From: Eric Bush and Peter Krupp

To: MTB Distribution

Date: April 30, 1981

Subject: Version 3 PL/I

1. INTRODUCTION

This memo presents an overview of the proposed design of a new code generator and associated optimizers for the Multics PL/I compiler. It is motivated by the fact that substantial changes to our PL/I compiler will be needed to insure adequate performance on ORION.

The present conjunction of ORION requirements, problems with the maintenance of the current code generator, the opportunity of achieving optimizations as good as those of the Multics FORTRAN compiler, and the availability, in the literature, of new code generator techniques make this an opportune time to begin a staged redesign and reimplementation of the Multics PL/I compiler.

2. REASONS FOR REDESIGN

There are three reasons:

1) The primary impetus for the redesign effort is the advent of ORION. The new hardware will both require that the code generator be able to generate ORION-specific instructions, and present the opportunity for generating ORION-optimized instructions. An example of the former is the necessity to use the lprin instructions instead of eppn instructions. An example of the latter is the opportunity to rearrange the order of generated instructions in order to minimize expected pipeline breaks.

2) Once we open up the code generator for redevelopment, though, it behooves us to look for opportunities to generate better code in general, taking advantage of techniques that have shown up in the literature since the present code generator was written. The effort spent on general improvement should, of course, be commensurate with the time and resources available, but since most of Multics is dependent for its peformance on the ability of this code generator to produce efficient code, the beneficial effects of better code can be

widely felt.

3) Perhaps the greatest need for redesign is to make the code generator more easily maintainable. The "conceptual" functions of the code generator are spread accross many individual procedures, many of whom depend for their proper functioning upon intimate knowledge of the workings of other procedures. Consequently, bug fixes or enhancements which are readily understood in the abstract, can be overwhelmingly difficult to implement. Recent work on table driven techinques for code generators similar to the LR(k) technology for parsing suggest ways to improve this situation substantially. (More on this later.) Given that most of the many bugs now outstanding on the PL/I bug list are code generator bugs, we are obliged to spend considerable resources on code generator work anyway. Better that these resources be spent on activities that help to reduce future resource consumption, than on projects that perpetuate or even increase it.

3. DESIGN PRINCIPLES

3.1 Data structure abstraction:

A compiler can be viewed in the abstract as a series of modules or "phases" that perform successive transformations on some representation of the program being compiled. With the exception of the modules at the extreme ends, those which turn source into internal representation and those that turn internal representation into object code, most of these modules are making tree to tree transformations, the internal representation usually being a tree. Other forms of internal representation show up (e.g.,graphs and "quads"), but trees dominate.

Human-readable descriptions of what compiler phases are up to (in the literature, in plm's, in mtb's, in blackboard discusssions, etc.) are naturally couched in terms of tree (or other structure) transformations. How nice it would be if compiler phases could be written and rewritten at this same level of abstraction. Since ours wasn't (whose was ?) we are saddled with two unhappy obstacles to local code changes (bug fixes, enhancements, etc.): one must know the architecture of the current internal tree down to the level of pointers and records, and one cannot change that architecture, even in small ways, without determining and fudging all the procedures that currently rely on its present structure. And the more one patches, the less readable the original procedures become.

We thus set as a design goal, for all of the tree-transforming modules that we write or rewrite, to express them at the level of tree transformations, and to keep them

ingnorant of the implementation details of those trees. We propose to accomplish this by employing a macro processor to write access functions to the current form of the tree. New source code will speak directly of tree-munging by employing the macros. Changes to the implementation of the tree will then involve only changes to the access functions. Since macros compile into in-line code, the modularity of procedures calls is enforced without the attendant runtime overhead.

3.2 Modularity:

Another property of the best of all possible compilers, is the modular independence of the code embodying the knowledge about how to compile in general and code embodying the knowledge about how to compile a particular language for a particular machine. A good example of this comes from the now commonplace LR(k) technology. Knowledge of how to parse (LR(k) languages) in general is embodied in the programing language in which the parsing algorithm is written. Knowledge of the grammar of the language that is being parsed is embodied in tables on which the parsing algorithm operates. The parser can parse a different language merely by using a different table. What's more, the tables can be automatically produced from highly abstract grammar descriptions, freeing the maintainer form the details of implementation and issues of compatibility.

We would like to get our share of this new technology for code generation too. As an ideal we would like all knowledge about the semantics of PL/I and about our current machine architecture to be embodied in tables on which a geneneral purpose code generator works. The tables would be generated from highly abstract specifications of language and machine semantics. It is not clear that we can completely achieve so lofty a goal in the time available, but it is a worthy end at which to aim. A code generator that is partially language and machine independent is better than one that is not so at all.

3.3 Staged implementation:

Given the very central role that the PL/I compiler plays in the life of Multics, a slow, hard to maintain code generator that produces correct code is preferable to a state of the art model that doesn't. And given the great complexity of the PL/I language, constructing a correct code generator for it out of whole cloth is not a task to be taken lightly. It seems desirable, then, to try to graft the new back-end onto the old front-end one module at a time so that correctness can be checked in increments. We can't tell at this time what the smallest retrofitable unit is, nor what overhead will be involved in trying to make new modules function in old environments, but we can identify phased implementation as a desirable goal and approximate to it as best we can.

4. OVERVIEW OF DESIGN

We propose to divide the back-end redevelopment into two logical phases: code generation and optimization. The plan is first to achieve correct, though not necessarily optimal, code generation by developing the various phases of the code generator and integrating them into the current compiler. Once correct code can be produced, we will develop the optimization phases.

We conceive the new code generator as itself consisting of two logical phases: register allocation and instruction selection. Other phases of code generation, eg, storage allocation, listing generation, object segment preparartion, will be retained from the current compiler. Their interfaces will probably have to be modified. The design we have in mind for table-driven instruction picking seems to require the intelligent cooperation of a register allocator to produce good code, i.e., one that does its job in light of the knowledge embodied in the same tables used by the instruction selector. We thus expect that the design of the instruction selector will strongly influence the design of the register allocator, though they may be implemented and tested separately.

Optimization will also consist of two phases: a global optimizer that will perform various improvements on the program tree, in light of global data flow analysis, prior to code generation, and a peephole optimzer that will effect more local improvements on the instruction sequences produced by code generation. Although the code generator should be able to run initially without the peepholer, our current view of the code generator indicates that to reduce the size of its tables to a managable magnitude, we may have to purposely let it generate inelegant sequences of code that only a peepholer can fix. If this turns out to be true, some form of peepholer may be a necessary concommitant of any acceptable code generator.

The next four sections address themselves to each of these four phases in more detail.

5. GLOBAL OPTIMIZER

5.1 Current compiler:

Code optimization in the current compiler is done both explicitly in an optional phase (when given "-optimize"), and behind your back in disparate locations in various phases.

The explicit optimizer phase does common subexpression elimination within basic blocks and pulls loop-invariant computations out of the loops created by the semantic translator. Since it does not build a global flow graph of the program it cannot recognize loops coded by the programmer and common subexpressions occuring across basic block boundaries and thus cannot do the appropriate code motion and redundant code elimination. It recognizes its own loops only because the semantic translator hangs them from a "loop" operator when it creates them.

The code generator has some appreciation of dead variables through the use of reference counts that are set and reset by various modules in various places. Keeping track of who does what with these is one of the more difficult problems confronting the maintainer.

Quick procedure optimization is done by a special module in the code generator.

5.2 Classical approach:

The classical approach to global optimization is to build a flow graph from some low level form of the program, compute data flow quantities for each node by iterativly running around the graph, and then improve the program by rearranging, consolidating and eliminating code in light of invariances revealed by the data flow analysis. The standard improvements go by various names but can be put into three classes: redundant code elimination (removal of code to compute a value that was computed or made available previously), dead code elimination (removal of computations or storage movements that do not materially effect the output of the program), and strength reduction (substitution of cheaper (i.e., faster or smaller) computations and resources for unnecessarily expensive ones).

5.3 Recent work:

The recent literature on optimization offers two sources of improvement. 1) Various new techniques have been developed to deal with procedure calls and pointer based variables, two contructs, ubiquitous in PL/I programs, that have traditionally thwarted data flow analysis. 2)Techniques for doing data flow analysis on more high level forms of control graphs have been developed which considerably speed up the time it takes to compute data flow quantities.

5.4 Our approach:

We could significantly improve the code generated by the current compiler just by doing classical data flow analysis. The new global optimizer should do at least this much.

The internal tree produced by the current semantic translator has had most of its high-level control structures (like "do while") removed and thus may not be suited to the rapid data flow techniques currently found in the literature, but since, via our macro access function strategy, we may view the internal tree in whatever form we wish (given that our preferred form can be computed from the old form), we may be able to view the tree at a higher level for the purposes of rapid data flow. It appears, from our initial study, that macro access functions can reconstruct the original high level constructs from the current low level ones. If this is possible we should take a shot at it.

Techniques to preserve data flow information across procedure calls and pointer indirection would seem to be optimizations that could pay big dividends for Multicians, given that we use lots of procedures and pointers, but there is still a lot of room for pioneering here. There is no well- entrenched, standard way of handling these things at present. This will probably have to be a topic to be taken up if time permits.

6. REGISTER ALLOCATION

6.1 Current compiler:

Presently register allocation is done by the instruction selector itself rather than in a separate module. Registers are assigned locally as needed. Fixed binary arithmetic results go into the Q or AQ. Float binary arithmetic results go into the EAQ. Some attempt is made to keep user index and pointer variables in index and pointer registers respectively. There is no attempt to keep busy loop variables in registers during loop execution, and all live user values are stored and loaded between basic blocks, since the global flow data for doing otherwise is not available.

6.2 Classical approach:

The approach generally recommended in the literature is to reserve some registers for special purposes (stack pointers, return addresses) and use the rest to hold the most referenced of a users variables during inner loop execution. This saves (on the average) one load and one store times the number of times around the loop for each variable so "registered".

6.3 Recent work:

Recent studies recommend a more global approach to allocating registers. The register allocator has access to all of the global data flow analysis that the global optimizer gets (and perhaps more). From this the "lifetimes" of all program variables (use's and compiler's) are determined. The variables are assigned preference weights indicating the relative desirabilty of having each in a register (and perhaps also the type of register, if there is a choice). An attempt is then made to assign all variables to the available registers according to "bin-packing" algoriths, in the order of greatest weigth. Variables whose lifetimes do not overlap can be multiplexed into a single register.

6.4 Our approach:

To some extent, our hardware prevents us from keeping strategic user variables in registers because we don't have many general purpose full-word registers , and the two that we do have (the A and the Q) can't be used as operands to the same instruction. Both the recent and standard studies tend to assume a "general purpose register" architechture like the IBM 360/370. We can, however, exploit our multiple index and pointer registers if the conditions are right (i.e., for pointer values, and integers whose precision fits in the indexes). Since we don't do anything like this now, and since the FORTRAN optimizing compiler has apparently achieved good results by thus playing around with the index registers, we expect that the recent approaches to global allocation would be worth our while. It appears likely that the elimination of many pointer load instructions (by a register allocator using information supplied by a global optimizer) will buy much more performance on ORION than any amount of instruction re-ordering.

7. INSTRUCTION SELECTION

7.1 Current compiler:

Code generation in the current compiler is done by simulating the execution of program statements down all possible control paths and by simulating the evaluation of expressions. The result of this simulated execution is a sequence of instructions that carry out the intent of the program when executed by the processor.

The simulation is carried out by a large number of PL/I procedures which comprise the PL/I compiler code generator. Each procedure handles a particular subtask of the simulation. For example, a procedure called arith_op simulates the evaluation of

real binary arithmetic expressions, another procedure handles complex binary expressions. The procedure, arith_op, is large, though not the largest in the code genrator, because of the immense amount of case analysis that must be done even for relatively simple arithmetic expressions. As a result of its size and complexity, a large number of bugs have been found and fixed in this procedure.

The main cause of complexity and bugs in the code generator seems to be the immense amount of case analysis that must be performed to select efficient and correct instruction sequences. However, even if the amount of case analysis could somehow be reduced, the lack of modularity of this approach to code generation makes it unacceptable from an economic point of view. Knowledge of the target machine and the PL/I language is embedded in the large number of procedures that comprise the code generator. Retargeting the code generator for new machines is impossible and even the simplest change to the processor can require global changes to the code generator. Thus, the usual approach to retargeting is reimplementation.

7.2 Classical approach:

The classical approach to code generation resembles the Multics PL/I approach. The intermediate representation of the program being compiled is scanned and the code generator simulates execution of the program by constructing a sequence of instructions which implements the program on the target machine. The difference lies not in the algorithm but the data structures. Classically, the intermediate representation is linear: quads, triples, reverse polish are all examples. Linear representations were favored in early compilers because of memory limitations. At any one time, only a small segment of intermediate text needed to be in memory.

Memory, in our case, is not as important of an constraint and so we favor the less restrictive tree intermediate representation. Multics PL/I uses the tree representation; this gives it more freedom in selecting the order of evaluation of expressions.

7.3 Recent work:

There has been a decent amount of research recently aimed at reducing the code generation process to a table-driven process, much as the "parsing process" has been so reduced by LR(k) technology. Briefly, the approach is to write a general purpose instruction selector which attempts match portions of the program tree against a repertoire of small tree patterns which represent machine instruction sequences. A successful match causes the represented instructions to be generated. A successful covering of the whole tree produces a machine language

program for it. The repertoire of tree-pattern/instruction-sequnce pairs is supplied to the instruction selector as a table. The appropriate table for a given machine/language combination is produced by a pre-compiler faciility which takes abstract descriptions of both machine and language as input, much as a parser generator takes a BNF grammer description as input to produce parsing tables for that language it describes.

7.4 Our approach:

As mentioned above, we think that table-driving is the way to go, particularly for a language so voluminous as PL/1. Perhaps the greatest gain we can expect from such an approach is the relative ease of maintaining such a code generator. Changes to the language, the machine, or to particular implementations of the language can be made at the level of the abstract inputs to the pre-compiler facility. This both speeds up mandated fixes, and allows for relatively easy and inexpensive experimentation with alternate implementation idioms. Differences between optimized and unoptimized compilation as well as differences in compilation for different machines (L68 vs ORION) can be simply a difference of tables used.

Our primary source from the literature has been the PQCC project (Production Cuality Compiler Compiler) at Carnegie Mellon headed by Bill Wulf. We were initially concerned that the design for the code generator, from which we have taken many ideas, had not actually been implemented at the time of its publication (1978), and that the language that it was modeled on (BLISS) is not of the same order of complexity as PL/I, but we've just heard from Wulf (via his recent lecture at MIT) that the design has been used to successfully implement an Ada code generator. Thus we now have empirical confirmation of the soundness of the design for a PL/I-like language.

8. PEEPHOLE OPTIMIZER

8.1 Current compiler:

Peephole optimization is done by the code generator in the Multics PL/I compiler. The code generator examines every instruction that it emits and determines if it can be combined with some preceding instruction. The code generator also performs strength reduction: it replaces expensive instructions with more efficient special case instructions where ever possible. A classic example is the combination of two left shift instructions into one left shift instruction. Another example, would be the replacement of an instruction which multiplies the A

by a power of 2 by a shift instruction.

Two basic problems exist with the current peephole optimizer. One problem is safety. The peephole optimizer deletes instructions and does so without examing the control flow of the program and without doing any live/ dead variable analysis. If an instruction which is the target of a jump is deleted, a bug is introduced into the object program. It gets away with these optimizations most of the time because of its intimate knowledge of the preceding phases of the compiler. If changes to those phases are made which violate its assumptions, incorrect object code is generated.

The other problem is efficiency. Every instruction emitted is considered to be a candidate for every possible peephole optimization. This brute force approach has acceptable performance only because of the relatively small number of special cases considered, approximately 30-40. Improvements can be made.

8.2 Classical approach:

The Multics PL/I compiler implements the classical approach. Classical peephole optimizers have a catalog of tricks and optimizations and examine the object code instruction by instruction and improve it wherever possible. Knowledge about the target machine is embedded in the code of the peephole optimizer again implying that retargeting entails reimplementation.

8.3 Recent work:

As in instruction selection, the trend is towards machine independent algorithms and machine dependent tables. The table contains pattern/action pairs. The algorithm matches the pattern against the instructions and takes the corresponding action whenever the pattern matches and object code sequence. The action is usually to replace the instruction sequence with a more efficient one.

8.4 Our approach:

We are leaning towrds the general pattern matching approach. But since building a peephole optimizer is relatively simple compared to the other tasks before us, we may adopt the special case approach. This will depend on our success with producing a quality code generator. If that can be done without making the compiler inefficient, then the peephole optimizer will have little to do in the classical area of peephole optimization.

Another task, which is not done by classical peephole

optimizers, is that of instruction scheduling. To achieve high quality object code it may be necessary to do instruction scheduling to minimize pipeline breaks on ORION. The peephole optimizer is the correct place to do this since instruction scheduling is strictly machine dependent and requires the same information and same operations as that of peephole optimizations.

9. IMPLEMENTATION ORDER

In keeping with the above goal of incremental development, we would like to subdivide the total redesign effort into smaller, independently testable units so that we have some measure of "correctness so far" as we are proceding. It seems possible that versions of the four phases just described could be developed and tested independently, i.e., we cound graft some version of any one of them into the current compiler and test its correctness without the presence of the other three. It seems clear, though, that the "desirable" version of any one of them depends on the presence of one or more of the others. We can think, for example, of the following dependencies right off:

Global optimization: what data flow quantities are computed depends on who needs what in later phases.

Register allocator: availability of global live-dead analysis (global optimizer) allows better use of registers; choice between types of registers will prejudge options for instruction selection; pipeline scheduling (peephole optimizer) is affected by contigious use of the same register.

Instruction selector: can minimize his own size and speed if peepholer is smart enough to cleen up redundancies, and register allocator has picked registers with potential instructions in mind.

Peephole optimizer: needs global live_dead analysis to do code motion for pipeline scheduling; needs coordination with register allocator to minimize contigious register usage; needs to know what mistakes instruction picker is most likely to make.

We might classify the interdependencies between modules as two kinds: those (as with the global optimizer) where the design details of one module depend on the design details of

others, and those (as with the instruction selector) where the performance or output quality (as opposed to output correctness) of one module depends on the cooperation of other modules. It seems counterproductive to implement stand-alone versions of modules afflicted by the first class of dependency, for they must then undergo redesign when their fellows arrive. The second class of dependency, however, looks more benign. We presume, in such cases, that the addition of new surrounding modules will not effect the design of the original module but merely its performance. For example, it appears that the instruction selector requires merely some form of register allocation (that , for instance, that is already done by the current compiler)and no form of peephole optimization to produce correct code. Thus it seems possible to implement and test the instruction selector prior to development of the other two modules. Without redesign, the same instruction selector should produce better code when it is fed a better configuration of registers and its output is edited by a peepholer.

The instruction selector indeed seems to be the module whose design is least influenced by the design of other modules and thus presents itself as a natural place to start the incremental development. It is also the module whose design is most pioneering relative to the literature, and thus the one for which we would most like an early confirmation of correctness.

Since both the register allocator and the peephole optimizer require global data flow analysis, it looks like development of the global optimizer should come next.

Which module is next, or whether the last two are done concurrently is too hard to tell at this early stage. The choice of order may well be made on the basis of release dates or other external considerations. Evidence from the FORTRAN compiler suggests both a higher benefit and a higher cost for the register allocator. In general, our current conceptual horizon begins to fade at the boundaries of the instruction selector. We will have better and more detailed opinions about other modules as we develop the details of this one.

10. BIBLIOGRAPHY

The following bibliography is intentionally brief. In most cases, the reference cited contains a rather extensive bibliography on its own subject. Readers wishing to persue a given topic further should find these references to be good starting points.

Global optimization:

Classical:

Alfred V. Aho & Jeffrey D. Ullman, PRINCIPLES OF COMPILER DESIGN (Addison Wesley, 1979), pp. 509-517.

Recent:

Ronald Mintz, Gerry Fisher, Micha Sharir, "The Design of the PDL Optimizer," (unpublished).

David B. Loveman, "Program Improvement by Source-to-Source Transformations," JOURNAL of the ACM Vol.24, no.1 (January 1977), pp. 121-145.

Barry K. Rosen, "Monoids for Rapid Data Flow Analysis," IBM Research Report RC 7032 (no.30111), IBM Thomas J. Watson Research Center, Yorktown Heights, NY (1978).

Rosen, "Data Flow Analysis for Procedural Languages," JOURNAL of the ACM Vol.26, no.2, (April 1979), pp. 322-344.

Register allocation:

Classical:

Aho & Ullman, pp. 533-537.

Aho, S.C. Johnson, Ullman "Code Generation for Expressions with Common Subexpressions," JOURNAL of the ACM Vol.24, no.1, (January 1977), pp. 146-160.

Recent:

Richard Karl Johnsson, "An Approach to Global Register Allocation," (PhD thesis, Carnegie-Mellon University, December 1975).

Instruction selection:

Classical:

Aho & Ullman, pp. 518-556.

Recent:

William A. Wulf, "PQCC: A Machine-Relative Compiler Technology," Technical Report CMU-CS-80-144 (Carnegie-Mellon University September 1980).

Susan L. Graham, "Table Driven Code Generation," IEEE COMPUTER Vol.13, no.8, (August 1980), pp. 25-34.

R.G.G. Cattell, "Automatic Derivation of Code Generators from Machine Descriptions," ACM TRANSACTIONS on PROGRAMMING LANGUAGES and SYSTEMS Vol.2, no.2, (April 1980) pp.173-190.

Cattell, "Formalization and Automatic Derivation of Code Generators," (PhD thesis, Carnegie-Mellon University, April 1978).

Robert S. Glanville, "A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers," (PhD thesis, University of California, Berkeley, December 1977).

Joseph M. Newcomer, "Machine-independent Generation of Optimal Local Code," (PhD thesis, Carnegie-Mellon University, May 1975).

Peephole optimization:

Classical:

Aho & Ullman pp. 548-556.

Recent:

David Alex Lamb, "Construction of a Peephole Optimizer," Technical Report CMU-CS-80-141 (Carnegie-Mellon University February 1980).

J.W. Davidson and C.W. Fraser, "The Design and Application of a Retargetable Peephole Optimizer," ACM TRANSACTIONS on PROGRAMMING LANGUAGES