37003	15000	2000	8	2002	6000
	Calculated number of floating point operations per second ($\times 10^6$)	Measured number of floating point operations per second ($\times 10^6$)	Instruction issuance efficiency (%)	Actual capacity ($\times 10^3$)	Kolerance capacity ($\times 10^3$)	Calculated power ($\times 10^6$)
CHAT optimized	3.306	3.220	31.8	2.106	3.084	3.306
PUTT	3.030	2.805	29.2	2.116	3.365	3.030
CHAT unoptimized	1.965	1.556	40.5	6.179	5.187	5.896

TABLE 4
Banded Linear Equations (Kernel 4)

```

DO 175 L = 7,107,50
  LW=L
  DO 4 J = 30,870,5
    X(L-1)=X(L-1)-X(LW)*Y(J)
  4 LW=LW+1
    X(L-1)=Y(5)*X(L-1)
175 CONTINUE

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	14958	5163	21	18	971	2079
PUTT	15465	6168	15	20	971	2049
CHAT unoptimized	31642	12225	1026	21	971	5604

	Calculated number of floating point operations per second ($\times 10^6$)	Measured number of floating point operations per second ($\times 10^6$)	Instruction issuance efficiency (%)	Actual capacity ($\times 10^3$)	Kolence capacity ($\times 10^3$)	Calculated power ($\times 10^6$)
CHAT optimized	2.480	2.426	34.5	2.115	2.061	5.054
PUTT	2.398	2.346	39.9	2.086	2.130	4.818
CHAT unoptimized	1.172	1.116	38.6	5.680	4.359	6.440

TABLE 5
Tri-Diagonal Elimination below Diagonal (Kernel 5)

```

DO 5 I = 2,997,3
  X(I ) = Z(I )*(Y(I )-X(I-1))
  X(I+1) = Z(I+1)*(Y(I+1)-X(I ))
5    X(I+2) = Z(I+2)*(Y(I+2)-X(I+1))

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	20583	6972	664	14	2997	3320
FUTT	27555	7636	332	15	2997	2988
CHAT unoptimized	29875	9960	1328	14	2997	4648
	Calculated number of floating point operations per second (x10 ⁶)	Measured number of floating point operations per second (x10 ⁶)	Instruction issuance efficiency (%)	Actual capacity (x10 ³)	Kolence capacity (x10 ³)	Calculated power (x10 ⁶)
CHAT optimized	3.533	3.294	33.9	3.469	2.935	5.865
FUTT	2.639	2.543	27.7	3.187	3.929	3.943
CHAT unoptimized	2.434	2.118	33.3	4.864	4.260	5.658

TABLE 6
Tri-Diagonal Elimination above Diagonal (Kernel 6)

```

DO 6 J = 3,997,3
  I = 1000-J+3
    X(I ) = X(I ) -Z(I )*X(I+1)
    X(I -1) = X(I-1) -Z(I-1)*X(I )
6    X(I -2) = X(I-2) -Z(I-2)*X(I-1)

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	22907	8300	332	14	1994	3320
FUTT	26227	9296	664	17	1994	2988
CHAT unoptimized	32863	11620	996	15	1994	5312
	Calculated number of floating point operations per second (x10 ⁶)	Measured number of floating point operations per second (x10 ⁶)	Instruction issuance efficiency (%)	Actual capacity (x10 ³)	Kolence capacity (x10 ³)	Calculated power (x10 ⁶)
CHAT optimized	3.175	3.101	36.2	3.431	3.211	5.270
FUTT	2.773	2.766	35.4	3.115	3.667	4.143
CHAT unoptimized	2.213	1.789	35.4	5.470	4.607	5.878

TABLE 7
Equation of State Excerpt (Kernel 7)

```

DO 7 M=1,120
X(M)=      U(M ) + R*( Z(M ) + R*Y(M ))
.      +T*( U(M+3) + R*( U(M+2) + R*U(M+1))
.      +T*( U(M+6) + R*( U(M+5) + R*U(M+4)))
7 CONTINUE

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	13671	4560	360	19	488	1440
PUTT	16671	5520	240	22	488	1680
CHAT unoptimized	15717	5400	120	18	488	1800
	Calculated number of floating point operations per second ($\times 10^6$)	Measured number of floating point operations per second ($\times 10^6$)	Instruction issuance efficiency (%)	Actual capacity ($\times 10^3$)	Kolence capacity ($\times 10^3$)	Calculated power ($\times 10^6$)
CHAT optimized	5.107	4.903	33.4	1.456	1.867	3.830
PUTT	4.188	3.824	33.1	1.701	2.277	3.665
CHAT unoptimized	4.442	4.309	34.4	1.819	2.146	4.165

TABLE 8
Partial Differential Equation Integration (Kernel 8)

```

NL1 = 1
NL2 = 2
DO 90 KX = 2,3
DO 90 KY = 2,21
DU1 = U1(KX,KY+1,NL1) - U1(KX,KY-1,NL1)
DU2 = U2(KX,KY+1,NL1) - U2(KX,KY-1,NL1)
DU3 = U3(KX,KY+1,NL1) - U3(KX,KY-1,NL1)
U1(KX,KY,NL2) = U1(KX,KY,NL1) +A11*DU1+A12*DU2 + A13*DU3 +
1 SIG*(U1(KX+1,KY,NL1)-2.*U1(KX,KY,NL1)+U1(KX-1,KY,NL1))
U2(KX,KY,NL2) = U2(KX,KY,NL1) +A21*DU1+A22*DU2 + A23*DU3 +
1 SIG*(U2(KX+1,KY,NL1)-2.*U2(KX,KY,NL1)+U2(KX-1,KY,NL1))
U3(KX,KY,NL2) = U3(KX,KY,NL1) +A31*DU1+A32*DU2 + A33*DU3 +
1 SIG*(U3(KX+1,KY,NL1)-2.*U3(KX,KY,NL1)+U3(KX-1,KY,NL1))
90 CONTINUE

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	10659	4064	330	54	678	1376
FUTT	12191	4438	202	57	678	1730
CHAT unoptimized	14448	4658	400	53	678	1606

	Calculated number of floating point operations per second (x10 ⁶)	Measured number of floating point operations per second (x10 ⁶)	Instruction issuance efficiency (%)	Actual capacity (x10 ³)	tolence capacity (x10 ³)	Calculated power (x10 ⁶)
CHAT optimized	4.913	4.538	38.1	1.395	1.461	4.694
FUTT	4.295	3.991	36.4	1.752	1.672	5.160
CHAT unoptimized	3.624	3.488	32.2	1.632	1.978	4.042

TABLE 9
Integration Predictors (Kernel 9)

```

DO 9 I=1,100
  PX( I,I)= BM28*PX(13,I) + BM27*PX(12,I) + BM26*PX(11,I) +
    .       BM25*PX(10,I) + BM24*PX(9,I) + BM23*PX(8,I) +
    .       BM22*PX(7,I) + C0*(PX(5,I) +      PX(6,I))+ PX( 3,I)
9 CONTINUE

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	11899	4800	400	24	1309	1900
PUTT	24891	8400	1100	58	1309	3600
CHAT unoptimized	14309	5600	1000	25	1309	2200

	Calculated number of floating point operations per second ($\times 10^6$)	Measured number of floating point operations per second ($\times 10^6$)	Instruction issuance efficiency (%)	Actual capacity ($\times 10^3$)	Kolence capacity ($\times 10^3$)	Calculated power ($\times 10^6$)
CHAT optimized	5.195	4.743	40.3	1.938	1.649	5.806
PUTT	2.484	2.319	33.7	3.682	3.451	5.259
CHAT unoptimized	4.320	3.962	39.1	2.245	1.982	5.591

TABLE 10
Difference Predictors (Kernel 10)

```

DO 10 I=1,100
AR      =      CX(5,I)
BR      =  AR - PX(5,I)
PX(5,I) =  AR
CR      =  BR - PX(6,I)
PX(6,I) =  BR
AR      =  CR - PX(7,I)
PX(7,I) =  CR
BR      =  AR - PX(8,I)
PX(8,I) =  AR
CR      =  BR - PX(9,I)
PX(9,I) =  BR
AR      =  CR - PX(10,I)
PX(10,I) = CR
BR      =  AR - PX(11,I)
PX(11,I) = AR
CR      =  BR - PX(12,I)
PX(12,I) = BR
PX(14,I) = CR - PX(13,I)
PX(13,I) = CR
10 CONTINUE

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	16491	5300	200	26	1404	2500
PUTT	31591	9900	800	58	1404	5200
CHAT unoptimized	18197	6000	300	23	1404	2600

	Calculated number of floating point operations per second ($\times 10^6$)	Measured number of floating point operations per second ($\times 10^6$)	Instruction issuance efficiency (%)	Actual capacity ($\times 10^3$)	Reliance capacity ($\times 10^3$)	Calculated power ($\times 10^6$)
CHAT optimized	1.985	1.857	32.1	2.557	2.289	5.513
PUTT	1.036	0.841	31.3	5.311	4.387	5.986
CHAT unoptimized	1.798	1.705	33.0	2.663	2.526	5.196

TABLE 11
First Sum (Kernel 11)

```

      X(1)= Y(1)
DO 11 K=2,1000
11    X(K)= X(K-1)+Y(K)

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	21977	6993	0	9	2001	2997
PUTT	21977	6993	0	9	2001	2997
CHAT unoptimized	31971	12987	1998	9	2001	4995
	Calculated number of floating point operations per second ($\times 10^6$)	Measured number of floating point operations per second ($\times 10^6$)	Instruction issuance efficiency (%)	Actual capacity ($\times 10^3$)	Kolence capacity ($\times 10^3$)	Calculated power ($\times 10^6$)
CHAT optimized	1.655	1.353	31.8	3.103	3.081	4.959
PUTT	1.655	1.326	31.8	3.103	3.081	4.959
CHAT unoptimized	1.137	0.718	40.6	5.150	4.482	5.681

TABLE 12
First Difference (Kernel 12)

D0 12 K=1,999
12 X(K)= Y(K+1)-Y(K)

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	21977	6993	0	7	2000	2997
FUTT	21977	6993	0	8	2000	2997
CHAT unoptimized	31971	12987	1998	7	2000	4995
	Calculated number of floating point operations per second (x10 ⁶)	Measured number of floating point operations per second (x10 ⁶)	Instruction issuance efficiency (%)	Actual capacity (x10 ³)	Kolence capacity (x10 ³)	Calculated power (x10 ⁶)
CHAT optimized	1.655	1.604	31.8	3.103	3.081	4.959
FUTT	1.655	1.622	31.8	3.103	3.081	4.959
CHAT unoptimized	1.137	0.542	40.6	5.150	4.482	5.681

TABLE 13
2-D Particle Pusher (Kernel 13)

```

DO 13 IP=1,128
  I1= P(1,IP)
  J1= P(2,IP)
  P(3,IP)= P(3,IP) +B(I1,J1)
  P(4,IP)= P(4,IP) +C(I1,J1)
  P(1,IP)= P(1,IP) +P(3,IP)
  P(2,IP)= P(2,IP) +P(4,IP)
  I2= P(1,IP)
  J2= P(2,IP)
  P(1,IP)= P(1,IP) +Y(I2+32)
  P(2,IP)= P(2,IP) +Z(J2+32)
  I2= I2 +E(I2+32)
  J2= J2 +F(J2+32)
  H(I2,J2)= H(I2,J2) +1.0

```

13 CONTINUE

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	23159	7808	512	29	4054	3584
FUTT	39552	12672	896	51	4054	4480
CHAT unoptimized	25725	9216	384	27	4054	3968

	Calculated number of floating point operations per second ($\times 10^6$)	Measured number of floating point operations per second ($\times 10^6$)	Instruction issuance efficiency (%)	Actual capacity ($\times 10^3$)	Reliance capacity ($\times 10^3$)	Calculated power ($\times 10^6$)
CHAT optimized	1.407	1.234	33.7	3.395	3.362	5.628
FUTT	0.824	0.764	32.0	4.870	5.744	4.119
CHAT unoptimized	1.267	1.111	35.8	4.221	3.735	5.609

TABLE 14
1-D Particle Pusher (Kernel 14)

```

DO 14 K=1,150
  IX = 3+(K/8)
    XI=IX
    VX(K)= VX(K)+EX(IX) +(XX(K)-XI)*DEX(IX)
    XX(K)= XX(K)+VX(K)+FLX
    IR= XX(K)
    RI= IR
    RX1= XX(K)-RI
    IR= IR.AND.63
    IR= IR-(IR/64)*64
    XX(K)= RI+RX1
    RH(IR)= RH(IR) +1.0-RX1
    RH(IR+1)= RH(IR+1)+RX1
14 CONTINUE

```

	Calculated number of cycles executed	Number of non-PASS instructions executed	Number of PASS instructions executed	Number of instruction words in loop	Number of data words used	Number of fetches and stores executed
CHAT optimized	21591	9000	600	25	510	2400
FUTP	35091	11700	450	34	510	2850
CHAT unoptimized	24897	10200	300	32	510	3000
	Calculated number of floating point operations per second ($\times 10^6$)	Measured number of floating point operations per second ($\times 10^6$)	Instruction issuance efficiency (%)	Actual capacity ($\times 10^3$)	Kolence capacity ($\times 10^3$)	Calculated power ($\times 10^6$)
CHAT optimized	2.779	2.505	41.7	2.428	2.950	4.042
FUTP	1.710	1.655	33.3	2.895	4.795	2.953
CHAT unoptimized	2.410	1.977	41.0	3.033	3.403	4.382

TABLE 15
Disk Charge Times for the CDC 817

	Block size 1000g	Block size 2000g	Block size 4000g	Block size 10,000g
Sector boundary read	3.293	1.339	0.784	0.376
Non-sector boundary read	3.013	1.231	0.851	0.313
Sector boundary write	2.551	1.377	0.598	0.565
Non-sector boundary write	6.979	3.683	1.878	0.912
Non-sector boundary write with no preserve bit on	4.839	3.704	1.828	0.895

TABLE 16
Disk Charge Times for the CDC 844

	Block size 1000g	Block size 2000g	Block size 4000g	Block size 10,000g
Sector boundary read	0.599	0.458	0.406	0.346
Non-sector boundary read	0.624	0.565	0.452	0.333
Sector boundary write	0.508	0.420	0.405	0.336
Non-sector boundary write	4.047	2.193	1.268	0.527
Non-sector boundary write with no preserve bit on	1.858	1.422	0.791	0.534

TABLE 17
Tape Read/Write timings

Operation	Type of timing	Timing values (sec) for 1,2,4 and 8 chained requests			
		1	2	4	8
Tape read	Charge time	6.260	6.224	6.208	6.218
	Real time	26.743	13.129	9.241	7.656
Tape write	Charge time	6.009	5.948	5.930	5.925
	Real time	24.099	15.897	10.337	8.155

TABLE 18
Compiler Summary

	Total calculated number of cycles executed	Total number of non-PASS instructions executed	Total number of PASS instructions executed	Total number of fetches and stores executed
CHAT optimized	257269	87953	4019	33513
PUTT	332985	109716	5299	40759
CHAT unoptimized	353016	142853	13650	53528

References

1. Control Data Corporation, Control Data 7600 Computer System Reference Manual, Publication number 60258200, Arden Hills, Minnesota, 1971.
2. DuBois, Pierre, et. al., CDC 7600 System Calls and I/C Requests, LTSS Chapter 10, Lawrence Livermore Laboratory, July, 1974.
3. Huskamp, Jeffrey C. and Pete Nickolas and Mathilde Killian, Utility Routine Programmer's Manual, LLL Document M-048, Lawrence Livermore Laboratory, May, 1974.
4. Kolence, Kenneth W., Technical Foundations of DP Cost Accounting, Prepared for the Bank Administration Institute, Park Ridge, Illinois, July, 1974.
5. Kolence, Kenneth W., DP Operational Cost Management, Prepared for the Bank Administration Institute, Park Ridge, Illinois, July, 1974.
6. Mendicino, Sam R., et. al., The LRLTRAN Compiler, Communications of the ACM, Volume 11, Number 11, November, 1968, p. 747-755.

Distribution

LLL Internal Distribution

Dieter Fuss, L-318

Perry Gluckman, L-607

Jeff Huskamp, L-307

3

Dave Pehrson, L-60

Ed Schoonover, L-67

Rich Zwackenberg, L-60

TID File

15

NOTICE

"This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights."