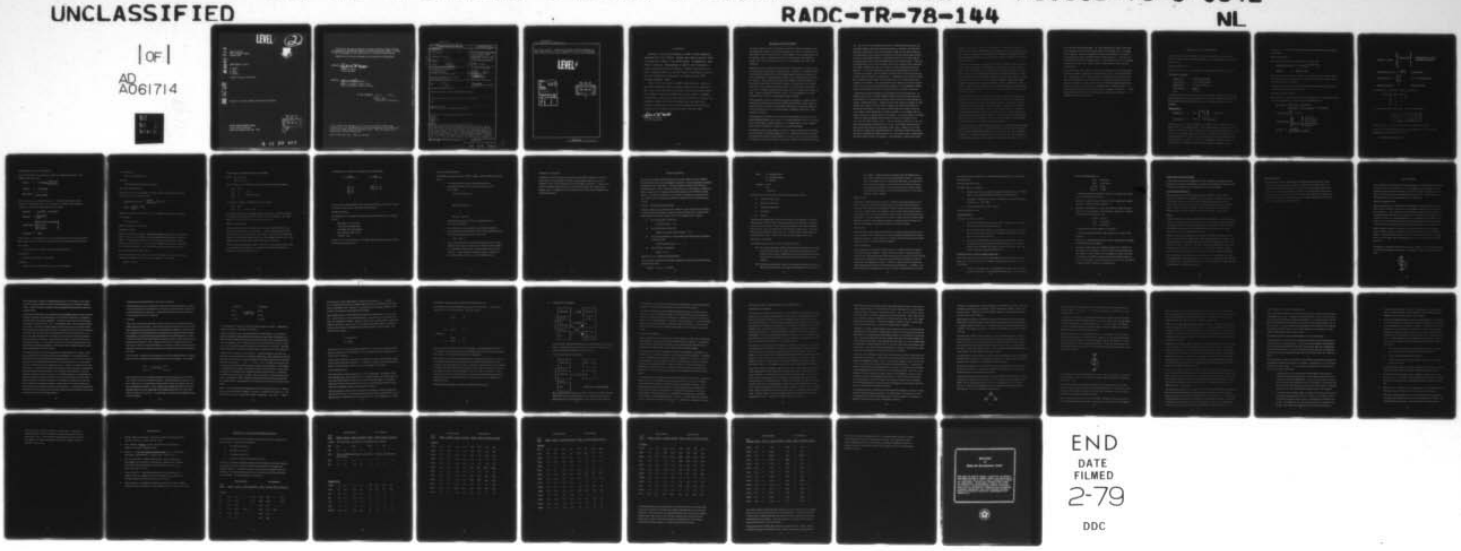
AD-A061 714 COMPUTER SCIENCES CORP EL SEGUNDO CA F/G 9/2
IMPROVEMENTS TO JOCIT.(U)
OCT 78 T DEVINE, L HYDE, H MCCOY, R RUSHALL F30602-76-C-0342
UNCLASSIFIED RADC-TR-78-144 NL

1 OF 1
AD A061714



END
DATE FILMED
2-79
DDC

LEVEL II

B.S.

2

AD A061714

RADC-TR-78-144
Final Technical Report
October 1978

IMPROVEMENTS TO JOCIT

T. DeVine
L. Hyde
H. McCoy
R. Rushall

Computer Sciences Corporation

DDC FILE COPY

Approved for public release; distribution unlimited.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-78-144 has been reviewed and is approved for publication.

APPROVED: Richard M. Motto

RICHARD M. MOTTO
Project Engineer

APPROVED: Wendall C Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-78-144	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): IMPROVEMENTS TO JOCIT,	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, Jul 76 – Mar 78
6. PERFORMING ORG. REPORT NUMBER: N/A	
7. AUTHOR(s): T. DeVine, H. McCoy, L. Hyde, R. Rushall	8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0342
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Sciences Corporation, 220 Continental Blvd, El Segundo CA 90245	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63728F, 55500839
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: October 1978
	13. NUMBER OF PAGES: 44
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Richard M. Motto (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

JOVIAL
Compilers
Optimizers
SYMPL
GENESIS

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Although the JOVIAL Compiler Implementation Tool (JOCIT) produced the most "J3-like" JOVIAL compiler built to date, the Air Force was not at liberty to develop compilers for other systems because of the proprietary nature of the JOCIT module, GENESIS. The intent of this effort was to lease GENESIS and to make various improvements to JOCIT. These improvements included fortifying the JOCIT optimizer with state-of-the-art optimizing features. For the most part, this effort accomplished the above, but resulted in the development of

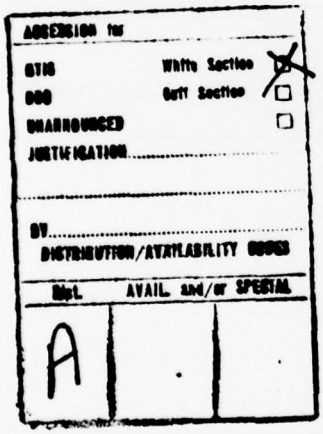
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

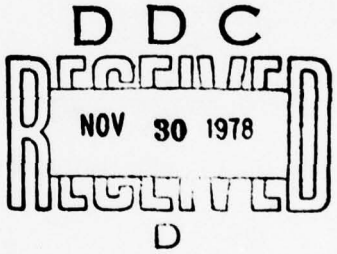
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

two separate optimizers. Additional improvements included optimizing the SYMPL compiler module of JOCIT and implementing an ASCII and floating point double precision data processing capability in the JOCIT produced compilers.

LEVEL II





EVALUATION

Because of the wide acceptance of RADC's JOVIAL Compiler Implementation Tool (JOCIT), demands were made on the Air Force to secure full rights to the JOCIT system. Consequently, this effort, entitled, "Improvements to JOCIT", was initiated to lease the proprietary meta-compiler, GENESIS, giving the Air Force complete control of the tool and to incorporate state-of-the-art optimization technqiues into JOCIT and the JOCIT building language, SYMPL.

This effort fell short of RADC's expectations, insofar as two JOCIT optimizers were delivered instead of one; however, a DOD "first" was realized in creating a 99-year software lease with Computer Sciences Corporation for the rights to GENESIS. In addition to the above, this contract served as a vehicle for producing double precision floating point and ASCII data handling capabilities for the JOCIT system and initiated the updating of the JOVIAL J3 specification, MIL-STD 1588.

Richard M. Motto

RICHARD M. MOTTO
Project Engineer

INTRODUCTION AND SUMMARY

The "Improvements to JOCIT" contract was awarded to improve the operation and versatility of the JOVIAL Compiler Implementation Tool (JOCIT) so that it would be more practical and conducive to use in a central Higher Order Language (HOL) control facility. JOCIT is operational at Rome Air Development Center (RADC) at Griffiss AFB, New York and permits the generation of high quality JOVIAL (J3) compilers.

Initially the contract called for the transfer of the GENESIS Meta-Compiler to operate on the RADC HIS 600/6000 computer complex. GENESIS is a Computer Sciences Corporation proprietary system for producing language syntax tables suitable for use in certain types of syntax-directed compilers from a convenient syntax description language. Also, the SYMPL compiler (the language in which the majority of the modules in the JOVIAL and SYMPL compilers are written) was to be modified so that more of the computer independent portions of the compiler resided in the "front-end". Additional optimization techniques were to be added to the SYMPL compiler. Finally, optimization techniques such as "code-straightening" and "dead-variable analysis" were to be added to the JOVIAL compiler.

Subsequently the contract was modified to include the addition of a double precision floating point capability not previously available in the JOVIAL compiler and to rewrite the JOVIAL (J3) MIL-STD-1588 dated 30 June 1976, to reflect the double precision floating point format and all other updates resulting from the JOCIT implementation of JOVIAL.

The installation of GENESIS was accomplished with virtually no problems and will not be discussed further in this report. The rewrite of the MIL-STD-1588 was much more extensive than originally anticipated. The changes were of such a magnitude that the entire document was renumbered and completely retyped.

The changes to the JOVIAL optimizer were more complex and required considerably more effort than was initially thought necessary. Originally the plan was to add the new optimizations into the existing optimizer. However, after some study, this plan was abandoned in favor of producing a new optimizer which would replace the existing

one. This decision was partially based on the fact that the existing optimizer was somewhat unstable, particularly with large programs. Therefore it was believed that major modifications to the existing optimizer might increase the instability to an unacceptable level and require more effort to correct the situation than would be expended in rewriting the optimizer. Rewriting the optimizer was begun with the new optimizations such as code-straightening and dead-variable analysis. After proceeding with this plan for several months, it became increasingly obvious that there was neither enough time or money to complete an entire replacement for the old optimizer. In order to provide the maximum possible optimizations for JOVIAL, a new plan was adopted. The old optimizer would be left as it was and the new optimizations would be added as a separate phase. Additionally it was decided to permit the old optimizer phase and the new optimizer phase to be run optionally and independently of one another. The only restriction to the optimizers is that the old optimizer, if it is run, must run before the new optimizer. Even adopting this method required considerably more time and manpower than was originally contracted.

A considerable amount of testing was accomplished on the new optimizations. In addition to the JOVIAL Compiler Validation System (JCVS) tests, there were fourteen Strategic Air Command (SAC) programs which could not be run through the old optimizer without fatal errors. At the conclusion of the optimizer testing, all of the fourteen SAC programs had at one time or another been successfully optimized through the new optimizer. Also, twelve of the fourteen SAC programs had been successfully compiled through the old optimizer thanks to some changes made by a former employee of Computer Sciences Corporation. It was not possible to do testing on the SAC programs with the final compiler delivered, because they were deleted from the RADC Computer System. The new optimizer handled all the JCVS tests with the exception of the large Class 1 tests. Five of these six tests had problems, primarily in the area of limitation of table space. However, two of these five tests do compile when using both optimizers. Appendix A of this report shows the time of compilation and object program size of the various modes of optimization for the JCVS tests and the SAC programs. Generally there is improvement in the

optimized program sizes with the new optimizer. There remain, however, a number of improvements which would add to the efficiency of the compiled programs. These include completion of redistribution and the addition of strength reduction, test replacement, variable overlay, register allocation and generalized name definition/use analysis.

A number of modifications were made to the SYMPL compiler to improve its operation. The most noticeable was changing this compiler from two phases to three phases. This reduced the core size required for a SYMPL compilation from 43K to 36K. At the same time new optimizations were added and the compiler was made more portable by rearranging machine dependent code into fewer modules. Testing of the new SYMPL compiler was very extensive, including compiling the SYMPL modules through itself to ensure the compiler was stable. Additionally, all the JOVIAL modules written in SYMPL were recompiled and linked into a test compiler that executed the JCVS tests as well as the JOVIAL compiler produced by the old SYMPL compiler.

The final change to JOCIT accomplished under this contract was the addition of a double precision capability to the JOVIAL compiler. This also necessitated the recognition of double precision by the SYMPL compiler. Changes were required throughout the JOVIAL compiler including the analysis, allocation, direct code, code generator and editor phases as well as the run-time library. Testing was accomplished by temporary changes to the JCVS tests. The results of these tests indicate that the double precision capability is now a solid part of the JOVIAL compiler and can be used confidently. However, permanent double precision testing should be added to JCVS for future compiler verification.

The final result of this contract was the production of a new JOVIAL compiler which is superior to the previous one. The amount of time and effort required to achieve this was underestimated and resulted in fewer optimizations than were originally anticipated. Several time extensions were also required. The entire contract was accomplished on terminals at the contractors facility with the exception of two man

weeks at RADC during final testing. The Time Sharing System (TSS) facility of the GCOS operating system for the Honeywell 6000 series computer was used throughout the contract. While this arrangement was generally satisfactory, the lack of a remote high speed printer capability definitely hindered progress. This was especially true during the final debugging stages. Large listings were sent by air freight from RADC on a daily basis, but this usually resulted in a two or three day turnaround period.

During the course of this contract, a number of requests were received concerning bugs in the compiler, particularly in the optimizer. However, the contract did not cover maintenance on the compiler and there was neither the time nor funds available to investigate these reported problems. Assuming that JOVIAL (J3) will continue to be used for some time, there definitely should be an on-going maintenance program for JOCIT. This could be accomplished either through an outside contractor or in-house if the talent is available. However, without a centralized responsible authority for future development and maintenance, the optimum usefulness of this valuable tool will not be realized.

SYMPL LANGUAGE ENHANCEMENTS

In order to make the JOCIT SYMPL Compiler more portable and more useful, a number of enhancements have been implemented. The enhancements are described in this section.

COMPILER DIRECTIVES

Compiler directives have been redefined as described in the following paragraphs.

Conditional Compilation

skip directive := !SKIP block indicator;

begin directive := !BEGIN block indicator;

end directive := !END block indicator;

block indicator := { letter / ∅ }

letter := A through Z

BEGIN and END directives having the same block indicator delimit a block of conditionally compiled code. If a skip directive occurs prior to the block and has the same block indicator, the code within the block is not compiled; otherwise the code is compiled.

Listing Options

list directive := !LIST { , { ON / OFF / ∅ } } ; default is ON

copy directive := !COPY { , LIST / ∅ } file name;

Listing of source code during compilation is handled in a hierarchical manner. The highest level is the compiler switches. If the NOLIST option is selected, no listing will occur regardless of what list directives may or may not be imbedded in the source. The next level of listing is the program itself. Lower levels are COPY files and nested COPY files. If a !LIST, OFF is encountered at any level, listing directives in lower levels have no effect. A !COPY is considered at a higher level than the code it refers to. Therefore, all listings will be suppressed until the code is

encountered. A !COPY, LIST would then allow listing control within the copy file to have effect.

DOUBLE PRECISION

Double precision ITEMs may be declared in one of two ways:

$$\underline{\text{double precision item}} := \text{ITEM } \underline{\text{name}} \left\{ \begin{matrix} \text{R } \underline{\text{mantissa}} \\ \text{D} \end{matrix} \right\};$$

$$\underline{\text{mantissa}} := \underline{\text{integer number}}$$

If the number indicated for the mantissa in a real item is greater than the number of bits in the mantissa of a single precision number of the target machine, the item is promoted to double precision.

Double precison constants are identical to single precision constants except that 'E' is replaced by 'D' to indicate the exponent.

COMPILER PACKED ARRAYS

In order to have a greater degree of machine independence, compiler packed arrays have been implemented. Arrays now have the following syntax:

$$\underline{\text{c.p. array}} := \text{ARRAY } \underline{\text{name}} \ [\underline{\text{dimension}}\ ] \ \underline{\text{array layout}} \ \ \underline{\text{array pack type}}; \ \ \text{ITEM } \underline{\text{item list}};$$

$$\underline{\text{array layout}} := \left\{ \begin{matrix} \text{P} \\ \text{S} \\ \phi \end{matrix} \right\} \quad \text{default is P}$$

$$\underline{\text{array pack type}} := \left\{ \begin{matrix} \text{N} \\ \text{M} \\ \text{D} \\ (\underline{\text{entry size}}) \\ \phi \end{matrix} \right\} \begin{matrix} \text{No packing} \\ \text{Medium packing} \\ \text{Dense packing} \\ \text{Specified packing} \\ \text{Default is N} \end{matrix}$$

$$\underline{\text{item list}} := \left\{ \begin{matrix} \underline{\text{item desc.}} \\ \underline{\text{item desc.}}, \ \underline{\text{item list}} \\ \phi \end{matrix} \right\}$$

item desc := name { I / U / S: Status Name / B / C / R / D / ø } { unspecified array info. / specified array info. }

unspecified array info := { size / ø } item pack type

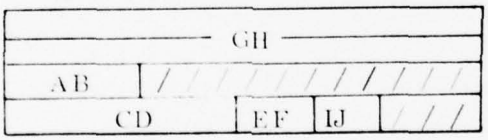
item pack type := { N / M / D / ø } default is array pack type

specified array info. := (word, fbit, size)

The following is an example of a compiler packed array:

```
ARRAY XY [1600] S  D; ITEM
        AB     C      2     N,
        CD     I     18     D,
        EF     U      4,
        GH     D,    --
        IJ     B;
```

The following allocation would result on the Honeywell machine:



Notice that AB was assigned a full word since it called for no packing.

If the array is unspecified and one of its items is specified, or vice versa, the following error message will be generated:

'INCONSISTENT WITH ARRAY'

PARAMETERIZED 'DEF' STATEMENTS

"DEF" statements have been enhanced to allow for embedded parameters. The following defines the syntax:

$$\underline{\text{def dec}} \quad := \quad \text{DEF } \underline{\text{defname}} \left\{ \begin{matrix} \underline{\text{elem def}} \\ \underline{\text{parm def}} \end{matrix} \right\}$$

$$\underline{\text{elem def}} \quad := \quad \text{"}\underline{\text{char string}}\text{"} \ ;$$

$$\underline{\text{char string}} \quad := \quad \left\{ \begin{matrix} \psi \\ \underline{\text{char string}} \end{matrix} \right\}$$

Where ψ represents any punchable character. Two consecutive double quotes represent a single double quote. Two consecutive exclamation points represent a single one. Single exclamation points are not allowed.

$$\underline{\text{parm def}} \quad := \quad (\underline{\text{parm list}}) \quad \text{"}\underline{\text{parm string}}\text{"}$$

$$\underline{\text{parm list}} \quad := \quad \left\{ \begin{matrix} \underline{\text{parm list}}, \\ \underline{\text{letter}} \end{matrix} \right\}$$

$$\underline{\text{parm string}} \quad := \quad \left\{ \begin{matrix} \underline{\text{parm string}} \ \underline{\text{parm string}} \\ \underline{\text{form parm}} \\ \underline{\text{char string}} \end{matrix} \right\}$$

$$\underline{\text{form parm}} \quad := \quad !\underline{\text{letter}}$$

There must be a corresponding letter in the parameter list for each formal parameter. When using the defined string, any unused parameters are replaced with null strings.

For example:

DEF ABC(X,Y,Z) "THAT !X MAN!Y!X IF HE GOES!Z!!"

If called with

ABC('WELL', 'SPEAKS', 'TOO SOON')

it becomes

THAT WELL MAN SPEAKS WELL IF HE GOES TOO SOON!

or if called with

ABC (, ' IS IN TROUBLE')

becomes

THAT MAN IS IN TROUBLE IF HE GOES!

MULTIPLE ASSIGNMENTS

Replacement statements now allow for multiple sinks on the left side of the equal sign(=), and have the following syntax:

$$\underline{\text{replacement statement}} := \left\{ \begin{matrix} \underline{\text{msink}} \\ \underline{\text{func}}\ \underline{\text{name}} \end{matrix} \right\} = \text{source;}$$

$$\underline{\text{msink}} := \left\{ \begin{matrix} \underline{\text{msink}}\ , \ \underline{\text{sink}} \\ \underline{\text{sink}} \end{matrix} \right\}$$

All sinks on the left side of the equal sign (=) will take on the value of the source.

For example in

A1, XYZ(3), BBB = 0

all three variables are set to zero.

COMPILER PHASES

Previously, the SYMPL compiler consisted of two phases (core overlays): ANZR and CODE. CODE, the largest phase, was broken into two smaller phases, CODE and ASMBL. An additional scratch file has been allocated to accommodate the object code intermediate information. These changes resulted in decreased core allocation. The compiler now occupies a 36K partition instead of the previous 43K partition.

FIELD EXTRACTION OPTIMIZATION

A new optimization has been added to cause more efficient code to be generated when assigning one field (partial word table item) to another. For example:

AAA(0) = BBB (0)

Assuming these variables were allocated as follows,

```
ITEM    AAA U(0, 6, 8),
        BBB U(0, 18, 8);
```

Then the following code would have been generated in the previous compiler:

```
LDQ     BBB
QLS     18          Extract
QRL     28          Position for Store
QLS     22
```

Using the new compiler, the following code is generated:

```
LDQ     BBB
QLS     12          Position Only
```

Code sequences will vary, depending on whether the items are signed or unsigned and depending on the length of the items with respect to each other. They will, however, generate shorter code sequences in most cases.

SWITCH OPTIMIZATION

Another optimization has been added to decrease core requirements for large switch lists (greater than 12 switch points). Under the old implementation the switch value is used as an index into a jump table. Each entry of the jump table contains a TRA instruction which jumps to its corresponding program label. This implies that one word per switch point is allocated in data space.

Under the new implementation, space is conserved by assigning two switch points per word using only the address. The appropriate half-word is loaded into a register and an indexed transfer occurs.

The following is an example of a jump table under both methods:

Old	New
.	.
.	.
.	.
TRA A1	ZERO A1, A2
TRA A2	ZERO A3, A1
TRA A3	.
TRA A1	.
.	.
.	.
.	.

The new method requires slightly more overhead, so unless a switch list contains more than 12 switch points, the old method is still used.

ERROR REPORTING

The following error messages have been added to accommodate the new compiler features:

'INCORRECT USE OF DEF'

'TOO MANY PARAMETERS'

'INCONSISTENT WITH ARRAY'

'MALFORMED CODE FILE'

'MISSING !END'

In addition, more error testing has been added to the expression scanner to report previously undiagnosed errors.

FUTURE CONSIDERATIONS

The following enhancements to the SYMPL compiler could be implemented in the future:

1. A modification to the language which would allow the programmer to perform array element and array moves. For example:

```
ARRAY ABC [100] S(4);
      .
      .
      .
ARRAY DEF [10] P(4);
      .
      .
      .
ABC [85] = DEF [3];
```

   This would cause four words from a parallel table to be moved to a serial table.

2. A re-evaluation of code generated by the SYMPL compiler to make use of special instructions for better localized optimizations. For example using AOS instruction in the case:

```
VAR = VAR + 1
```

3. Better code generated for compound IF statements. The current algorithm calls for the evaluation of each element of the compound IF, maintaining a Boolean value representing the sense of the IF condition. A better approach would be to jump out immediately if an AND condition is false or if an OR condition is true.

SUMMARY OF BENEFITS

Benefits derived from the enhancements done on the SYMPL compiler not only have made SYMPL a more portable and effective implementation tool, but have also resulted in better code being generated for both SYMPL and JOVIAL. In some cases, better local optimization occurs and, in the case of switches, more efficient data packing occurs. In addition the systems programmer now has better debugging tools and language features.

DOUBLE PRECISION

The JOCIT J3 compiler has been enhanced with a full double precision arithmetic processing data structuring and diagnostic capability. These enchancements required modification to the syntax tables, precognition processing (part of the analyzer), pragmatic functions, direct code processing, allocation, code generator, editing and library segments of the compiler within both the program and COMPOOL handling sections. In addition, modifications to the SYMPL compiler were required to process double precision declarations required for double precision compile-time processing.

SYNTAX - VARIABLE DECLARATIONS

Both the compiler syntax (JSYN) and the COMPOOL syntax (CSYN) were modified to recognize the following double precision syntactical contexts and to establish entry points within the Pratmatic Function (PF1, PF2) segments of the compiler:

- Scalar declarations of the form

  ITEM item-name D $

- Array declarations of the form

  ARRAY array-name (dimension-list) D $

- Table item declarations for both ordinary and defined table declarations of the basic form

  ITEM table-item-name D . . . $

- Mode directives of the form

  MODE D $

PRECOGNITION - CONSTANT RECOGNITION

The precognition segment of the compiler recognizes references to double precision constants of the form

[context] I.F D $\pm$ E [context]

Where I = interger portion

F= fractional portion

E= exponent

Examples: 0.025D

1.D4

1.27632D-10

Double precision constants are recognized in the following contexts:

- Arithmetic expressions
- Relational expressions
- Assignment statements
- Parameters
- Presets

The necessary modifications to the Preset Processor were minimal. The Preset routine contains calls to KONS. The second parameter of each call was changed from a value to the address of the value using the SYMPL LOCN internal function. There are also some cases where the call to KONS lies within a loop. Previously the loop increment had been one in all cases. A double precision check has been inserted in these places. If the test is true, the loop variable is set to two.

PRAGMATIC FUNCTIONS

The pragmatic functions PF1 and PF2 are defined as follows:

PF1- All explicit and implicit (MODE) double precision declarations of scalars, arrays and table items are posted to the Symbol Table during the first syntactical pass (ANZ1). Double precision constant references that have been parsed are resolved and posted to the Symbol Table as two-word constants.

PF2- Double precision operand contexts are processed for required conversion precedence determination, and legal context during the second syntactical

pass (ANZ2). Double precision is required to have the highest precedence within arithmetic and relational expression contexts. Conversion from or to double precision in assignment statements is dictated by the mode of the receiving item. Conversion from or to double precision is accomplished by generation of the appropriate intrinsic function within the Intermediate language passed to the optimization and/or code generation phases of the compiler.

DIRECT CODE

The routines which handle direct code were modified to allow double precision constants. Since the compiler contains many status lists which deal with the type of constants, each of these had to be updated to include a status value for double precision. The scanning routine was modified to allow constants containing the letter D. The D indicates the end of the characterisitc portion of the value and the beginning of the mantissa. The rules which govern the syntax of single precision constants (using the letter E instead), apply in the same manner. Double precision variables were added to the compiler to internally develop the value of the constants.

ALLOCATION

Double precision scalars and array declarations, exclusive of those declared in common, are linked to force allocation to an even-word addressing boundary for optimal fetching and storage. The SLC chains are serially searched for occurrence of double precision variables, and are relinked accordingly.

CODE GENERATOR

The code generator converts the input IL sequence to a linked triad form, then processes the triad table to generate output code to the code file. To generate double precision code, the code generator takes advantage of the fact that for the host machine of the JOCIT J3 compiler, the double precision floating point register overlaps the single precision floating point register. Therefore, double precision floating point follows the same code sequence as single precision floating point. In addition, a one bit field DPFT in a triad table entry is used as a flag to indicate that this triad entry

is a double precision primitive item, double precision value triad, or a double precision expression.

SETTING THE DPFT FLAG

The DPFT flag is set whenever:

1. A double precision IL operator or operand is encountered (e.g., RDEXP, DPLUS or any data item with type S'DBL' in the symbol table).

2. An integer, single precision item is converted to or from a double precision item (e.g., IDX, DRX).

3. A two-word temporary triad is created.

USE OF THE DPFT FLAG

Constant Handling

Constants are handled as follows:

1. The front end routine PCON is used to post a double precision constant; PICON is used to post the other constants.

2. The found value of the constant is returned in an array form, instead of just a one word item as previously, so that the second word of a double precision item can be retrieved.

3. The LOCN and ASEQ fields of constant entries in the Symbol Table are rearranged in such a way that all double precision items are grouped together and come ahead of the other constants and literals. Thus, the Editor only needs to align the constants once.

Loading, Storing, or Other Arithmetic Operations

If the actual location of the two words in the double precision operand are not contiguous (e.g., table item), three new routines accommodate the situations, as described below.

1. DLOAD is called whenever a load instruction of a double precision item needs to be generated. If the operand is a table item, a series of instruc-

tions will be generated, e.g.:

```
    LDA     First Word
    LDQ     Second Word
[*  STAQ    Temp]
[*  DFLD    Temp]
```

* (Not generated if the destination register is an AQ pair where temp is a two word temporary location.)

Otherwise, a single instruction LDAQ or DFLD is generated, depending on the specified destination register.

2. DSTORE is called whenever a store instruction to a double precision item needs to be generated. If the operand is a table item, a series of instructions will be generated, e.g.:

```
[*  LDAQ    Temp**]
    STA     First Word
    STQ     Second Word
```

* (Not generated if source register is an AQ pair.)

** (Temp was generated earlier in the program as a synonym of the operand.)

Otherwise, a single instruction STAQ or DFST is generated, depending on the specified source register.

3. DTLOAD is called before any arithmetic operation on a double precision table item is generated. It calls DLOAD to load the double precision operand. Later, if there is no immediate use of that item in the FP register, the SAVREG routine will store the computed table item into a two word temp. Thus, the operation, except a store, will operate on temp and will be treated as a non-table item.

Double Precision Code Generation

In GENS where the actual code generation occurs, whenever a floating point code sequence is encountered, the program will check to see if the DPFT flag is set, and will generate double precision or single precision code accordingly.

I/O Call and Exponentiation

The code generator calls the FORTRAN conversion library routine to handle ENCODE and DECODE. Consequently, an entry point of .FCNVD has been added in the library routine list in ICAL to handle double precision data conversion. Similar entry points have been added for FORTRAN exponentiation library routine calls to .FDXP1, .FDXP2, .FDXP3, .FXP4 to handle double exponentiation to integer, double to double, double to real, and real to double exponentiation.

Editor

Within the Editor phase a few procedures required modification. Switch points were added to accommodate double precision items. A double precision flag was added to the calling sequence for the real to decimal conversion routine (RTOD). RTOD will now output double precision values with the letter D for double precision items rather than an E as with single precision items.

KONS was the routine requiring the most extensive modification within the Editor phase. KONS is called to produce both listings and object code for constants. The second parameter in the calling sequence to KONS was changed from a value to the address of a value. In this way, the routine could easily obtain the second word of a double precision constant. If the constant is a double precision type, the two words are written to the object file and the proper constant is output to the listing.

The TEMP chain is now reordered in the Editor phase. The ASEQ for this chain was straightened so that the double precision temporaries would come first, followed by all other types. This permits the double precision constants to be aligned on even word boundaries (a system requirement).

Run-Time Library

Only the JOVIAL monitor routine required modification in the Run-Time Library. Previously, it did not have the capability to monitor double precision items. As with the direct code processing, a double precision type was added to the internal status list. A special code (bit flag) was added to the calling sequence produced by the code generator. If the flag is true, the value being monitored is a double precision item. In this event, the monitored value is output with the letter D rather than an E as is done with single precision items.

NEW OPTIMIZER

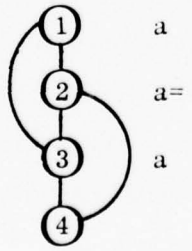
A new optimizer was developed under the Improvements to JOCIT contract. This new optimizer includes such capabilities as code straightening, dead variable analysis, unreachable and unexitable code deletion, and several varieties of folding. It is possible to run the new optimizer independently from the old one. Either optimizer or both optimizers may be executed during a given compilation.

DESIGN CONSIDERATIONS

The new optimizer uses a technique called P-graphing to collect information about where a given variable is set or used. The P-graphing technique used is discussed by Loveman [3]. The P-graph for a variable supplies answers to such questions as "which generations affect the value of a given use of variable X?" and "which uses of variable X result from a particular generation?" In addition, if the P-graph is complete (as it is in this particular optimizer), the optimizer answers questions such as "is the variable X guaranteed to have the same value at point a as it is at point b?"

Rather than assuming that each variable is both set and used by every subroutine called, an effort is made to maintain lists of variables actually used by internal procedures, so that generations and uses will not be created unnecessarily. This helps not only to allow better code to be generated, but also to cut down on the size of the data base.

P-graphing is a powerful technique because it pays attention only to program flow and not to the way the program was written. For example, in the following program segment,



the old optimizer is unable to tell that the two uses of "a" (in blocks 1 and 3) result from the same generation due to the fact that the branch at 2 is treated as unconditional. The new optimizer, however, does recognize the fact that the two uses have the same value.

At the start of the project it was necessary to decide whether the best course of action would be to attempt to add enhancements to the old JOCIT optimizer or to develop a new optimizer with increased capability. The decision was made in favor of the new optimizer for several reasons. First, P-graphing provides a more complete analysis of data flow. This allows a higher degree of optimization since an optimizer should only perform those optimizations which will not change the results of correct programs. Obviously, then, the more that is known about the flow of data and control in a program, the safer the optimization. Secondly, there have been reliability problems in the past with the old optimizer, and an attempt to modify it could conceivably introduce other problems. To avoid such problems and to take advantage of the data flow information provided by P-graphing, a decision was made to adapt a design incorporating P-graphing into the JOCIT system.

The original intent was for the new optimizer to supersede the old. However, there was insufficient time to incorporate into the new optimizer all of the optimizations performed by the old optimizer and the added optimizations. For this reason, the old optimizer was retained and either optimizer or both can now be run. In the current scheme, the old optimizer runs before the new one. (There is no option for the order of execution.) This particular ordering was chosen primarily because the old optimizer performs optimizations which depend on the existence of loop operators. The new optimizer transforms loops into sequences of ordinary operators, so running the new optimizer first would degrade the performance of the old one with respect to loops. A further advantage is that the new optimizer may generate sequences of IL which the old one might not be prepared to handle. In fact, there were several interface problems between the new optimizer and the code generator. It is likely that the problems would have been too numerous and too hard to remedy had the old optimizer been forced to read the IL produced by the new optimizer.

OPTIMIZATIONS PERFORMED BY THE NEW OPTIMIZER

The following describes the optimizations performed by the new optimizer. Optimizations performed by the old optimizer are not discussed here except to contrast the results produced by each optimizer. Optimizations performed by the old optimizer are documented in the JOCIT workbook.

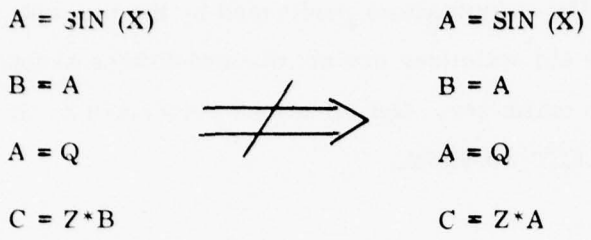
FOLDING

Three types of folding are performed in the new optimizer: constant folding, scalar folding and expression folding. These folding types derive their names from the value types appearing on the right-hand side of the assignment statement. The occurrence of a variable on the left-hand side of the assignment is a generation of that variable. This optimizer is concerned only with assignments whose left-hand side is a scalar. It would be possible to extend the folding the optimizer does to include array elements but this would be difficult. While constants and scalars are properly classified as expressions, the term "expression folding" as used here is meant to exclude constants and scalars and embrace only those expressions for which some actual computation is performed.

The three types of folding involve replacing a use of the variable which was on the left-hand side of the assignment by the right-hand side of the assignment. For example:

```
A = 1          ⟹    A = 1
B = A+C             B = 1+C
```

The main reason for the distinction between the types of folding involves the differences in when each can and should be applied. Constant folding can be applied to any use (as opposed to a use/generation) of the generation since a constant has the same value throughout the program. Scalar folding can be applied only when the value of the scalar which was on the right side of the assignment has the same value at the point of assignment as at the use. The following is an example of when scalar folding can not be applied:



In this example, the value of A in line 4 is not the same as on line 2. Therefore, it would be incorrect to substitute A for B here.

Expression folding is even more restricted than scalar folding. It is incorrect to perform expression folding if the value of any of the components of the expression differs between the generation and use occurrences of the scalar to which the expression was assigned. Also, it makes no sense to fold an expression onto two or more uses since this would amount to undoing common expression elimination. Therefore, expression folding is performed only when there is exactly one use of the generation.

Folding can be valuable for several reasons. Constant folding can allow the code generator to make greater use of immediate instructions. Scalar folding can allow the optimizer to find common expressions which have the same value but are not textually the same; for instance, . . . X+Y . . . Z=Y$. . .X+Z. Note that since the new optimizer does not currently perform common subexpression elimination, full advantage is not being taken of folding. Folding also increases the possibilities for dead store elimination by making dead any assignments which formally had uses. Folding can also help the code generator to maintain values in registers, eliminating unnecessary loads. Unfortunately, little is gained on the Honeywell machine due to the fact that the contents of the A and Q register must be destroyed for all but the simplest computations.

There is an overlap between the folding done by the old and new optimizers. However, the new optimizer performs folding in some cases in which the old one does not. One example of such a case is shown above under "P-graphing." It is an IF. . . THEN . .

ELSE construct, which might appear in JOVIAL as IFEITH A $ A = . . . ORIF 1 . . . A $. The old optimizer does not recognize that the two uses of A have the same value and are unaffected by the assignment. The new optimizer recognizes that they result from the same generator and would fold accordingly.

Dead variable analysis is performed for the scalars in each procedure. If, at some time, a generator has no uses and the generation is the result of an assignment statement, the assignment is deleted. While this optimization is performed for assignments which are dead at the source level, it has a larger payoff with regard to assignments which become dead due to the application of other optimizers such as folding. Consider the code sequence:

```
A = expression

IF A NQ Q
```

If this is the only use of that particular generation of A, the expression will be folded into the IF and the store will be deleted. Constant folding and scalar folding may also create similar situations.

At the present time there is no prevision to deallocate variables which become unused as the result of dead store elimination. A variable's P-graph contains sufficient information so that an optimizer such as this could be added at a future date.

CODE STRAIGHTENING

One of the features of the new optimizer is code straightening. The purpose of the code straightening as implemented here is to eliminate unnecessary GOTOs and to ensure that a block precedes all those blocks which it back dominates. Eliminating unnecessary GOTOs saves execution time and space and allows good code to be generated for conditionals without complicating the front end.

The code generator requires that value definitions (VALSs or VALDs) textually precede any uses (VALUs). In an unstraightened program it may be possible for a use to logically follow a definition but to textually precede it. Code straightening, as imple-

mented here, guarantees that a generation will procede its uses.

The algorithm used can be divided logically into two distinct parts. The first part eliminates GOTOs to labeled GOTOs. The code sequence

```
        GOTO    L1
        .
        .
        .
L1.     GOTO    L2
```

is changed to

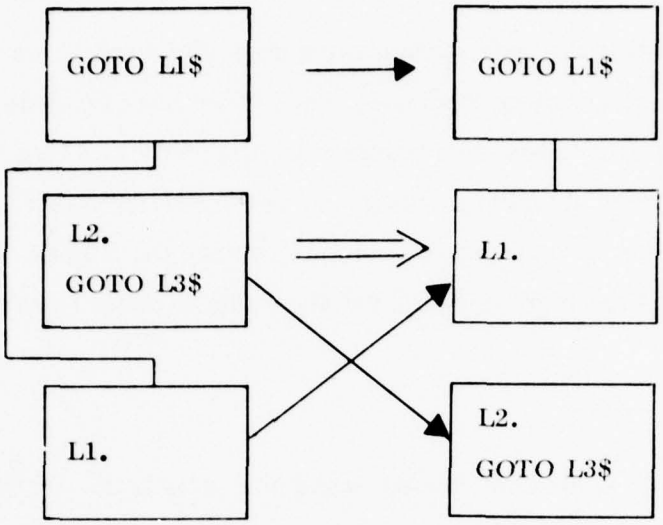
```
        GOTO    L2
L1.     GOTO    L2
```

If at any point no references to L1 remain and there is no fall through, the GOTO at L1 is deleted. While such sequences are relatively rare in source code, this situation does occur in the code which is generated for conditionals.

After GOTOs to GOTOs have been eliminated in the manner described above, the code is reordered both to ensure that back dominators precede their dominatees and to eliminate some more unnecessary GOTOs. Starting from the program entry an attempt is made to have each block followed textually by a block which also follows it logically. The logical successors of a block ending in a GOTO or TSST are examined. If one such successor has not yet been ordered, it is placed immediately following the block being processed and the other successor, if any, placed on a list of candidates for future processing.

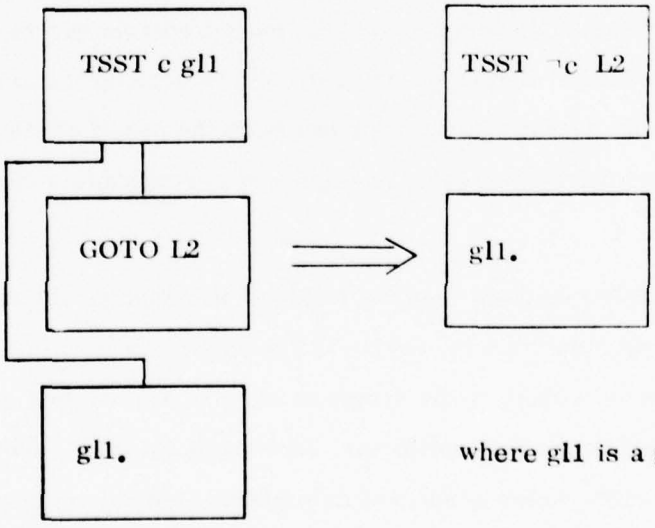
The following are examples of the types of optimizations performed:

1. Useless GOTO elimination:



Since the block containing the definition of L1 now follows the GOTO, that GOTO is superfluous and can be deleted. The block containing L2 will be placed somewhere else following a reference to L2.

2. Test reversal and GOTO elimination



where g11 is a generated label.

Since the fall through block after a TSST contains only a GOTO, the sense of the TSST can be reversed and the target changed to branch directly. Since there are no other predecessors of the GOTO, it can be deleted.

The algorithm is complicated somewhat by the code generator's requirement that the IL operator PTRM be both the physical and logical end of the program.

It should be noted that the code straightening algorithm used is not that of Earnest, et al. [1]. Since P-graphing eliminates much of the need for code straightening (in the sense of moving code based on whether it is logically in a loop or not) it was possible to use a simpler algorithm. Inasmuch as the new optimizer, as currently implemented, is not powerful enough to completely replace the old one, it may have been better to have used the more complicated algorithm in order to enhance loop optimization.

CONSTANT ARITHMETIC

Constant arithmetic is also performed by the new optimizer. While the constant arithmetic package is basically the one which appeared in the old optimizer, but modified to work with the new optimizer's data base, some improvement in the optimization obtained can be expected due to the broadened scope of constant folding.

The new optimizer deletes code which is found to be unreachable or unexitable. Unreachable code is code for which there exists no path from the entry point. Unexitable code is code which provides no path to the exit. The analysis for these cases is performed globally so that code which is labeled but unreferenced, as well as unlabeled following an unconditional GOTO, is deleted. While most of the unreachable code which has been found in test cases so far has been the result of constant arithmetic deleting tests, there have been some examples of unreachable code found in real programs.

The algorithm used to complete forward and back dominators for the blocks in the flow graph is based on an algorithm by Tarjan [5] and operates in O (n log n + e) where n is the number of basic blocks in the program and e is the number of edges. In the implementation used in the new optimizer, forward dominators are computed by reversing all of the edges in the graph and calculating "back dominators" for this reversed graph, starting from the exit. Tarjan [6] has developed a faster algorithm (almost linear) for calculating dominators. However, since the dominator calculation is such a small part of the optimizer process, it is unlikely that changing to the newer

algorithm would have an appreciable effect on compilation time.

OPTIMIZER WEAKNESSES

The new optimizer has two main weaknesses - it is relatively slow and requires large amounts of core to handle large programs. These problems are explained at least in part by the fact that the optimizer is almost completely untuned at the present time. The optimization process is iterative because performing one optimization may uncover other optimizations which can be performed. For example, folding may allow constant arithmetic to be performed in an IF statement. It may then be possible to determine that one branch of the IF is never taken which may allow some code to be deleted as unreachable. This may allow additional folding to be done, etc. It is possible to limit the number of iterations allowed at the cost of "overloading" some optimizations; however, no attempt has been made to do so at the present time.

The optimizer can be expected to be large due to the nature of its data base. In order to perform global optimizations it is necessary to maintain large amounts of information in core unless compile time is allowed to increase unreasonably. There are a number of in-core tables in this implementation which hold information about the source program. All of the entries in a given table are contiguous. This allows smaller pointers and, hence, smaller entry sizes, but this approach does require additional work when a table is to be expanded as opposed to using a linked list table structure.

Tables are allocated in both phase space (an area the size of the difference of the sizes of the largest phase and the optimizer) and symbol table space. When a new table entry is needed, it is obtained from the freed entry list of that table, if possible. If no entry can be determined there, an attempt is made to obtain the entry from the unused space associated with the table. If there is not enough room there, an attempt is made to move the surrounding tables to obtain more space for that table. If that fails, then the unused space for the other table is returned. If there is still not enough space, tables are shuffled from one area to the other in an attempt to obtain enough space. Finally, if there is still not enough space and the symbol table has not been expanded to its limit, a system call is made to obtain more space.

Rather than to force each routine which creates table entries to check, each time the space manager returns, to see if there really was enough space to satisfy the request, a block of space in each area is held in the reserve. This, at least in theory, allows the module which is currently executing to terminate gracefully. Then, between modules, a check is made to see whether or not the reserve has been used. If so, appropriate action can be taken. If the space manager is unable to satisfy a request after the reserve has been used, it is forced to terminate the compilation.

In practice, a number of large programs have caused the reserve to be exhausted and compilation to be terminated. Experimentation with the size of the reserve would probably lead to a size which would allow larger programs to be compiled in a given core size. Each routine which calls the space handler would provide it with an address of a routine which could allow a more graceful exit when there is no more available space. An even better method would be to devise a scheme in which pieces of programs could be optimized. This could require considerable work, however, and there was insufficient time to investigate any of these possibilities to any extent.

There were several problems which made the final result less than it might have been under ideal circumstances. Perhaps the most important of these was the lack of good interactive debugging facilities on TSS. An interactive debugger such as PCF on CSTS or DDT on the DEC-10 could have made the job significantly easier. A good deal of effort was spent building in debugging dumps and traces. While these proved valuable in the debugging effort and could not have been eliminated entirely, even if an on-line debugger had been available, the space occupied by the debugging routines was used for data when none of the debugging options were on. This caused some difficulties in reproducing bugs, however, because some problems which occurred when there were no debugging options on were masked with debugging on because the compilations ran out of space before the problem occurred.

The debugging features developed include formatted table dumping, tracing (at 4 different levels) and data change monitoring. Formatted dumps of all of the major tables in the optimizer are available during the execution of the compiler and all tables are dumped if an internal inconsistency (OERROR) is detected.

Tracing is also triggered by control card option and CONTROL statement. The levels of tracing available are major module, routines within major modules, utility, and low level utility. Through use of the CONTROL statement, tracing can be varied for each procedure being compiled.

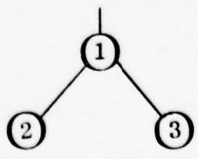
Data change monitoring is available for specific core locations and for table entires. Whenever a traced subroutine is called, the old value for the item being monitored is compared with the current value. A dump of the item is produced if there has been a change. Data change monitoring is available only through the use of the CONTROL statement.

The other major difficulty arose because of a lack of available disk space. Due to the fact that there was only space enough for one optimizer compiler for the bulk of the time, it was necessary to link all new (and unstable) modules into one compiler. This meant that everyone had to contend with everyone else's bugs. Things would have gone smoother had it been possible to debug unstable modules individually and to "make public" only those modules which were stable.

This problem was circumvented to a certain extent by introducing control card options to allow certain modules (e.g., folding) to be bypassed. In other cases (for example, when the space manager was modified) this could not be done.

Several interesting and as yet unanswered questions have been raised as a result of the work on this project. One concerns a possibly novel approach to optimization. Another concerns the manipulation of P-graphs.

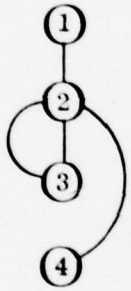
At the start of the project the possibility of using Kirchoff's laws [3] to obtain relative frequences of execution for the various blocks in a program was investigated. Kirchoff's law was stated with regard to electrical circuit theory and says "flow in = flow out." In the case of flow graphs, we also know that the flow along any path is non-negative. It is easy to deduce local information.



If we let F_i stand for the flow into (or out of) block$_i$ and f_{ij} be the flow from block i to block j then $F_1 \geq F_2$ and $F_1 \geq F_3$ since $F_2 = f_{12}$ and $F_3 = f_{13}$ and $F_1 = f_{12} + f_{13}$ and all of the $f_{ij} \geq 0$.

It is difficult to determine the relationships between blocks, which are the relations to be determined and which indicate whether or not it is possible to tell anything definite about the relationship. (In the flow diagram above, it is impossible to tell whether 2 or 3 has a higher frequency of execution without knowing something about the probability of taking one branch or the other at 1).

Relative flow frequencies are important because it is desirable to move code from a region of high frequency to a region of lower frequency. By calculating frequencies, it may be possible to perform code motion without regard to formal loops. It should be noted, however, that this method has drawbacks. No definite relationship can be established between the predecessor of a DO WHILE loop and the code within the loop itself. For example:



No relationship can be established between 1 and 3. This is correct because 1 may be executed while 3 is not (1 > 3) or 3 may be executed many times while 1 is executed only once (3 > 1).

It is not clear how time-consuming the calculation of frequencies would be or what the payoff, if any, would be in terms of improved optimization. No attempt was made to program the frequency evaluator or to prove that it would work in all cases due to the limited time available for implementation.

Another interesting question concerns P-graphing. Although no empirical evidence has been gathered with respect to the new optimizer, it would appear that P-graphing

takes a sizeable amount of the time required for optimization. Since the algorithm used in this optimizer was originally developed, an algorithm which is faster (at least for very large programs) has been published [2]. Whether it would be faster for programs which are small enough to be optimized is not obvious.

There may be another approach which could prove fruitful. Common variables tend to have P-graphs which are basically the same -- composed chiefly of implicit uses and generations at calls to external procedures and a relatively small number of uses elsewhere. Would it be possible to construct a basic P-graph template to obtain the P-graph for each individual variable? If so, would it be faster than constructing the P-graph from scratch?

Although a comprehensive performance analysis of the new optimizer was not carried out, some figures are available with respect to the J series test cases. There are roughly 100 such test cases. Of those for which compilation statistics are available (76), the smallest was under 20 source lines, and the larger programs were between 100 and 150 source lines. Object sizes ranged from 35 (octal) words to 3651 words unoptimized and 35 to 2677 optimized. (These figures exclude COMPOOL compilations which are not optimizable.) Compilation times ranged from .0003 to .0035 unoptimized and .0004 to .0097 optimized.

In 61 cases the optimized code was smaller, in 11 cases there was no change, and in 9 cases the optimized code was larger. In absolute terms the best test case was one which shrank from 2475 to 601 words. Percentagewise, the best results were obtained with a program which was 304 words unoptimized and 53 words optimized. A total of 29 test cases experienced what could be described as significant reductions in size (defined here as $>100_8$ words).

The largest degradation was 13_8 words. This is due to the fact that the code generator does not remember the length of a Hollerith function which has been VALSed and is forced to generate a subroutine call rather than use an EIS instruction. Other increases in the generated code are due to the fact that, on the Honeywell machine, it takes one instruction to increment memory, but two instructions to add one to a register and store the value. In addition, there were 16 programs in which there was only a

small decrease ($1\text{-}20_8$ words) in program size.

Thus, a substantial reduction in size occurred in 29 cases, and there were no significant changes in 31 cases. It should be cautioned, however, that test cases tend to be more amenable to optimization than actual programs. It is unlikely that any real programs would exhibit the size reductions of nearly 50% which were exhibited in 12 of the test cases. It is also unlikely that real programs would show the sort of degradations which occurred in four test cases. Real programs can be expected to show a modest improvement in most cases, if not all.

POSSIBILITIES FOR FUTURE IMPROVEMENTS TO THE OPTIMIZER

The new optimizer is probably more significant for the potential it provides than for the optimizations it actually performs. The new optimizer enhances the folding and constant arithmetic capabilities of the JOCIT compiler and adds code straightening and dead store elimination. It provides a degree of optimization for some programs which were formerly unoptimizable under the JOCIT system (notably the 14 SAC programs).

More importantly, however, it provides a base to which more powerful optimizations and additional diagnostic capabilities can be added. The following is a list of possible additions and improvements. It is not intended to be exhaustive, but merely to provide an idea of what could be done.

1. The optimizer can use some tuning and improvements with respect to compilation time and space required. Some improvement could be seen with minor testing and modification. The size of the compiler and size of the reserve could be specified by control card options. Modifying the optimizer so that it could optimize sections of code would require a significant amount of work, but it would enable larger programs to be compiled.

2. Loop optimizations could be added. Currently the old optimizer performs all of the loop optimizations for JOCIT. There is some code for code redistribution in the new optimizer but it is incomplete and, therefore, bypassed. The approach to loop optimization could be the conventional approach or the experimental approach based on execution frequencies.

3. Folding could be extended to include array references and overlaid variables. Overlaid variables could be handled by combining their P-graphs. Handling array references would be more complex.

4. An improved register allocation scheme could be developed. The P-graph for a variable provides sufficient information to determine what variables are good candidates to be assigned to registers. At present, most of this information is lost because the code operator assigns registers and the P-graphs no longer exist when registers are assigned. Only that information which can be transmitted via the VALD/VALS/VALU mechanism survives. A global register allocation algorithm would be a major undertaking. However, there are some smaller changes which could be made at a lesser cost:

   a. VALSs or VALDs could be created for merges as well as generations. This could result in better code for item switches.

   b. VALDs could be generated for parameters.

   c. VALDs could be used instead of VALSs in some cases to eliminate unnecessary stores.

   It should be noted, however, that these modifications would have a greater affect on a machine with more registers than the Honeywell machine.

5. It would be possible to diagnose cases in which a variable may be used before it is set. A variable P-graph together with an indication of whether the variable is preset provides sufficient information to detect this condition.

6. Self tests (e.g., A EQ A) and self assignments (e.g., A = A) can be deleted with relative ease. Self tests occur in some forms of FOR loops.

7. The code for Hollerith functions could be improved if the code generator remembered the length of the result of a function which was VALSed. This would enable EIS instructions to be used in place of a subroutine call.

8. The user interface could be cleaned up in several cases. Some of the OERROR messages could be replaced by messages which would be more meaningful to users. The present OPT options tend to be more historical than logical. The options and defaults could be reassigned to make the optimizer easier to use.

BIBLIOGRAPHY

1. Earnest, Blake and Anderson, "Analysis of Graphs by Ordering of Nodes, "JACM, vol. 19, no. 1, Jan. 1972, pp. 23-42.

2. Karr, Michael, P-graphs, Report CAID-7501-1551, Massachusetts Computer Associated, Wakefield, Mass.

3. Knuth, D. E., The Art of Computer Programming, vol. I., Fundamental Algorithms, Addison-Wesley. Reading, Mass., 1965, P. 167.

4. Loveman and Faneuf, "Program Optimization - Theory and Practice." Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines, SIGPLAN Notices, vol. 10, no. 3, March 1975, pp. 97 - 102.

5. Tarjan, Robert E., "Edge-Disjoint Spanning Trees, Dominators and Depth-First Search, "Stanford University Computer Science Department Technical Report STAN-CS-74-455, Sept. 1974, p. 40.

6. Tarjan, Robert E., "Solving Path Problems on Directed Graphs," Stanford Computer Science Department, Technical Report STAN-CS-75-528, Oct. 1975.

APPENDIX A - JCVS and SAC Program Comparisons

The following table presents a comparison of the various modes of compilation for the JCVS tests. There are four categories:

a. No optimization (NOPT)

b. Old optimizer (OPT/1/)

c. New optimizer (OPT/2/)

d. Combined old and new optimizers (OPT/12/)

There was essentially no difference between the old compiler and the new compiler in the NOPT and OPT/1/ modes, but this was expected since no changes were planned in that area. Therefore, only the above modes are compared in the new compiler. The compile times shown are seconds and the sizes of the object code are in decimal. The asterisk indicates a fatal error.

	Time (seconds)				Size (decimal)			
JCVS Test	NOPT	OPT/1/	OPT/2/	OPT/12/	NOPT	OPT/1/	OPT/2/	OPT/12/
CLASS1								
C1	31.3	46.1	*	115.9	2107	1870	*	1664
C2	28.4	40.0	*	79.6	1839	1594	*	1406
C3	35.6	61.2	*	*	2792	2077	*	*
C4	27.4	39.2	132.1	*	2004	1770	1694	*
C5	32.0	49.0	*	*	2109	1855	*	*
C6	36.7	55.8	*	*	2421	2150	*	*

JCVS Test	Time (seconds) NOPT	OPT/1/	OPT/2/	OPT/12/	Size (decimal) NOPT	OPT/1/	OPT/2/	OPT/12/
CLASS2	(This class tests error detection capabilities of the compiler)							
ER1	19.4	*	65.9	*	1046	*	991	*
ER2	1.1	1.1	1.1	1.4	4	4	4	4
ER3	This test deliberately produces a fatal error - all modes operated in the same manner.							
ER4	1.1	1.1	1.1	1.4	5	5	5	5
ER5	1.1	1.1	1.4	1.4	7	6	4	4
Compool Test								
COMP	1.8	1.8	1.8	1.8	109	109	109	109
CTST	3.2	4.3	4.3	7.2	215	176	176	176
PA	1.1	1.1	1.1	1.4	27	6	16	6
PB	1.1	1.1	1.1	1.4	18	6	6	6
CP1	.7	.7	.7	.7	5	5	5	5
CP2	.7	.7	.7	.7	0	0	0	0
TSTCP	1.8	2.2	2.2	2.9	89	75	75	75

JCVS Test	Time (seconds) NOPT	OPT/1/	OPT/2/	OPT/12/	Size (decimal) NOPT	OPT/1/	OPT/2/	OPT/12/
A Series								
TE01	2.2	2.9	3.6	4.3	119	108	108	104
TE02	2.5	3.2	4.3	5.0	124	123	123	123
TE03	2.2	3.2	3.6	4.7	146	143	143	143
TE04	1.4	2.2	2.5	2.9	63	64	58	61
TE05	2.5	3.2	4.0	5.0	97	97	96	96
TE06	2.2	2.9	3.2	4.0	3185	3177	3188	3180
TE07	1.1	1.4	1.4	1.8	24	24	24	24
TE08	2.5	2.9	3.6	4.3	157	153	155	155
TE09	5.4	7.2	6.8	9.7	569	505	503	502
TE10	1.8	2.2	2.5	2.9	66	62	60	59
TE11	6.5	8.3	10.8	14.4	620	394	426	370
TE12	1.4	1.8	1.8	2.2	52	52	48	54

JCVS Test	Time (seconds) NOPT	OPT/1/	OPT/2/	OPT/12/	Size (decimal) NOPT	OPT/1/	OPT/2/	OPT/12/
B Series								
TE14	2.9	4.0	5.8	6.1	189	187	190	190
TE15	2.2	2.9	3.2	4.3	219	213	221	215
TPOL	1.1	1.1	1.1	1.1	21	21	21	21
TE20	1.4	2.2	2.2	2.9	48	51	46	48
IFFR	.7	.7	.7	.7	4	4	4	4
FRIF	1.4	2.2	2.5	3.2	64	63	63	63
BMLO	2.2	2.5	2.9	3.6	113	113	113	113
MINU	1.4	1.8	1.8	2.2	55	47	55	46
STCT	1.1	1.1	1.1	1.4	4	4	4	4
LABL	1.1	1.4	1.4	1.8	10	10	10	10
TABL	2.5	2.9	3.2	3.6	165	164	164	164
CSTP	.7	1.1	1.1	1.4	3	3	3	3
DEFI	1.4	1.8	2.5	2.9	48	46	50	46
PRES	1.1	1.4	1.4	1.8	31	31	31	31
TRAN	1.4	1.8	1.8	2.2	56	56	56	56

JCVS Test	Time (seconds) NOPT	OPT/1/	OPT/2/	OPT/12/	Size (decimal) NOPT	OPT/1/	OPT/2/	OPT/12/
C Series								
CAPY	8.3	9.7	16.9	18.4	445	409	380	380
DIR1	6.1	7.6	10.4	14.0	487	346	335	347
DIRE	13.3	17.3	44.6	58.0	1229	995	984	984
STC	5.0	5.8	7.2	8.3	599	549	572	554
STOP	2.5	3.2	4.0	4.3	163	161	161	161
CLOS	2.9	4.0	5.0	6.1	197	185	188	187
MODE	4.3	5.4	6.8	7.9	270	262	263	263
WORD	1.8	2.2	2.5	2.9	82	82	82	82
LSWD	1.4	1.8	2.2	2.5	66	66	66	66
ABS	2.2	2.9	3.6	4.3	137	119	116	115
BLNK	1.4	1.8	1.8	2.5	74	74	74	74
LLIT	10.1	14.8	16.6	23.0	977	941	936	935

The following table gives the compile time in seconds and the size of the generated object code in decimal for fourteen SAC programs that had fatal errors on the old optimizer. These results were not obtained with the latest version of the compiler produced under this contract, since the SAC programs were removed from the disk earlier. However, the values should be a good indication of the expected results with the latest compiler. An asterisk indicates a fatal error.

SAC Program	Time (seconds) NOPT	OPT/1/	OPT/2/	OPT/12/	Size (decimal) NOPT	OPT/1/	OPT/2/	OPT/12/
BPAD	9.0	**	19.1	**	508	**	478	**
BPBD	137.5	**	1110.2	**	8345	**	7588	**
BPCD	97.2	**	*(570.2)	**	7268	**	*(6394)	**
BPDD	29.5	**	120.2	**	2068	**	1836	**
BPED	47.9	**	173.2	**	3339	**	3127	**
BPFD	111.2	**	718.2	**	6799	**	6349	**
BPID	114.8	**	814.0	**	9847	**	8405	**
BPOR	32.8	**	140.0	**	2422	**	2259	**
BPPR	28.8	**	156.6	**	1984	**	1900	**
BPQR	88.9	**	581.0	**	7002	**	6482	**
BPSA	32.0	**	118.4	**	2627	**	2539	**
BPTR	112.7	**	1385.3	**	8539	**	7399	**
BPUR	64.8	**	192.6	**	4766	**	4682	**
BPVR	30.6	**	158.4	**	6224	**	6108	**

The double asterisks indicate that the compilation for those modes was not attempted after some corrections had been made to the old optimizer. Therefore, the status of those modes is unknown until the SAC programs can be restored to the disk and retried with the new compiler. Previously, however, all of the fourteen SAC programs had fatal errors in the old compiler.

The program BPCD is shown with a fatal error under the OPT/2/ mode. Earlier it had been compiled successfully in 570.2 seconds and generated 6394_{10} words of

object code. Since that compilation, however, the upper limit to which the compiler was allowed to grow during a compilation was lowered. This apparently caused a lack of sufficient table space to successfully complete the compilation of BPCD. A similar situation occurred with a number of the CLASS 1 JCVS tests compiled under the OPT/12/ mode.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.