factor would have hopelessly overburdened an already extensive task of experimentation. I can only invite Dr. Proll and others to illuminate the effect of variable slackness by conducting additional investigations. My intuitive prediction would be that variation in slackness would decrease the number of 'effective' constraints since the most severe constraints should dominate the path to optimisation. Unfortunately the three problems of Petersen presented by Proll do not provide conclusive evidence one way or the other as to the effect of variability of slackness.

As stated in the paper the specific regression results are recommended only for problems in the range of the factors B and C as studied. Rather obviously, regression should be performed over the ranges of importance to a specific user if larger problems are involved. The examples shown by Dr. Proll do indeed exhibit a divergence of Z_0/Z_u predicted from the Z_0/Z_u actual. Nevertheless a broader, systematically generated sample would give us a better idea as to whether the divergence exhibited is specific to the problems reported or more general in nature.

Quite frequently, problems in integer programming do not allow meaningful 'intermediate' feasible solutions. This is the case in problems with equal c_j values in certain formulations of line balancing, multi-project scheduling, set covering, and trim loss problems. Thus Balasian techniques do not always yield 'a sequence of feasible solutions of increasing value'. Nevertheless this is indeed sometimes the case as Proll points out. There have been several techniques proposed (and used) for initialising branch-and-bound codes including the 'quick-trick' and linear programming starts of Geoffrion and Salkin-Spielberg, the heuristics of Holcomb, Lemke-Spielberg as well as the Byrne and Proll procedure. It would be of particular interest to see a systematic comparison of all these techniques with regard to their ability to find a feasible solution quickly. However, I strongly suspect they would nearly always outperform the original Balasian search for feasibility since it was not designed for special efficiency. It would also be interesting to see a comparison of the techniques' capacity to reduce final optimisation times. Other promising avenues of research include applying 0-1 heuristic techniques to general integer programming (e.g. the Senju-Toyoda heuristic in Kochberger, McCarl, Wyman (1973) and the Byrne-Proll heuristic in Bedenbender (1972)), and the efficiency of alternative Balasian codes on specific problem classes (Patterson, 1973).

The main point of my paper is really that a feasible solution is frequently identical to an optimal solution at least in the eyes of a manager, if not in point of fact. Hence there are circumstances where the use of heuristics needs no apology and in fact may be economically superior to the use of optimising algorithms.

References

BEDENBENDER, R. J. (1972). General Integer Programming: A Heuristic Algorithm. Unpublished MBA Paper, Pennsylvania State University.

HOLCOMB, B. D. (1968). Zero-One Integer Programming with Heuristics, IBM contributed program (360D-15.2.011).

KOCHBERGER, G. A., MCCARL, B. A., and WYMAN, F. P. (1973). A Heuristic for General Integer Programming, *Decision Sciences*, (forthcoming).

LEMKE, C. E., and SPIELBERG, K. (1968). DZIP1, Direct Search Zero-One Integer Programming 1, IBM contributed program (360-D-15.2.001).

PATTERSON, J. H. (1973). Zero-One Integer Programming: A Study in Computational Efficiency and a State of the Art Survey, Unpublished Manuscript, Pennsylvania State University.

SALKIN, H. M., and SPIELBERG, K. (1969). DZLP, Adaptive Binary Programming, IBM contributed program (360L-15.001).

To the Editor
The Computer Journal

Sir

I was interested to read in the August issue of the *Journal* the two papers on mixing interpretive and machine code. As you mention, the generation of mixed code is a relatively new topic and my experience with a project in this area may help to supplement the available information. The project concerned the development of a POP-2 compiler on a CDC 6000 Series machine.

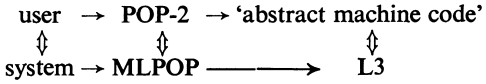
As POP-2 is an interactive language it requires the compiler to co-exist with user programs; thus the system area consisting of compiler and run-time functions and structures is mixed with the user area consisting of user defined functions and data structures. The design philosophy of POP-2, being centred on the unit of the function, allows a 'POP-2 abstract machine' to be defined by a minimum of 10 operation codes. These codes consist of stack and jump instructions. In the original implementation of POP-2 the operation codes were machine instructions or machine extracodes, and thus the compiler generated only machine code. However, for these same codes to be implemented satisfactorily on other machines usually requires that the compiler produces interpretive code. This requirement was almost a necessity on the CDC 6000 machine where it was found that a typical abstract machine code took 2 to 4 CDC words (each of 60 bits) to execute, as opposed to the half word occupied by an equivalent code to be interpreted. The use of interpretive code introduces a factor of about 4 into CPU usage but as this is not an all-important factor in the implementation of an interactive language it was clear that only interpretive code should be produced by the POP-2 compiler.

However, the relationship between the system and user areas is closer than at might first appear. The functions in the system area have a similar structure to the functions defined by the user and as almost all of them can be written in POP-2 they could assume an identical structure and be similarly interpreted. Thus in practice the line between system and user is arbitrary and the question arises as to whether system functions should be in machine code or interpretive code. The method chosen to implement POP-2 on the CDC allows a choice to be made.

The solution to the problem of transition areas is that adopted by most POP-2 compilers. As the basic unit of execution is the function, this is also taken as the unit for use of code type, and all entries to and exits from the interpreter are made in small pieces of machine code on function entry and exit. This approach means that there are no overheads (greater than usual) for running only machine code. Calling a machine code function from an interpretive code function causes an extra address to be placed on the stack. This address (the return link) is an address in the interpreter itself and thus exit from the machine code function automatically enters the interpreter. While this approach is obvious and elegant it may not be selective enough.

The generation of mixed system code followed a similar approach to that of Dakin and Poole with one significant difference. Using a macro-processor (ML/1) a number of macros were created which defined a high level system language remarkably similar in many respects to POP-2 and named MLPOP. The result of the text translation process was a low level language (christened L3) which in many respects was similar to an extended POP-2 abstract machine. The L3 operations were simple enough to be defined as macros by the CDC macro assembler, and the choice could be made to generate machine code or interpretive code. Thus while the system portability is maintained at the high level, the code type selection and generation is kept at a low level.

The relationship between the user and the system area has already been pointed out and it is interesting to see how this relationship is maintained at all levels as illustrated in the diagram.



With the significance of the development of microprogrammable computers pointed out by Dawson this relationship at the abstract machine level suggests that POP-2 will be an ideal candidate for the benefits of microprogramming.

Yours faithfully
K. J. MACCALLUM

43 Woodland Gardens
Isleworth
Middlesex
24 September 1973

To the Editor
The Computer Journal

Sir

As a member residing in the US, I get no favours. The August issue has reached me on October 10. For this I have no complaint; for your

editorial note (in that issue) regarding mixed interpretive and machine code—it happens that I do.

As a brother editor, I feel that we are often handicapped by too little application of our tool to our own profession. I have toyed with the idea of making an experiment, in such an uncomplicated field as algorithms for computation of square root, where in a bibliography of papers on the subject would be annotated (preferably by the author) by a summary of what the germ of the contribution was, and what its advantage was over previous methods. Thus the aspiring author of yet another paper on square root computation would have this document with which to assess his own temerity in coming out with a further contribution.

Obviously no such bibliography exists for the interpretive mode of computation. Even Jean Sammet's book does not provide such information for programming languages. One of your authors references a paper in 1967; the authors of the other paper quote five from 1970 on, four of which were by the author or his associate.

Perhaps this led you to term it a *new* idea. In fact, this mixed mode existed in the PRINT I system for the IBM 705, which software system was delivered to the field in the summer of 1956 (for emphasis, this was 17 *years ago*). Floating point computation, flow, and formatting were done interpretively for the reasons of economy stated in your first paper. I/O and such were in machine code, as generated by the assembly language AUTOCODE.

The combination was most effective, thus supporting the arguments of your authors. In fact, it competed well against 704 FORTRAN, which machine was certainly more oriented to scientific computation than was the 705, a business machine. I recall with glee that my interpretive division routine, programmed in machine language with a Newtonian iteration to get the desired precision and accuracy, was actually faster than the single division command built into the hardware, to no little consternation of the hardware engineers.

As a note to history, the PRINT I system (PRINT stood for Pre-edited Interpretive) was the first load-and-go compiler to operate, as far as I know.

Yours faithfully,

R. W. BEMER

Editor
Honeywell Computer Journal
Honeywell Information Systems Inc
Advanced Systems and Technology
Deer Valley Park
PO Box 6000
Phoenix
Arizona 85005
USA
11 October 1973

References

BEMER, R. W. (1956). PRINT I—A proposed coding system for the IBM type 705, *Proc. Western Joint Comput. Conf.*

BEMER, R. W. (1956). Reference manual, PRINT I system for the 705, IBM Corp.

BEMER, R. W. (1956). PRINT I—An automatic coding system for the IBM 705, Automatic Coding, Monograph No. 3, Proc. Automatic Coding Symposium, Franklin Inst., pp. 29-38.

To the Editor
The Computer Journal

Sir

In his interesting article on Himmelbett Mr. Bell states he knows no reference for 'Space War'—may I give him one:

ALBERT W. KUHFELD (1971). Space War, *Analog*, Vol. LXXXVII, No. 5, pp. 67-79.

Analog may not have the scientific reputation of the majority of journals quoted as references—it does, however, have a very wide distribution amongst engineers, scientists and computer personnel.

Yours faithfully,

K. FREEMAN

Vice Chairman
The British Science Fiction Association Limited
128 Fairford Road
Tilehurst
Reading RG3 6QP
14 October 1973

To the Editor
The Computer Journal

Sir

Ternary Logic in Parallel Multipliers

In a recent paper Vranesic and Hamacher (1972) discuss the relative speed and cost of binary and ternary multipliers. A speed improvement and some simplification of the design are possible if balanced ternary arithmetic is used.

Suppose the units are $-$, 0, $+$ and both true and complement forms of the operands are available. The multiplier bits are not grouped but each selects the multiplicand, its complement, or zero. The adder is not required. In the 16 × 16 digit example discussed summand selection will take 2τ (originally 11τ).

More blocks will be required in the carry save adder tree, but in this particular example no more levels are required and the time taken remains 15τ. The full adder cells are identical in design as each input is weighted by one requiring the output to be weighted by four; i.e., by one in both sum and carry.

The adder must be redesigned as both positive and negative carry one possible. Positive and negative propogate and generate terms are required and these can be formed in 3τ. Positive and negative carries can be generated separately in 4τ and applied consecutively to the sum in a further 2τ. The total adder delay is 9τ (originally 8τ).

The complete multiplier delay is therefore 26τ, substantially better than the 34τ of the original design and slightly better than the 27τ of the equivalent binary multiplier. However for most operand sizes the carry save adder tree has one more level than in the original design. For most operand sizes in the range studied the balanced ternary multiplier is 3τ faster than the original ternary design and 1τ slower than the equivalent binary design.

The cost of the summand selection logic (counting either gates or inputs) is approximately half that of the original design. The cost of the carry save adder tree is approximately doubled, and the adder is approximately 50% more expensive. A balanced ternary multiplier is thus very similar in cost, as well as speed, to the corresponding binary multiplier.

Yours faithfully,

A. G. BROMLEY

Basser Department of Computer Science
School of Physics
University of Sydney
Sydney
New South Wales 2006
Australia
25 October 1973

Reference

VRANESIC, Z. G., and HAMACHER, V. C. (1972). Ternary Logic in Parallel Multipliers, *The Computer Journal*, Vol. 15, p. 254.

Professors Vranesic and Hamacher reply:

The main point of Dr. Bromley's letter, that there is a speed improvement in using balanced representation without multiplier digit grouping, is correct at most operand lengths. However, we do not agree with all of the quantitative comparisons. Our detailed comments can conveniently be grouped in terms of the three multiplier subsystems:

1. Summand selection: Using the same type of multivalued algebra and operand forms as assumed in our paper, summand selection in the balanced ternary design would take 4τ, not 2τ. If we had assumed that certain unary functions of the multiplier digits were available (equivalent to a zero-delay digit decoding function), our summand selection would have taken 9τ instead of 11τ. Thus, for consistency, we will assume a 4τ value for the balanced ternary design and 11τ for our design.

2. Carry save adder tree: Although the number of levels (three) in the tree is the same in the 16 × 16 digit case, there is at least one case, 21 × 21, where the balanced ternary tree requires five levels while our design still only requires three levels, a difference of 10τ.

3. Final adder: We agree with the 9τ balanced ternary adder and its associated relative cost.

In general, we agree with the cost comparisons made by Dr. Bromley. However, consolidating our timing comments, and accounting for a small extra adder delay in our 21 × 21 case, we feel that the correct 16 × 16 and 21 × 21 total delays should be 28τ