Prof. Saltzer

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Programming Linguistics Group Memo No. 2

Use of High Level Languages for Systems Programming*

Robert M. Graham

Note: This memo is an edited transcript of a talk given at a meeting of the NSA Computer and Information Sciences Institute on November 20, 1969.

* Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

Use of High Level Languages for Systems Programming

Robert M. Graham

Massachusetts Institute of Technology

(This paper is a slightly edited version of a transcript so that it still contains the colloquial flavor of the oral presentation.)

I'm going to talk about languages for systems programming, what they can do for us, and what we might expect from them in the future. These comments are largely based on my experience with the Multics System and I'll quote a few figures from Multics as we go along. I'm concerned particularly with large and complex systems.

I'd like to just quickly review the basic problems in the design and implementation of large software systems such as Multics, OS-360, TSS-360, or any of the other monumental systems that have been built in the last five years or so. The first problem one runs into is that of complexity. The systems I'm talking about are large. They have a large number of modules (subroutines). Multics is composed of well over five hundred separate subroutines. With a system this large you get a large number of interconnections and a large number of different interactions between the various modules. There is no regularity or iterative nature to the structure. Every module is different, structured differently. There is no clear hierarchy of calls between the modules. In general, large systems like this have conflicting objectives which add to the complexity. Multics, for example, allows complete sharing and complete privacy. These are two opposing, conflicting objectives and this adds to the complexity of the system.

The second major problem is that the project is usually very large. A system of the kind I'm talking about is too large and complex for one person to keep in his mind at once; even the very best systems programmers just can't keep all of the Multics System in their head at one time. Typically you need more than three or four people to implement such a system. Multics was implemented by about fifty people. TSS-360, which is a general purpose time-sharing system for the IBM 360/67 with objectives similar to those of Multics, took about 300 people to implement.

The third major problem is the length of time from the conception of the system until the system is working in a real environment. There is a long period of evolution and iteration of the design; for example Multics and TSS each took three to four years before the system was working at all.

This long period of time results in subordinate problems such as turnover of personnel which introduces a continuity problem. Most projects experience anywhere from 10 to 20 percent turnover in personnel per year. The training period required before a new programmer is effective in such a project can be as long as a year, with a minimum of at least six months. The number of people involved causes management problems. You get a hierarchy of management: projects and sub-projects and managers managing managers.

It also appears to be a universal law of some kind that in any large, long-run project the best you can get is one hundred to two hundred lines of debugged code per man month. Only on a very small, short-run project can you expect to get a higher productivity than that.

The length of the project introduces another problem: critical go-no-go decision points occur at one or more times during the life of the project. Large scale funds have to be committed to the project. Usually concrete results do not show themselves for some long period of time. You have to believe the people who are doing the work and commit yourself to continue funding the project to completion or cut it off at some point. The point at which to cut off the project is before you lose so much face that you have to dump in another hundred million dollars in order to make it work, just to prove your point. Multics cost, I'm sure, well over ten million dollars and I suspect that TSS must have cost as much as fifty million dollars to implement, probably higher. I'm just guessing at the IBM figure and I was partially guessing at the Multics figure because there were a number of organizations involved in it and I don't know all of their financial contributions.

A fourth problem in a large complex system is the inability to predict performance. We are still unable to accurately model any large software system and so we have to build a prototype in order to understand the system in the first place. So, only through building the prototype do we gain understanding and insight and only by building it do we get any sort of measure of how it will perform. In other words, we develop systems much like the Wright Brothers developed airplanes; they built an airplane and then pushed it off a cliff and it either flew or didn't fly. If it didn't, they started all over again. Now we don't have to start over again but we do have to redesign. If the prototype doesn't work right when it comes on the air, then we have to start redesigning. Multics has been completely redesigned and reimplemented at least once. There are some sections of it that have gone through this iteration of redesign and reimplementation several times.

Systems of this complexity and size can't be simulated effectively with any of the simulation techniques that we have today because they are too time consuming and too expensive. In general, to use any of the simulation languages of today you have to make a sidestep; you have to redesign the system, so to speak, and design out all of the irrelevant things. But you probably don't know what is irrelevant. Then you have to recode the redesigned system in the simulation language in order to simulate it.

It is a common property of human beings that intuition is not something one is born with; it is something one has developed and so one's intuition is poor when dealing with poorly understood relationships. So system programmers who say, "Well my intuition tells me that such and such is going to be really red hot in that system" probably will be wrong unless he has had considerable experience with very similar systems. Multics held a number of surprises for us. Many of the algorithms which were used in order to be sure that there was efficiency in some local area turned out to be so inefficient globally that choosing a simpler algorithm which ignored local efficiency actually achieved a much higher global efficiency. For example, the Multics hardware was built to deal with two different page sizes, obviously, we decided it would be more efficient to use both page sizes in transferring between the core and the drum. It turned out the bookkeeping and overhead of managing two different page sizes was far more expensive than using just one page size with its attendant breakage cost.

Now this process of iteration and redesign in the development of a large complex system is not new. In most other areas of engineering you build a prototype, observe its performance, experiment with it, modify the design,

rebuild it, experiment again, and so forth, until you reach the level of performance that you set out to get. This iterative technique is fairly common in the aircraft business, only one doesn't usually fly them; one uses a wind tunnel to test them out first. What I'd like to talk about today is the fact that software system developers today very seldom use the modern tools that are available; tools that they provide for other users of the computer system. In other words, they don't use wind tunnels; they don't use high level languages; they don't use symbolic debugging techniques; they don't use simulation; in fact, they don't do much of anything except write machine code and try and figure out why it doesn't work. But they do provide all those nice tools for the average user of the system. So I would like to address myself to what advantages one particular tool, namely high level languages, has in the design and implementation of software systems and particularly how its use can aid us in simulating or analyzing the performance of the system before it is actually built.

Lets look first at the situation today. How can high level languages immediately help the situation? In other words how can we use PL/I, FORTRAN, or some other existing high level language to help us? Let me say 95 per cent or more of Multics was implemented in a subset of PL/I, so there is no question that it can be done: it can. The Multics System is now operational and has been since October 1969 - operational in the sense of being open to the MIT community in general. Anyone can walk in who wants to use Multics, get a number, sign up, and use it (provided he has some money). It supports 30 users with performance equivalent to CTSS, which is the system we had on the 7094 before Multics. And this is a one processer system. So

it is not impractical to think of implementing an operating system in a language like PL/I. I understand that SABRE has been redone in a subset of PL/I, so that Multics is not a one shot fluke.

Using a high level language for a system implementation aids us in a number of ways. I'm going to mention four that are all somewhat interrelated. One way high level language can help is to reduce the complexity. The way we say this in programming linguistic terms is that there is a better match of the primitives in the language to the problem that we're trying to do. There is a lot of string manipulation in system work and a language that has strings as a basic data type and some operations on them such as concatenation or move is very useful. It is a much better match to the problem of manipulating strings than having to write copy loops that move words full of characters around.

Operating systems can be viewed as a very large number of very simple manipulations of a very complicated data structure. There are many tables of one kind or another in a system. Languages which have facilities for describing data structures like PL/I put us much closer to the problem because we can talk in terms of manipulating the entities in the table rather than shifting, masking, and extracting a certain field out of a word. In a PL/I program the structure of a table is described only once. All the details of the format, and how the data is going to be stored in memory are handled by the compiler. The user is unconcerned with them and unaware of them. He can think in terms of manipulating the entries in the table rather than pushing bits around.

These comments apply even to functions like input and output. Most of the modern computers have an input/output controller which is a small, special purpose computer. The way you do input/output is to assemble a small program for that computer. So, one can talk about this process of assembling a program for the I/O controller as a manipulation of certain data structures. Things like I/O and table management are much more natural and more easily expressed in these terms.

A second major way that high level languages can help is to magnify productivity. People seem to write the same number of lines of code per day regardless of what language they are using. It's a sort of measure of the number of marks they put on paper rather than the number of machine instructions. The use of a high level language magnifies the productivity of the person by whatever ratio there is between machine language instructions and source language statements. I think a good example of this is as follows: Unless you claim the Multics group is far superior in terms of talent than the IBM people, comparison of the two efforts proves the point. Fifty people implemented Multics while 300 or more poeple implemented TSS. Both groups took about the same amount of time and the systems are approximately the same size. Thus, you get a magnification in productivity of six or higher as a result of the fact that Multics was written in PL/I while TSS was written in assembly language.

Another way productivity is helped by a high level language is that the compiler can use the cleverest code generation techniques that are invented by the cleverest machine language programmer. The fact that the high level language is better matched to the problem means that it is easier and faster to change and redesign a program in high level language so that people can move faster and in effect do more work.

A third way high level language helps is to enhance understandibility. Programs in high level language tend to be briefer and more lucid than assembly language programs. One has less to cope with. There is a smaller amount of information at which one has to look in order to understand what's going on in the program. Another way of saying this is that the higher the level of the language the more the programmer says what is supposed to happen as opposed to how it's going to happen. And this understandibility is a help because it makes the system less complex. Much of the complexity in a large system is due to implementation details (actual bit manipulations) and is not inherent in the problem itself. Hence, a high level language helps to sort out, and relegate to automatic bookkeeping by the compiler, those irrelevant details which, if one were dealing on a machine language level, would add to the complexity although they are not an inherent part of the complexity. An operating system like Multics has enough complexity that is inherent in the nature of what it is trying to do without adding any more due to machine language programming.

High level languages also aid in transferability. We have the personnel turnover problem in a large project. High level languages are generally easier and quicker to learn. This means that new people coming onto the project can become effective more rapidly. It is easier for them to pick up programs that someone else has worked on in order to make improvements on them or to fix bugs. Transferability is aided by the fact that the compiler enforces standards on everyone writing in the compiler's language. One has no choice on making a standard call to a subroutine. The compiler always uses the standard. This can often be bypassed in machine language

and sometimes is by the bit pusher because he has found a way to save one or two cycles. Not observing standards of course makes it more difficult for anybody else to pick up the program and figure out what the original programmer was doing.

Now, as I said, some sections of Multics were rewritten a number of times. Let me give you a couple of figures. These are some of the improvements that were made in rewrites. One module was redone in three man-months getting a 26 to 1 improvement in size and 50 to 1 improvement in time. Another one got a 20 times improvement in size and a 40 times improvement in time and took two man-months. Now the interesting thing about figures like this is that the improvement generally came from a redesign. The programmer really, in some sense, understood the function that was going on in the module and discovered a much better and simpler way to do the same thing. In other words, once the system was implemented and working, people stood back and looked at it. They had a chance to look at individual modules and see how they fitted in the system. People had a chance to think about what the essential function of the module was and thus were able to understand it better and see how to simplify it greatly.

Now these are all things that the high level languages in existence today can do for us. These are things that I claim from experience with Multics have actually happened. We have actually gotten these kinds of advantages from using PL/I. Now, as of today none of the high level languages that exist do anything to aid us in predicting the performance of the system, except to the extent that when we understand the system better, we understand its performance a little better. But in general, there is no assistance in predicting performance. It is my contention that one can, by choosing an

appropriate high enough level language, get enough information in the source language description of a program so that one can predict the performance of the module without requiring the designer to separately formulate it in a simulation language and run a simulation of it. I propose to give you an indication of how that might be done.

So now I am going to look at what a high level language which is designed specifically for software design and implementation should look like; What ought we to have in it, what might we have in it. First, in order to match it better to the problem, we want the right kind of primitives. That means we need to isolate the basic building blocks that are used in software systems and the basic transformations that are used in software systems.

For example, I mean the following kinds of things. Table management is a very common thing that is encountered in software systems. You find symbol tables in both compilers and assemblers. You find symbol tables in most modern loaders. You find symbol tables in file systems. You find all sorts of other kinds of tables in system software - tables of processes waiting, tables of processes executing, tables of I/O device assignment, etc. So table management is certainly a basic function in software building. One can envision a data type called TABLE and a number of basic functions that operate on entities of this data type, such as, adding an entry to the table, deleting an entry from the table, modifying an existing entry, locating an entry in the table for a given key, and copying the contents of an entry from the table. A language incorporating this data type and these functions would allow one to specify the elements in a table entry, possibly

in a manner analogous to a structure declaration of PL/I. It would also allow one to specify what elements of an entry are going to be used as keys in searching, allow one to specify (if one cares) the particular search method to be used (there are a number of fairly standard search methods), and might even ask one to specify which operations are going to be performed relative to each of the different keys that one specifies.

Now this I claim is enough information so that the compiler can do a much better, more efficient job of implementing references to tables and the management of tables. Also, the compiler has a much better hold on the cost (performance) of each table manipulation. So that if a compiler for a language which included things like the above were coupled with supporting analysis programs, we could automatically get some sort of hold on the performance of a module described in this language.

Now to give you a little better idea of what I mean, let me describe how one could get some sort of performance information out of a compiler for the PL/I language. The compiler is certainly capable of giving you information with regard to the amount of space used and the amount of time it takes to execute many of the constructions in PL/I. For an assignment statement, it can figure out the number of machine language instructions that are going to be compiled for that statement. Thus, it knows how much space the statement will take. It also knows how much time it will take to execute each of those instructions, so it can tell you how long the assignment statement is going to take to execute.

PL/I requires declarations for the data that the program uses. For the more complicated data like structures, the compiler itself figures out how it is going to be formatted in memory. Thus, it knows how much space it's going to take. It also knows how much supporting information needs to be generated. I'm thinking of things like dope, which is supporting information describing the layout of data structures in memory which is required when the structure is passed as an argument to a subroutine.

For DO loops which have a fixed number of iterations, the compiler can compute a summary figure for that entire loop. By multiplying the number of times around the loop by the times for the various statements within the DO loop, a total time for the loop can be computed. If the number of iterations is variable, it could print out a formula for you which is a function of that variable. In certain simple programs the compiler might even be able to develop some formula that expresses the performance of the entire program as a function of a number of variables. One could then evaluate this for the differing values of the variables.

The compiler is in a position to do this much better than you because compilers for languages like PL/I put in calls to runtime support routines and additional instructions for the setup for various kinds of arguments in different ways depending upon the data type. It does a lot of things that are hidden to you and unless you read the machine code you can't even get a feel for the way the program is going to perform. We ran into this in PL/I. This is one of the disadvantages of using a high level language; it is difficult for the programmer to get a feeling for how long his program is going to run. But the compiler is in a position to do something about this.

For example, one program consisted of two statements. The first was an assignment statement which concatenated a carriage return on the end of a character string. The second statement in the program called a print subroutine to print the augmented string. And that was the whole program. The compiled program occupied 180 odd words of memory. The writer had no feeling whatsoever that it was going to be anywhere near that magnitude of compiled instructions for such a simple program.

Now where the compiler begins to fall down with languages like PL/I in attempting to put these figures together is that it doesn't know a lot about what it is you are trying to do. It merely knows how you are doing it. If you write a DO loop to search a table, it doesn't know that you are searching a table. So a compiler for high level language which included table as a data type and functions operating on tables would be able to evaluate the table manipulation operations and compare them according to various search techniques. You could leave the search technique unspecified and give the maximum size of the table, the average number of entries, etc. The compiler could then figure out which search technique would be most efficient under the circumstances. So it could develop formulae giving the performance in terms of the table size and the average number of entries in it.

Now I have gone through the table manipulation discussion merely as an exercise to indicate what one might do immediately. My feeling is that this first step is too small and that you want to go much further. In fact, members of my research group are investigating high level concepts in system implementation and design. We would like to evolve a language such that enough information to build a good model of the program would be inherent in

the program. Because if you can build a good model of the program, you can predict what it is going to do. You can also compare the efficiency of different varients of the same function.

It is even conceivable that one may be able someday to calculate theoretical lower bounds on the various operations. For example, one may be able to calculate a lower bound on the time for table searches given a particular table organization, in much the same way as Winograd has developed formulae that give absolute bounds on the speed with which hardware can possibly do arithmetic. The value of knowing such a bound is that, if you know a bound and if you get fairly close to it you need not look for another, more efficient method, because any other method you find is not going to be much better. The trouble with trying to implement things efficiently is you don't know how well you can do. You never know whether another day's or week's work of looking for another method will yield a very high pay-off. If we can build good models for these things then we have some hope of trying to achieve such theoretical bounds.

Let me give an example of what I mean by a much higher level concept, or much lower level concept depending on how you look at things. In Multics there is a file system which is based on the virtual memory concept. The user has a view of the file system as consisting of a number of files that have symbolic names. These are organized into a number of directories which are structured in a hierarchical way. He refers to a file by giving a name which consists of a concatenation of the names of the directories in this tree needed to locate the particular file he wants. The hardware has two dimensional addressing called segment addressing. This is the GE-645 hardware. Similar hardware exists on the IBM 360/67 on which TSS is implemented.

One of the major functions of the file system in Multics is to transform a symbolic reference to a file into an actual two dimensional address which the hardware can use, which is a pair of integers representing a segment number and a word number. A very large part of the file system is involved with that transformation. One can view this as a mapping from one address space to another. The user's address space consists of symbolic tree names and the hardware's address space consists of pairs of integers. Thus we require a mapping from one address space to another. It is very similar to the kind of address space mapping that a compiler does. When you write in PL/I, or some other compiler language, the only memory you have is in terms of the variables you define. They may be single scalar variables, they may be arrays, or they may be structures. You use symbolic names to refer to them. The task of the compiler is to map these symbolic references into machine addresses which are usable by the hardware to refer to its memory. So that address mapping - taking a reference in one address space and mapping it into a reference in another address space - is a large part of what the file system in Multics does. There are a number of other functions in systems which can be expressed as address space mappings which we will not explore here.

Now actually the Multics hierarchy of files also has some things in it called links which make it into a directed graph as opposed to a strict tree. Most hardware memories are linear; just a single address identifies an element of them. Some of the newer segmented memories have two dimensional address spaces. One can talk about address spaces as having structure and talk about primitive operations on address spaces, e.g., creating new address spaces by adding a new address to an existing address space. One

can further talk about mapping between them and one can conceive of a theory of address space mapping or a theory of structure transformations that will allow one to say some very general things about them.

This kind of model for certain functions of the operating system would be powerful enough, in terms of the information inherent in it, so that a compiler would have an extremely wide latitude of choices on how to implement it. The compiler would also be able to evaluate many different implementations of it. If the compiler is able to do that, then of course it has a lot of information on how the thing is going to perform.

If you have a language that includes concepts like address spaces, then you can envision a sort of iterative design procedure taking place using such a language. You get a rough design for the system in terms of the major modules of the system. Then you sketch the structure and the control sequencing some way. Then you define and describe the data structures that are going to be involved; the tables you are going to use, the stacks you are going to need, the queues, and so forth. Then if you describe the functions of the various modules in terms of table manipulations or address space mappings or some such concepts, the compiler and its supporting analysis routines would be able to extract enough information from this description to build a reasonable model of the system which you had so far specified.

Now the model presumably would contain a number of parameters. I mentioned some in connection with table searching. The density of a table is important if you are using a random access organization while the average number of entries in the table may be important if you are using some other organization. If the designer were on-line with this system he could vary

these critical parameters in the system and observe the effect of the variation on the performance of the system by displaying the values of certain critical variables which meter the performance of the system. Now this is way out in the future because the problem of identifying what variables measure the performance of a system is a very wide open research problem. In other words, if you are going to predict the performance of the system ahead of time, you have got to know what things you have to measure. This is an area which is very undeveloped at the present time. The problem is to find out what variables in the system really characterize its performance and then try to generalize these some way so that we can say, for example, "The average search time of critical tables is what really is the key to performance". In summary, the design process consists of varying the parameters of the system, such as, the average size of the tables, the amount of core memory available, the number of separate channels to the secondary storage and other parameters which characterize the performance, observing the result and change in performance, and then modifying your design accordingly. The designer iterates around this loop until he homes in on satisfactory performance characteristics.

Now all this can be done, you see, without building any hardware or without actually implementing the system, because what you are dealing with is the description of what the system is going to do rather than an actual implementation of it. Now if we are clever enough in the compiler business, we will be able to build a compiler that can take that language and actually compile code for the system for some particular computer; and if we are really clever, we can figure out how to describe hardware so that we can describe the hardware to a non-existent computer to the system and it will

give the performance for that hardware. Then you can try other hardware designs to see what hardware will make the thing work well. Of course beyond that, if we are really extremely clever, we won't even bother separating hardware from software; we will just describe the system and the design system will figure out what ought to be in hardware and what ought to be in software. Well...maybe our grand-children can do that for us.