Brian Marks

Compilation to Compact Code

A compilation process is described that emphasizes small object code rather than fast object code. The approach entails synthesizing an instruction set and an interpreter for that instruction set during compilation of an individual source program. Numerical results are given for compiling a systems programming subset of PL/I to System/370 code.

Introduction

The functions which a particular computer hardware configuration can execute efficiently are often limited by the amount of main storage in the configuration. Over the years the functions expected of a computer system have increased, and the programmer's ability to provide those functions has also increased through the use of more productive tools. Hence main storage is still a key consideration, despite its reduced cost per bit.

The problem of combining high function with small main stores is particularly relevant when processors are combined in a network. The system designer will want to minimize the difference in the functions provided by processors of different sizes. The smaller processors will have significant limitations in addressability as well as amount of storage. Where programs are transmitted about the network, their size will influence the time for transmission.

In this paper we consider some of the contributors to storage utilization to be fixed, and we examine improvements in others. The fixed items are

- Storage for data. We are concerned here with storage for executing programs and not with methods of compacting data.
- The skill of the designers and programmers.
- The language in which the programs are written. To fix this factor we assume that the programs are written in a language which does not trade object program efficiency for programmer productivity. (Or if it does, we assume that the programmer has avoided using language features that are difficult to compile efficiently.)

Given these factors, there remain several approaches to reducing the amount of storage devoted to executing programs.

The hardware approach consists of using a processor with an instruction set that is highly space efficient, *i.e.*, keeps the number of bytes needed to represent a given function low. Existing processors have space efficiency as a consideration in their instruction set design, but extra emphasis on space can give more compactness. If a set of programs is available that is "typical" of the programs to be executed on the processor, then the statistics of this set can be used in the instruction set design. Short instructions will be used for frequently occurring functions, along the lines of Huffman coding [1, 2].

The same approach to instruction set design can be taken when a software interpreter is used to execute programs. The software interpreter itself will need storage, which affects the overall compactness achieved.

Unless the range of applications for the processor is very narrow, any set of "typical" programs will be a compromise and may well not be statistically similar to the code that comprises a particular application for the processor.

This paper details a software approach to code compaction that does not involve the compromises implicit in using a set of "typical" programs. The design of the instruction set and the construction of the interpreter for that instruction set are done dynamically during compilation of an individual source program. The instruction

set of the interpreter is approached by first compiling to the instruction set of the actual hardware and then recognizing common sequences.

To measure the effectiveness of code compaction one needs a standard of comparison—for a given function how many bytes of code should there be if the compaction is good? Could the compaction be merely a reflection of poor quality in the uncompacted compiled code? Within IBM, the natural standard of comparison is code produced by the PL/S II systems programming language [3] for System/370. This compiles a dialect of PL/I, and variations of it have been used to develop many software products over the last ten years. It is similar to BLISS and BCPL [4, 5] in having demonstrated the combination of ease of programming and the object code efficiency that systems programmers require.

We adopt as the base for our measurements a compiler that uses compilation techniques and a run time environment similar to those used by the PL/S II compiler, including a simple flow trace and calling sequences similar to the SAVE/RESTORE mechanism of OS/370. This base compiler differs from PL/S II in ways that do not affect object code size. A proper subset of PL/I is supported, so that it can be compared with other PL/I compilers. Compilation is directly to machine code, and not a cascade through assembly language.

The base compiler produces nonreentrant OS/370 object code. The phases of the compiler itself are used as sample source programs for measurements. When self-compiling, the compiler produces an overall 11 bytes per PL/I statement from its 6500 source statements. The count of statements is a semicolon count so that "IF A=1 THEN B=2;" counts as one statement. The compilation speed is 6000 statements per minute. Machine times are IBM System/370 Model 168 CPU times.

This paper details three variations on subroutine recognition. High level subroutine recognition, recognizing in the source program subroutines that the programmer might have written, was not found to be effective. Low level subroutine recognition, recognizing small pieces of machine code, is more effective. The technical difficulties lie in predicting what the size of a piece of code will be before all the code details are settled and in choosing between potential subroutines that are mutually exclusive because the pieces of code that would comprise them have an overlap. Simple but nonoptimal strategies are given to overcome these difficulties.

If the calls to the recognized subroutines are made by the natural linkage instructions of the hardware, a typical saving will be 15% in space at the expense of 15% in speed.

Tailored interpretation, where the recognition is extended to very small routines which are invoked by an interpreter, can halve code size at a much greater cost in speed.

High level subroutine recognition

If similar sequences of code were being recognized mechanically, there would appear to be advantages in feeding back the results to the programmer. The programmer could then evaluate the possibility of using a subroutine for these sequences and change the source program accordingly.

The advantages of advising the programmer of potential changes to the source program, as opposed to having the compiler create routines, are

1. The problem is made easier. When the ultimate decision is the programmer's, the subroutine recognizer does not need to be so accurate in measuring the merit of some potential subroutine. It can list the things which appear on a mechanical basis to be marginally not worthwhile, as well as those that look worthwhile.
2. Pattern matching is expensive in CPU time. Incorporating the results into the source program avoids doing the work on every compilation.
3. There are benefits in involving the programmer. It appears in practice that repetitious code tends to occur in areas where the programmer has not really thought through the problem (even if the code works). Directing attention to the area may result in a clean-up far beyond what could be done mechanically.

The utility program we use to measure the potential of this approach operates on one of the compiler's internal versions of the source program, an object-machine-independent, *n*-address form (Fig. 1). This is a compromise—operating on an internal text from a later stage would be more accurate from the point of view of assessing the real gain in code space from a subroutine; operating on an earlier text would make it easier to limit the search to those pieces of code that can be isolated as source. (Consider "IF A>B THEN J=K" and "IF C>D THEN J=K." At the source level we can make only "J=K" a subroutine, although at the hardware level we would be able to include in the subroutine the Branch on Condition instruction that tested the condition code setting.)

The utility evaluates potential subroutines in isolation, *i.e.*, on the assumption that there is only one being created. (If the decision were not in the user's hands one

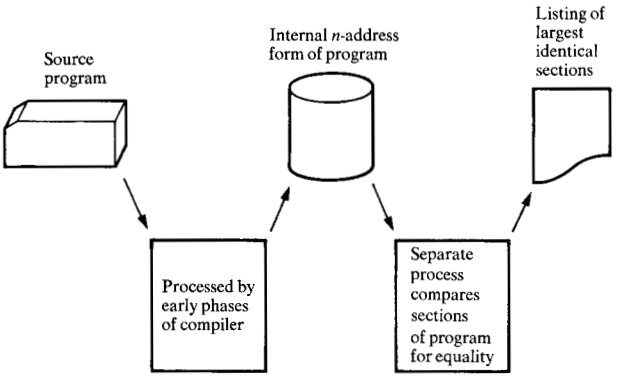

Figure 1 High level routine recognition.

would have to worry about how the decision to create subroutine X could influence the value of creating subroutine Y.) The utility searches for sections of code that are similar to the extent that they become the same if up to three of their operands are made into parameters. Each of these parameter-operands may be referenced many times in the section.

The utility does not attempt to re-order the text in order to promote the presence of similar sections. That would require flow analysis to determine what changes were allowable, and there would be many permutations of what changes ought to be tried. In practice the failure to re-order is not a great loss. If there are two sections that are similar except for a stray statement in the middle of one of them, then the utility will indicate that the piece before the stray instruction is similar to the start of the other section and that the piece after the stray instruction is similar to the finish of the other section. This should provoke the user into thinking about moving the stray instruction.

Because operators (as opposed to operands) cannot be parameters, the comparison of sections against all other sections can be done separately for each operator, comparing sections that start with that operator. (This makes the necessary computing reasonable—less than a minute of Model 168 time for a 1000-statement program.) Sections are considered in pairs, the text following an operator appearance being compared with the text following each subsequent appearance of that operator. For a given operator, quite short similar sections must be recorded, since subsequent appearances of pairs that are similar to each other and to the recorded pair are possible, thus increasing the merit of this section as a subroutine. After all the comparisons for one operator have been made, only the most meritorious need remain recorded. As the utility works through the operators, it keeps a record of the "best-so-far" potential subroutines. Finally the line numbers of these similar pieces are listed for the user. There is also a listing of those subroutines that the programmer wrote which were only called once, so that he can review whether they should have been subroutines.

As usual, the compiler itself was used as a source of test cases. When the utility was applied to the compiler source code, it uncovered a section of 18 lines suitable for a subroutine, some smaller sections suitable as subroutines, and a great many other similarities that were either too small or alternatives to the best routines. The total saving attributable to use of the utility was 922 bytes, 1.3% of the object code. This saving is not entirely negligible—if some program were reaching the limits on hardware with 16-bit addressing, one would be very glad to recover a few hundred bytes—but it is not large when compared with other approaches to compaction.

Some of the saving from this high level recognition would be obtained by low level recognition (*q.v.*) in the absence of high level. Other test cases and a more sophisticated utility might achieve more than 1.3%, but it seems safe to say that the effectiveness of this approach will always remain limited—a way of doing a little tidying of the source program but not a major contributor to compaction. It takes programmer time to decide which sections suggested by the utility are actually worth changing in the source—we do not have a way of measuring what the same amount of time spent in actually redesigning the programs would achieve.

Low level subroutine recognition

Here we consider automatic subroutine recognition on a more detailed internal text, leading to subroutines that the programmer could not have written. The text chosen is *n*-address from a late stage of compilation, and its relevant characteristics are

1. It is machine code to the extent that the final length of any sequence is known (with minor exceptions). Given the length of a sequence and the number of times it appears, one can calculate the saving from turning it into a subroutine.
2. Branches are not resolved to final addresses. Because System/370 branches have a displacement which is not relative to the branching instruction, the branch instructions will tend to appear as all different. Leaving the flow of control in an IF-THEN-ELSE format makes it possible for sequences to be identical even when they imply branching.
3. Registers have been allocated. In part this is a disadvantage, since two sequences may differ only in some transient register usage. However, it seems to be

necessary for a simple scheme since the short lengths of code we are considering cannot have their lengths estimated with any accuracy in the absence of information about whether registers are to be used in their expansion. In practice the method of register allocation adopted (with each variable having its preferred register for transient residence) rarely leads to code sequences that differ only in transient register use. However, the freedom to allocate registers subsequently, as in the calling sequences for nested recognized routines, is lost. It may be that leaving register allocation until after subroutine recognition, accepting the inaccuracies in measuring the potential of different subroutines, would be a more effective scheme overall; this has not been measured.

The recognition algorithm does not allow for parameters; it only recognizes identical sequences. The advantage of this is that it makes recognition much easier and hence makes it reasonable to do the recognition on each compilation. It takes about two seconds of Model 168 time to do the recognition for a 1000-statement program.

Whether the absence of parameters is a disadvantage is probably a matter of machine architecture. On the System/370 the Branch and Link mechanism requires a register to be used for the return link. The gains from making subroutines are marginally offset by the poorer code that results from there being one fewer register available for the nonhousekeeping code. If additional registers were taken for passing parameters, this loss would be increased, and any other way of passing parameters would probably make the subroutine so much longer than the code it replaced that it would rarely be profitable.

The recognition algorithm assumes that a space saving is worth making irrespective of its cost in speed. It would be possible to heuristically determine a figure of merit that would cover both speed and space [6]. This would get very complicated in the case of deciding between alternative sequences that occurred many times.

Figure 2 describes the subroutine recognition. The operations of the internal representation are divided up by hashing and making chains through operators with the same hashed value of operation code. The effect of this is to split the computation up, since equivalent operations will be on the same chain.

For each chain, each pair of operations from the chain is considered. The operators of the pair are taken as the starting points for comparing two sections of code. The distance over which these sections compare equal is found. The distance is measured in terms of the halfwords

Compile source program through early stages of compiler.

⇓

Develop n-address internal text with hash chaining of operations.

⇓

Pairwise examination of operations and their following text, for each hash chain. Matching sections of text form groups.

⇓

"Best" compatible groups become routines. Reflect this in the internal text.

⇓

Complete the compilation.

Figure 2 Low level routine creation.

of machine code that the section represents. (Strictly speaking, the comparison is not just for equality—if both sections have an IF-THEN-ELSE construct in them and these constructs are equivalent at their beginnings but not totally equivalent, then no part of the constructs is considered equivalent. The actual branching, as opposed to the THEN and ELSE clauses, is thus prevented from being split by a subroutine call.)

Although equivalent sections are found as pairs, they are subsequently held as groups, *i.e.*, if (A,B) is an equivalent pair and also (B,C), then the group (A,B,C) is formed.

Each group is considered as a potential subroutine plus calls. The group may be unsuitable because the code sequence it represents is short and there are not enough occurrences of it to outweigh the space overhead of a subroutine. The groups that look profitable when considered individually cannot, in general, all be made into routines, because the code sequences will overlap. On System/370 the simple BAL mechanism with a single dedicated link register does not support nested calls to subroutines. This makes a potential subroutine that is totally contained in another just as much of a problem as partially overlapping potential routines.

Theoretically it would be possible to find the best combination of subroutines by collecting all the candidates and evaluating the merit of every combination. However, it would be computationally beyond reason to do this on every compilation, so a less than optimal approach is taken. A list of "best" groups is maintained, and the groups are considered serially for inclusion in the list. The decision for the group is made on the basis of net gain in space. Note that the outcome is not necessarily a yes/no on addition of the whole group to the "best" list, but may

Table 1 Results of low level subroutine recognition.

	Stmts	*Bytes* (after saving)	*Seconds*	*Saving* (% of original)	*Number of routines*	*Average length* (bytes)
Phase I	1114	8236	13	10.9	52	17.0
Phase J	826	8640	12	7.2	55	15.6
Phase K	1191	10894	16	11.6	83	14.4
Phase L	590	5904	8	8.1	37	13.0
Phase M	1862	18384	55	20.1	129	15.2
Phase N	1076	9894	11	9.9	68	13.9
Phase O	418	2592	6	5.0	11	16.0
	7077	64544	121			

be some combination of keeping the group and overlapped "best" groups with fewer members.

In principle the result of this whole sequence of decisions depends on the order in which the groups are considered. However, in practice the space difference between the solutions from different orderings is not great.

Table 1 presents the figures for self-compiling the compiler with low level subroutine recognition. These figures show that the gain is made by creating a relatively large number of small routines. Approximately one subroutine was created for each 15 statements of source code. The average size of these routines was 15 bytes. A BAL to one of them was compiled once per seven machine instructions (on average) giving a 15% degradation in speed from the original code.

As one would intuitively expect, the results are nonlinear. The larger phases show a greater percentage saving, and the time for finding and choosing the best routines increases rapidly with the size of the source module and the savings made.

Tailored interpretation—the interpreter framework

As more and smaller subroutines are used, it is the length of the calling sequences that limits the saving in space. In order to reduce the calling sequence we can represent the call in a more compact form than the hardware instructions and make the actual call by inspecting that compact form at execution time. This interpretation of a representation of the source program that is not totally comprised of hardware operations is an extreme trade-off of speed against space.

There are many interpreters of compact forms. APL is an example where the internal form is very compact in terms of the function that it represents [7]. However, a very compact and general form of the text being interpreted implies a large interpreter. Since we are concerned with the case where the interpreter is in software and regarded as part of the overall size of the program, such an approach can only show space savings on very large programs.

Here we are concerned with smaller programs and restrict the interpreter to a few hundred bytes. This makes the internal form and the interpreter more oriented to a particular hardware. The internal form can be regarded as code for a software-enhanced version of the S/370 architecture.

We can usefully view the interpreter in two parts, a fixed part that handles the flow of control and calls routines and a tailored part that consists of suitable routines for the program being executed. In our examples the compiler derives the tailored part from the source program. The details of the fixed part depend crucially on the underlying hardware, but the techniques for developing the tailored part do not.

A natural fixed-part mechanism for the System/370 would be to use the invalid and privileged operation codes, in amongst normal System/370 object code. Nonprivileged execution of these by the System/370 would give rise to exceptions that could be made to call tailored routines. If only hardware timings are considered, this looks to be an efficient method, but in practice, using the MVS operating system, it is not, since the operating system assumes that the exceptions are the result of errors and need not be handled quickly. Hundreds of machine operations are executed in the operating system when handling each exception.

The fixed part used in our experiments operates as follows. It is described in terms of how it processes the "instructions" in the internal form that it interprets—some of these instructions are in the System/370 instruction format and some are not. The compact forms of the calling sequences to both the subroutines written by the programmer and the recognized subroutines will not be System/370 instructions.

The current instruction's operation code is used as an index to a table. If it is marked in the table as a System/370 operation, then the instruction is executed using the hardware EX instruction. (The EX operation itself is an exception.) Note that the subject operation may be a supervisor call, in which case the single EXecute results in some operating system action—it is never necessary for the interpreter to interpret operating system code. If the operation code is not marked as a System/370 operation, then it is to be interpreted by the mechanism as a branching or subroutine calling operation. The table contents are determined by the compiler, and only those System/370 operations actually used by the particular program have to be marked as System/370 operations—this increases the operation code values available for other uses.

Some operation codes are reserved for relative branching, with the condition mask being part of the operation code. One operation code is reserved for calling routines without parameters—the following byte contains the routine number. This routine number is the index of the routine's address in the table of routine information. It may be equal to some System/370 operation code. The procedures written explicitly by the user are also invoked in this way, after loading arguments into registers.

Operation codes that are not marked as System/370 operations and not used as the operation codes for parameterless routines are used as operation codes for routines with parameters. The argument being passed to the routine as a parameter is in the byte succeeding the operation code. The interpreter copies the argument byte into the actual code of the routine before calling the routine. (This would qualify as "tricky coding" if a human coder did it, since even an operation code in the routine may be overwritten, but since only the compiler-developer has to understand such code, this is not a severe problem.)

The mechanism for calling the routine distinguishes among single instruction routines (which can be invoked using the EXecute instruction and need no return linkage), routines where all the operation codes are System/370 codes (these can be branched to and not interpreted), and other routines (which need a general call mechanism with a stack of return addresses). See Fig. 3.

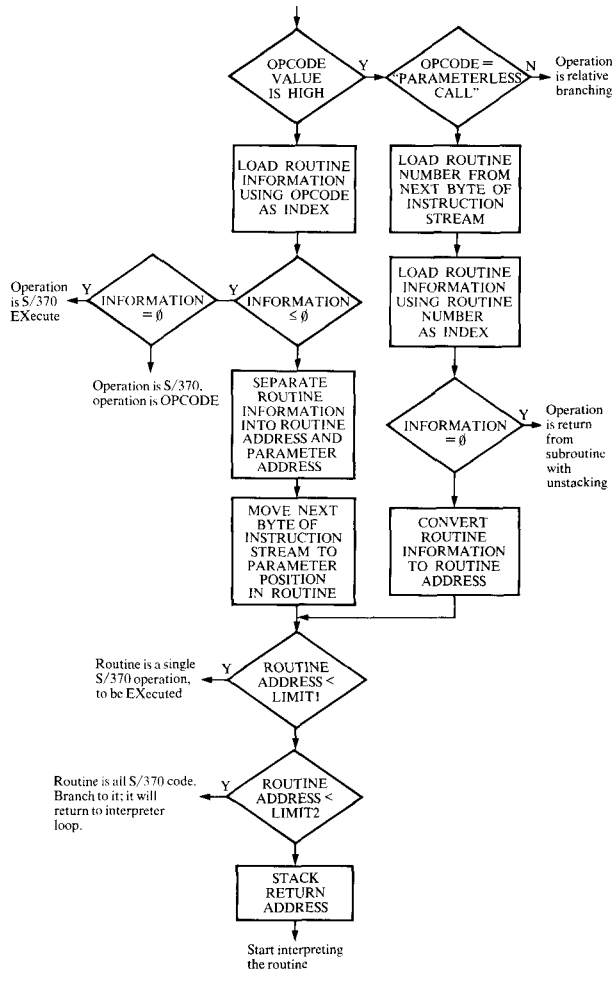

Figure 3 Operation decoding by the interpreter.

This fixed part represents a compromise between small size, 320 bytes, and the desire to cover most features that offer a space economy.

The tailored part of the interpreter for a particular program consists of a routine table, three program-specific values, and the routines themselves. The routine table has an entry for each operation code indicating whether or not it is a System/370 code, the routine address, and the position of any parameter. (By using an aligned offset for the address and restricting the parameter position to the first 16 bytes of the routine, this information can be held in 16 bits.)

Two of the values are used in distinguishing three types of routines: one-instruction routines, routines comprised

entirely of System/370 code, and others. The different classes of routines are physically separated in memory at run time, and these values are the boundaries of the classes, *i.e.*, the class of a routine can be determined by comparing its address with these values.

The third program-specific value is an adjustment used in relative branches. The fixed part of the interpreter computes the target address for short branches, relative to the position of the instruction, as twice the one-byte argument (regarded as unsigned binary) minus the adjustment. If the adjustment were 256 bytes, each relative branch would cover a range symmetrical about the instruction. In practice there are more forward branches than backward ones, so that increasing the range forward at the expense of the range backward will permit more branches to be two-byte branch instructions. The compiler chooses a near-optimal value for the adjustment. (In practice the actual advantage of this mechanism over symmetric branching is small.)

Tailored interpretation—creating the routines

The compiler that takes the fixed part of the interpreter as the target machine is a variation of the compiler that does low level subroutine creation. The extra mechanisms are required for

1. The creation of routines with parameters and of routines that call other created routines.
2. The sorting of routines so that they will lie in the appropriate physical order in memory at run time.
3. The relative branch optimization.
4. Creating the routine table, etc., as part of the object module.

The latter three are straightforward. The complications of parameters and nested routines make the algorithm given in Fig. 2 for routine selection impractical. Instead a multipass algorithm is used:

1. Potential routines are discovered and given a figure of merit, which is the space saving that the routine would achieve in isolation. This discovery is done in the same way as for low level routine creation, comparing members on hash chains, with appropriate elaboration to record parameter possibilities.
2. The potential routines are entered on a list. Since no routines are discarded because of overlap at this stage, the list may become large. If it becomes too large, the least meritorious routines are dropped from the list.
3. When the list is complete, the potential routines are considered in order of merit. A routine is accepted if it does not overlap any routine higher in merit. The text of the program is altered to contain the body of the routine and the calls to it.
4. If any potential routines were rejected in steps 2 and 3, the process is repeated from step 1 with the new text. (Typically it takes three or four passes before all the routines are selected.)

There are many minor elaborations to the process described above, because for example there is a limit to the number of operation codes available and hence to the number of routines. This limit is determined by inspection of the text before the process starts.

Note that compaction could be applied selectively to the text, *i.e.*, some routines could be given an attribute by the programmer to indicate that they were not to be compacted. They would then not be scanned in the search for routines and would be compiled to System/370 code and become routines that the interpreter would run at full System/370 speed. Such attributes would allow the programmer to balance the speed and space of the object module.

Table 2 presents the results for creating a tailored interpreter for each of the phases of the compiler. These subject phases are the same as those used to measure subroutine recognition.

The effect of the fixed interpreter architecture is to make more small routines profitable. The interpreter and routine table are included in the size. This adds to the nonlinearity, with the larger phases showing the larger percentage savings. From a practical point of view the figures for large phases are the most relevant. The overall figure is 6.0 bytes per statement.

No separate measurements were made to determine how much saving was due to subroutine recognition and how much due to other mechanisms of the interpreter, but from inspection of the code compiled it appears that some 10% of the original code size would be saved if the interpreter implemented only the short relative branch mechanism.

Phase M was the only phase for which the compaction was limited by the number of operation codes available. Many of the 48 internal procedures in the source of the phase are called only once, and some further compaction would be obtained if the source were rewritten with this code in line, freeing up operation codes and allowing more created routines.

In raw speed the fixed interpreter is some 15 times slower than the speed of the System/370 instructions it executes. This is diluted by time spent in the operating system and in noninterpreted subroutines. The compile speed of the compacted compiler was measured on small

Table 2 Results for tailored interpretation.

	Stmts	*Bytes* (after saving)	*Seconds*	*Saving* (% of original)	*Number of routines*	*Average length* (bytes)
Phase I	1114	5680	35	39	128	7.3
Phase J	826	6128	49	34	145	6.5
Phase K	1191	7172	86	42	168	6.6
Phase L	590	4514	24	30	92	7.1
Phase M	1862	11654	182	49	190	7.6
Phase N	1076	6942	59	37	171	6.5
Phase O	418	2278	8	16	42	7.5
	7077	44368	443			

test cases and found to be eight times slower than the uncompacted version.

Conclusions

Over the years, more work has been done on compiling fast object code than on compiling compact object code. The numeric results presented in this paper indicate a worthwhile return from attention to compacting. The results will, no doubt, be improved in the future when more work has been done on algorithms for discovering and choosing between potential subroutines.

Looking for subroutines that could be reflected in the source program did not prove to be a success.

Looking for subroutines at a lower level gave a noticeable compaction, as the result of the compiler creating a large number of small routines. With the run-time mechanisms constrained to those provided directly by the System/370 hardware, a typical saving would be 15% at a cost of 15% in speed.

Extending this approach to allow parameters and the nesting of routines, together with relative branch instructions, can produce a 50% reduction in code space. This size includes the interpreter necessary to make the mechanism run on System/370. Execution is many times slower.

The programmer could be involved in balancing the speed and space of the object program by marking some procedures in the source program for exclusion from the compacting process.

References

1. E. C. R. Hehner, "Computing Design to Minimise Memory Requirements," *Computer*, 65-70 (1976).
2. A. S. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Commun. ACM* **21,** 237-246 (1978).
3. *Guide to PLS/II*, Order No. GC28-6794, available through the local IBM branch office.
4. W. A. Wulf, D. B. Russell, and A. N. Habermann, "BLISS: A Language for Systems Programming," *Commun. ACM* **14,** 780-790 (1971).
5. M. Richards, "BCPL: A Tool for Compiler Writing and Structured Programming," *AFIPS Conf. Proc.* **34** (1969).
6. C. M. Geschke, "Global Program Optimizations," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1972.
7. M. Alfonseca, M. L. Tavera, and R. Casajuana, "An APL Interpreter System for a Small Computer," *IBM Syst. J.* **16,** 18-40 (1970).

Received December 17, 1979; revised June 19, 1980

The author is located at the IBM United Kingdom Laboratories Limited, Hursley Park, Winchester, Hampshire SO21 2JN, England.