Interprocedural Constant Propagation: An Empirical Study

ROBERT METZGER and SEAN STROUD
CONVEX Computer Corporation

Constant propagation is an optimization that substitutes values for names. Interprocedural constant propagation performs this substitution throughout an entire program, propagating constant values across procedure boundaries. CONVEX Computer Corporation has developed an interprocedural optimizer for imperative languages, which is available to users of its C-series supercomputers. An aggressive interprocedural constant propagation algorithm, such as the one implemented in this optimizer, can find many constants to propagate into procedures in scientific FORTRAN applications. Detailed statistics derived from compiling a large set of real scientific applications characterize both the opportunities for interprocedural constant propagation in these codes and the effectiveness of the algorithm described. These statistics include the frequency of interprocedural constant propagation, the different types of interprocedural constant propagation, which constants are most frequently propagated, and the relationship between interprocedural constant propagation and other interprocedural optimizations.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*data types and structures*; *procedures, functions, and subroutines*; D.3.4 [**Programming Languages**]: Processors—*code generation*; *compilers*; *optimization*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Abstract interpretation, code optimization, constant propagation, control-flow graph, data-flow graph, interprocedural analysis

1. INTRODUCTION

The need for interprocedural constant propagation results from a conflict between the goals of software engineering and performance optimization. The wise programmer generalizes code to make it more extensible and maintainable. Two ways to generalize systems are to postpone binding values to names until run time, and to modularize the system using procedures. Generality is a noble goal, but there is some performance penalty to be paid for extra memory loads and procedure calls.

An optimizing compiler particularizes code to make it execute more quickly. It promotes the binding of values to names to compile time (procedural

Authors' address: CONVEX Computer Corporation, 3000 Waterview Parkway, Richardson, TX 75083-3851.

constant propagation). An interprocedural optimizing compiler takes the process one step further. It will propagate constants across procedure boundaries or eliminate the procedure boundary altogether by inline substitution.

CONVEX Computer Corporation has developed an interprocedural optimizer for imperative languages, which is available to users of its C-series supercomputers. This optimizer is packaged together with existing CONVEX compilers in a product called the Application Compiler.

Currently, the Application Compiler processes programs containing FORTRAN and C source code. The driver program for the Application Compiler determines which source files need to be compiled and applies the appropriate front end to those files. The front end performs lexical analysis, parsing, and semantic analysis. It writes to disk the symbol table, annotated parse tree, and miscellaneous information for each procedure.

After all source files that need to be (re)compiled have been processed, the driver invokes the interprocedural optimizer. The following analyses are performed in the order listed:

—*Interprocedural type checking*. Which calls have type clashes and which globals are declared incompatibly in different source files?

—*Interprocedural call analysis*. Which procedures are invoked by each call?

—*Interprocedural alias analysis*. Which names refer to the same location?

—*Interprocedural pointer tracking*. Which pointers point to which locations?

—*Interprocedural scalar analysis*. Which procedures (and subordinates) use and assign which scalars?

—*Interprocedural constant propagation*. Which globals and arguments are constant on entry to which procedures?

—*Inline analysis*. Which procedures should be inlined at which call sites? Both subroutines and functions are candidates for inlining, though the optimizer does not attempt to inline procedures written in one language into a procedure written in another language.

—*Clone analysis*. Which procedures should be duplicated so that they can be modified with information about a specific call site?

—*Interprocedural array analysis*. Which procedures (and subordinates) use and assign which sections of arrays?

—*Storage optimization*. How should application data structures be rearranged to improve the usage of the memory hierarchy?

Each analysis algorithm reads from the program database and writes the information it generates back to that database.

After the interprocedural algorithms have completed, the driver program invokes the compiler common back end for each procedure in the application. The back end consists of a machine-independent optimizer and a machine-dependent code generator. In the Application Compiler, the optimizer has been modified to make use of information gathered by an interprocedural analysis phase, rather than to make worst-case assumptions about things like procedure side effects.

The optimizer first performs all of the standard procedure-scope scalar optimizations; it then performs automatic vectorization and parallelization. The code generator is table driven and supports several similar architectures.

Interprocedural constant propagation by itself does not greatly increase the performance of applications. The benefits result from enabling the following optimizations to be more effective:

—constant folding and propagation;

—dead-code elimination, including test elimination;

—loop strip mining for vectorization;

—dependency analysis for vectorization, parallelization, and memory hierarchy optimization; and

—loop interchange for vectorization, parallelization, and memory hierarchy optimization.

The last two optimizations are the most important for high-performance computers. The desire to improve their effectiveness was the chief motivation for doing interprocedural constant propagation.

In this paper a constant is one of the following:

—a literal whose value is determined at compile time;

—a literal expression that can be evaluated at compile time; or

—a name/literal value/source location tuple, where the compiler can determine that, on all execution paths that could reach the location, the name will always have the corresponding literal value.

A procedure is a FORTRAN subprogram or a C function. Substitution of FORTRAN statement functions is performed by front end and is outside the scope of this paper.

The interprocedural constant propagation algorithm identifies those argument and global variables that have a single constant value, known at compile time, at all call sites of a given procedure. It also records those argument and global variables that have multiple constant values, all known at compile time, at different call sites of a given procedure. The cloning analysis algorithm uses this information to determine whether one or more copies of a procedure should be made in order to allow propagation of each distinct combination of constant values into the copies.

2. IMPLEMENTATION

The following data structures are created prior to interprocedural constant propagation, and are used as input to this and other algorithms:

—call graph, represented as lists of predecessors and successors for each procedure;

—list of initialized globals;

—global variable/formal argument aliasing information;

—formal/actual argument type mismatch information;

—bit vectors indicating which variables may be assigned by procedure calls; and

—an abstraction of each procedure, consisting of a flow-graph dominator tree containing the following:
  —procedure calls, with symbolic expressions for actual arguments;
  —assignments to scalars, with symbolic expressions for right-hand-side values;
  —loop heads; and
  —loop tails.

The algorithm assumes at the start that all variables have unknown, varying values. It then discovers those variables that have constant values through the algorithm whose pseudocode is given in Figure 1.

The output of the interprocedural constant propagation algorithm is a list of symbol/value pairs for each procedure. These variables always have the same constant value upon invocation of the procedure.

The current implementation propagates all floating-point and integer types, both signed and unsigned, supported by the CONVEX FORTRAN and C compilers and the CONVEX C-series architecture. It also propagates the FORTRAN complex data types. Although we have seen character constants that could be propagated, our implementation does not currently do this. Such constants rarely contribute to optimizing the scientific FORTRAN and C codes we investigated. The algorithm does not propagate address constants either. This would be useful, since indirect calls could be converted to direct calls. This would make it possible to substitute the called procedure inline, if that optimization was appropriate.

The Application Compiler will propagate constants to procedures that are called indirectly, since it creates a list of potential callees for each call site. Aliasing between variables of different types does not prevent constant propagation as long as only one of the variables is assigned values.

3. USING PROPAGATED CONSTANTS

The primary use for the results of *interprocedural* constant propagation is for increasing the effectiveness of the *procedural* constant propagation algorithm. Procedural constant propagation is performed after the scalar optimizer has converted the procedure into a graph representation and has performed data-flow analysis on it.

Each basic block is represented as a directed acyclic graph. Within basic blocks, data-flow arcs connect operators to operands. Use-definition arcs constrain the order in which memory references occur. Basic-block DAGs are the nodes in a control-flow graph. Within the control-flow graph, arcs show possible control flow between blocks and identify loop structures. The control flow graph is reducible and contains use-definition information. Such information tells, for a given use of a variable, what definitions of that variable have a path that reaches the use.

The procedural constant propagation algorithm in the scalar optimizer substitutes uses of variables with uses of constants if the variable has the

```
done = FALSE
while done == FALSE
  for each procedure P in call graph in forward topological sort order
    set "current constants" equal to "constants on entry" for P
    for each node of dominator tree of P in forward topological sort order
      symbolic_interpretation(node)
    end
    set "constants on exit" equal to "current constants" for P
  end
  if any new constants found then
    for each procedure P in call graph in reverse topological sort order
      set "current constants" equal to "constants on entry" for P
      for each node of dominator tree of P in forward topological sort order
        symbolic_interpretation(node)
      end
      set "constants on exit" equal to "current constants" for P
    end
  else
    done = TRUE
  endif
end
symbolic_interpretation(node):
case assignment:
  add <lvalue/rvalue> pair to "current constants" for procedure
case call:
  merge "current constants" for caller procedure with
    "constants on entry" for called procedure
  merge "constants on exit" for called procedure with
    "current constants" for caller procedure
case loop:
  remove all <symbol/constant> pairs from "current constants" for which
    the symbol may be assigned directly or indirectly within loop
end
```

Fig. 1. Interprocedural constant propagation algorithm.

same value on all control-flow paths. It finds constants to propagate from three sources:

(1) assignments,
(2) static initializations (if the procedure is the "main" procedure), and
(3) constants on entry.

The last source is available only if interprocedural constant propagation is performed. This algorithm provides input to other optimizations, such as constant folding, control-flow simplification, etc.

The target architecture for this compiler (CONVEX C-series) provides scalar instruction formats that allow short and word integer and single-precision floating-point operands as immediate fields. This mitigates, to some extent, the problem created by giving the register allocator more constants than it can put in a relatively small register file. Double-precision floating-point operands and scalar operands for vector instructions must be in registers. We tuned the priorities of the register allocator to make good choices when faced with a greatly increased number of constants provided by our algorithm.

The results of interprocedural constant propagation are also used by the interprocedural algorithms that follow it. Automatic inlining is performed by identifying call sites that are executed frequently, and procedures that cause the smallest code expansion when substituted inline. Propagated constants often provide exact loop limits. This makes the compile-time frequency estimation technique nearly as effective as using execution-time profile data, which can be supplied by the user when available.

Interprocedural array section analysis also uses the results of constant propagation. This phase analyzes and summarizes the reference patterns for the elements of arrays. Sections are initially created on a per-procedure basis. Before the interprocedural algorithm executes, a pass goes through all of the input array sections and substitutes interprocedural constants for variables referenced in the section descriptors. This increases the subsequent effectiveness and efficiency of array section analysis.

4. RELATED WORK

Torczon [1985] described a variety of algorithms that can be used for interprocedural constant propagation:

—Inline all nonrecursive calls, and then perform procedural constant propagation.

—Iterate over the call multigraph. Perform symbolic procedural constant propagation of each called procedure. Use the known constant-valued formal parameters and global variables to determine which potentially constant actual arguments should be transferred to formal parameters of called procedures.

—Iterate over the call multigraph. Perform procedural constant propagation of each called procedure. Use the results to determine which actual arguments have constant values that should be transferred to formal parameters of called procedures.

—Iterate over the call multigraph. Use DEF-USE chains previously computed to transfer constant formal parameters to actual arguments passed to called procedures.

—Use a map that associates each formal parameter with every formal to which it may be bound, directly or indirectly. Perform procedural constant propagation on each procedure. Propagate constant-valued actual arguments to all associated formal parameters that have the same constant value.

—Perform procedural constant propagation. Transfer constant actual arguments to the directly associated formal parameters.

Torczon characterized these algorithms in terms of the types of constants potentially propagated, the complexity of the algorithm, and the interprocedural information required to perform it.

Our algorithm iterates over the call multigraph, as do three of the methods Torczon mentioned. Our solution algorithm for each call combines aspects of each of the iterative methods she described.

Burke and Cytron [1986] discussed interprocedural constant propagation in the context of dependency analysis for parallelization. They asserted that it is essential to propagate constants to scalar variables that are used in dependence equations. They did not provide for procedure cloning, stating that "such splitting can become very expensive." Our statistics show that uncontrolled cloning can be very expensive, but that targeted cloning increases the number of useful constants efficiently. Their approach to interprocedural analysis did not support recursion. Ours does.

Callahan et al. [1986] implemented an interprocedural constant propagation algorithm in the context of their FORTRAN vectorization system (PFC). It performs interprocedural analysis after analyzing each individual procedure and before attempting to vectorize those procedures.

Callahan et al. represented the value of each procedure parameter (including global variables) using a symbolic transition function. These functions are evaluated as each procedure is visited in topological sort order. The transition function for each parameter and return value at each call contain a general expression tree. Our algorithm does not use general expressions, but is currently limited to linear forms $Ax + By + C$, or degenerate cases thereof. This form represents most of the subscript values we were interested in, and is more efficient to evaluate and to store. We could easily substitute a general expression tree if it was useful. Their implementation supports propagation of static initializations, and return values back out of called procedures. Our implementation supports propagation of static initializations. We support propagation of values from a called procedure to a calling procedure through global variables, but not through function return values.

Wegman and Zadeck [1991] discussed *inter* procedural constant propagation after presenting four increasingly powerful algorithms for *intra* procedural constant propagation. They did not perform procedure cloning and conceded that "the number of constants propagated across a procedure boundary may be small, since the only constants propagated across a procedure boundary are those having the same constant value at all call sites." Our observations substantiate their conjecture.

Wegman and Zadeck felt that one of the strong points of their approach was that they feed back the results of constant propagation into dead-code elimination and alias analysis. This feedback might remove procedure calls and assignments, thus opening up additional opportunities for constant propagation. The other approaches, including ours, do not feed the results of interprocedural constant propagation back to call graph analysis and alias analysis.

Table I. General Application Characteristics

Application	Total procedures	Total source lines	Calls per procedure	
			Average	Maximum
Ballistics 1	222	19,292	9.4	144
Ballistics 2	210	28,971	4.8	76
Circuit sim	238	60,774	17.3	231
Comp chem 1	158	32,031	5.2	137
Comp chem 2	338	79,552	5.7	78
Finite elem 1	1,398	207,619	17.2	251
Fluid dynam 1	38	11,506	3.0	16
Fluid dynam 2	629	31,387	10.8	248
Fluid dynam 3	263	52,326	5.5	106
Fluid dynam 4	31	3,175	2.0	34
Mech CAE 1	43	9,426	2.2	23
Mech CAE 2	223	24,667	5.5	54
Mech CAE 3	422	108,607	9.8	112
Mole model 1	578	71,678	11.0	430
Mole model 2	58	6,794	3.0	26
Nuclear sim	373	87,973	3.5	47
Oper research	102	14,882	2.0	21
Reserv model 1	115	16,508	7.9	99
Reserv model 2	1,061	210,019	8.8	339
Reserv model 3	688	148,915	9.5	237
Reserv model 4	293	88,924	4.0	85
Signal proc	307	86,306	7.3	97
Simulation 1	86	24,334	7.4	161
Simulation 2	221	30,464	9.7	70
Struct mech 1	648	45,854	3.8	224
Struct mech 2	989	195,132	4.9	101
Weather sim 1	54	20,776	5.8	78
Weather sim 2	400	68,127	9.4	199
Weather sim 3	77	15,290	1.7	19
Average	353.9	62,114.1	6.8	129.0

The Wegman and Zadeck approach applies to a wide range of programming languages. The other two papers addressed only FORTRAN. Our implementation supports both FORTRAN and C, and should support any similar high-level language. None of these papers provides any empirical evidence on the effectiveness of their approach when compiling actual application programs.

5. EMPIRICAL STUDIES

5.1 Applications

Table I describes the relevant characteristics of the applications that were studied for this paper. All of them were coded primarily in FORTRAN, with occasional calls to C code for system interfaces.

Table II. Constants Available for Propagation

Application	Total scalar globals	Scalar formals		Available global constants		Available formal constants	
		Average	Maximum	Average	Maximum	Average	Maximum
Ballistics 1	545	3.8	33	213.0	233	0.7	7
Ballistics 2	294	3.5	28	16.6	32	0.4	2
Circuit sim	609	2.0	10	15.2	18	0.6	4
Comp chem 1	136	2.8	12	1.8	9	0.2	4
Comp chem 2	514	2.8	14	25.6	52	0.8	7
Finite elem 1	2,628	2.2	19	10.5	124	0.7	4
Fluid dynam 1	198	1.4	5	36.9	58	0.5	2
Fluid dynam 2	2,190	3.7	24	2.1	19	0.4	10
Fluid dynam 3	390	5.0	21	49.2	127	0.1	3
Fluid dynam 4	60	4.3	15	2.4	9	0.3	3
Mech CAE 1	22	5.2	19	6.8	8	0.7	4
Mech CAE 2	177	2.2	19	42.6	86	0.6	6
Mech CAE 3	349	2.7	17	17.6	34	0.5	5
Mole model 1	849	4.5	37	0.5	65	0.7	11
Mole model 2	201	2.5	8	3.0	10	0.3	2
Nuclear sim	893	2.1	21	2.1	4	0.3	4
Oper research	415	3.1	13	0.0	0	1.3	5
Reserv model 1	673	1.8	10	0.5	25	0.5	2
Reserv model 2	978	3.0	28	0.2	55	0.8	8
Reserv model 3	1,341	4.2	62	0.4	57	0.9	9
Reserv model 4	1,243	2.9	21	8.0	61	0.4	3
Signal proc	457	4.0	22	1.5	63	1.2	22
Simulation 1	233	1.6	8	68.3	71	0.2	4
Simulation 2	649	1.8	15	0.8	177	0.5	7
Struct mech 1	1,103	1.9	12	6.6	21	0.3	7
Struct mech 2	1,386	2.9	21	1.2	98	0.7	8
Weather sim 1	404	10.9	47	58.0	76	2.9	8
Weather sim 2	652	3.1	18	34.0	57	0.5	5
Weather sim 3	693	1.8	9	5.3	18	0.3	2
Average	676.1	3.1	19.6	21.0	55.6	0.6	5.6

The first column of Table I identifies the application. The second column lists the total number of procedures in the application. The third column gives the total number of source lines in the application. The next two columns list the mean and maximum number of procedure calls per procedure.

5.2 Available Constants

Table II gives statistics on the number of constants that are available on entry to procedures, as computed by our algorithm. In general, there are many more global variables that are known to be constant on entry to a procedure than those that are actually used in any given procedure.

The first column identifies the application. The second column gives the total number of COMMON (global) scalar variables in the application. This is

a count of unique scalar locations. It does not include aliases. The next two columns give the mean and maximum number of formal arguments per procedure. The next two columns give the mean and maximum number of constants available through scalar COMMON variables. The last two columns list the mean and maximum number of constants available through formal arguments. These numbers were generated with inline substitution turned off, and procedure cloning for the default case (integer variables that feed loop bounds or subscripts).

Our applications range in number of scalar global variables from 22 to 2628, with a median of 545. In 12 of the applications, our algorithm determined that 10 percent or more of all global variables were constant on entry to at least one procedure. In a further 7 of the applications, we found that between 5 and 10 percent of all global variables were constant on entry to at least one call.

5.3 Constant Propagation and Other Optimizations

Table III gives totals of constants propagated when procedure cloning and procedure inlining are turned on or off. The purpose of performing procedure cloning is to increase the opportunities for constant propagation. The numbers reported in this table are the number of nodes in a data-flow-graph representation of the procedures that were converted from variable USE nodes to CONSTANT nodes. This is different from the number of unique variable names that corresponded to constants.

By definition, inline substitution reduces the number of interprocedural constants propagated explicitly, since the procedure boundary is removed. Of course, inlining propagates these constants implicitly, but they no longer appear in constant propagation statistics.

The first column identifies the application. The second column gives the total when inlining is turned off and when cloning is performed for integer variables that feed loop bounds or subscripts. The third column gives the total number of constants propagated when procedure inlining and procedure cloning are both turned off. The next column lists the total when inlining is turned off and when cloning is performed whenever it can create the opportunity for propagating a constant. The last column gives the total constants propagated when inlining and procedure cloning are both performed at the default level used by the Application Compiler.

Comparing the first and second columns of numbers shows how important procedure cloning is in obtaining usable constants. In 14 of the applications, more than 50 percent of the propagated constants were due to cloning. In 4 additional applications, more than 25 percent of the propagated constants were due to cloning.

The wisdom of targeting cloning at specific variables that may improve other optimizations is demonstrated by subtracting the values in the second column of numbers from the first, and subtracting the first column from the third. The first result is the number of constants that are added due to targeted procedure cloning. The second result is the maximum number of constants that could be propagated, with uncontrolled cloning.

Table III. Constant Propagation and Other Optimizations

	Total number of constants propagated			
Application	Inline none/clone default	Inline none/clone none	Inline none/clone all	Inline medium/clone default
Ballistics 1	914	705	2,720	973
Ballistics 2	700	637	1,452	694
Circuit sim	895	48	2,429	380
Comp chem 1	76	41	173	75
Comp chem 2	1,367	1,069	8,286	1,367
Finite elem 1	15,806	150	Too big	NA
Fluid dynam 1	149	137	334	149
Fluid dynam 2	215	108	1,584	NA
Fluid dynam 3	1,119	1,103	2,753	1,235
Fluid dynam 4	15	12	33	16
Mech CAE 1	109	80	115	104
Mech CAE 2	1,076	629	28,288	1,064
Mech CAE 3	364	182	3,279	506
Mole model 1	1,312	1,112	4,773	1,300
Mole model 2	70	63	97	70
Nuclear sim	272	42	750	270
Oper research	105	25	348	105
Reserv model 1	120	37	314	67
Reserv model 2	2,260	341	Too big	2,228
Reserv model 3	1,277	285	Too big	1,264
Reserv model 4	1,392	646	2,681	1,298
Signal proc	1,010	196	2,133	709
Simulation 1	86	84	1,877	116
Simulation 2	27	17	536	26
Struct mech 1	214	110	508	137
Struct mech 2	1,028	432	Too big	958
Weather sim 1	6,998	2,902	7,164	6,715
Weather sim 2	1,832	1,640	4,506	1,819
Weather sim 3	250	185	620	238
Average	1,368.6	433.9	3,110.1	884.6

5.4 Variables and References

Table IV correlates the number of distinct variables and the number of variable references converted to constant references. The CONVEX compilers represent user procedures in a data-flow-graph representation. The data in this table count variable references after this representation has been created and after standard scalar optimizations have been performed.

The first column identifies the application. The remainder of the table compares the number of variables that were determined to be constant with the number of variable references that were converted to a constant. The first three numeric columns provide this comparison when procedure inlining is turned off. The last three numeric columns provide this comparative data when inlining is performed at the default level.

In all but five of the applications, the number of unique interface variables determined to be constant and the number of variable references actually

Table IV. Variables and Uses Absorbing Constants

	Inline none			Inline medium		
Application	Number of variables	Number of uses	Average per variable	Number of variables	Number of uses	Average per variable
Ballistics 1	586	914	1.6	580	973	1.7
Ballistics 2	211	700	3.3	210	694	3.3
Circuit sim	175	895	5.1	172	380	2.2
Comp chem 1	35	76	2.2	34	75	2.2
Comp chem 2	496	1,367	2.8	495	1,367	2.8
Finite elem 1	1,417	15,806	11.2	NA	NA	NA
Fluid dynam 1	85	149	1.8	85	149	1.8
Fluid dynam 2	77	108	1.4	NA	NA	NA
Fluid dynam 3	278	1,119	4.0	301	1,235	4.1
Fluid dynam 4	13	16	1.2	13	16	1.2
Mech CAE 1	40	109	2.7	35	104	3.0
Mech CAE 2	283	1,076	3.8	272	1,064	3.9
Mech CAE 3	219	364	1.7	284	506	1.8
Mole model 1	340	1,312	3.9	318	1,300	4.1
Mole model 2	20	70	3.5	19	70	3.7
Nuclear sim	85	272	3.2	84	270	3.2
Oper research	35	105	3.0	35	105	3.0
Reserv model 1	87	120	1.4	34	67	2.0
Reserv model 2	528	2,260	4.3	490	2,228	4.5
Reserv model 3	426	1,277	3.0	413	1,264	3.1
Reserv model 4	135	1,392	10.3	109	1,298	11.9
Signal proc	424	1,010	2.4	368	709	1.9
Simulation 1	56	86	1.5	62	116	1.9
Simulation 2	21	27	1.3	20	26	1.3
Struct mech 1	163	214	1.3	109	137	1.3
Struct mech 2	581	1,028	1.8	509	958	1.9
Weather sim 1	369	6,998	19.0	344	6,715	19.5
Weather sim 2	443	1,832	4.1	439	1,819	4.1
Weather sim 3	33	250	7.6	32	238	7.4
Average	255.4	1,365.1	3.8	217.3	884.6	3.8

converted to constants both decreased when inline substitution was performed. These results fit with intuition, since procedure inlining reduces the total number of arguments passed in the application. When inlining is performed, constant actual arguments are substituted for formal arguments, but these are not counted as interprocedural constants propagated by our compiler.

The five exceptional cases are a result of the interaction between constant propagation and inlining. If a procedure is substituted inline for more than one call, the total number of variable references in the application goes up. This makes it possible for the number of variable references converted to constants to go up, even though the number of interface variables has gone down. Even the number of interface variables found to be constant can go up when inlining is performed. This happens when a variable that could be

Table V. Types of Interprocedural Constant Propagation

Application	Literal constant arguments	Evaluated constant arguments	Percent arguments found constant	Assigned to common variable	Initialized in common block	Percent globals found constant
Ballistics 1	127	84	24.8	58	416	87
Ballistics 2	28	167	25.1	171	0	58
Circuit sim	134	6	34.4	1	35	6
Comp chem 1	22	5	5.4	8	0	6
Comp chem 2	198	61	28.3	7	247	49
Finite elem 1	1,245	154	54.2	48	6	2
Fluid dynam 1	9	11	42.6	72	0	36
Fluid dynam 2	132	15	6.2	2	0	0
Fluid dynam 3	15	80	1.7	211	49	67
Fluid dynam 4	10	0	7.6	3	0	5
Mech CAE 1	11	23	11.8	8	0	36
Mech CAE 2	106	10	26.6	173	0	98
Mech CAE 3	186	14	11.6	23	0	7
Mole model 1	309	99	9.8	2	0	0
Mole model 2	14	2	8.9	5	0	2
Nuclear sim	69	11	11.0	5	0	1
Oper research	29	10	13.4	0	0	0
Reserv model 1	80	7	43.0	0	0	0
Reserv model 2	493	58	18.4	3	0	0
Reserv model 3	388	41	14.3	11	0	1
Reserv model 4	91	45	16.0	27	0	2
Signal proc	349	72	14.3	12	0	3
Simulation 1	9	1	6.8	21	26	20
Simulation 2	18	3	4.9	0	0	0
Struct mech 1	120	37	12.7	0	28	3
Struct mech 2	557	39	16.5	2	2	0
Weather sim 1	221	199	60.0	140	0	35
Weather sim 2	141	83	18.7	225	44	41
Weather sim 3	14	12	18.3	13	0	2
Average	170.8	45.0	8.8	41.7	28.4	10.4

forced constant through a clone is now used in a context in which the cloning algorithm considers cloning to be profitable.

5.5 Types of Interprocedural Constant Propagation

Table V breaks down the constants propagated by their origin. The first column identifies the application. The second column gives the total number of constants propagated that were literal constants (any type) passed as arguments. The next column lists the total number of constants propagated that were variables passed as arguments. The fourth column gives the sum of the previous two columns as a percentage of the total number of formal arguments. The next column gives the number of values that were assigned to COMMON variables. The following column gives the number of values that were initialized in a COMMON block. The last column gives the percentage of global variables that were found to have a constant value on entry to at least

Table VI. Constant Values Propagated

Application	Total 0	Total 1	Total 0.0	Total 1.0	Total −20..−1	Total 2..20	Other integer	Other float
Ballistics 1	67	78	34	60	2	73	14	258
Ballistics 2	2	4	0	0	4	165	36	0
Circuit sim	4	66	0	0	0	63	15	27
Comp chem 1	1	12	0	0	2	16	4	0
Comp chem 3	26	41	33	63	0	162	84	87
Finite elem 1	35	137	1	0	46	884	306	8
Fluid dynam 1	14	5	2	3	4	19	27	11
Fluid dynam 2	NA	NA	NA	NA	NA	NA	NA	NA
Fluid dynam 3	5	12	0	14	0	84	128	35
Fluid dynam 4	3	5	3	2	0	0	0	0
Mech CAE 1	4	7	0	1	0	23	4	1
Mech CAE 2	7	30	5	0	2	185	54	0
Mech CAE 3	27	68	2	1	13	60	46	2
Mole model 1	58	29	1	1	6	114	123	8
Mole model 2	8	4	1	0	0	6	1	0
Nuclear sim	17	7	4	1	1	11	43	1
Oper research	1	7	0	0	0	6	21	0
Reserv model 1	1	12	1	0	1	38	34	0
Reserv model 2	85	134	3	0	8	211	86	1
Reserv model 3	61	83	4	0	7	203	67	1
Reserv model 4	12	49	1	3	1	56	11	2
Signal proc	33	115	0	2	11	208	55	0
Simulation 1	1	0	7	3	0	4	1	40
Simulation 2	2	5	4	0	0	10	0	0
Struct mech 1	15	17	4	0	0	83	42	2
Struct mech 2	45	95	0	0	4	314	121	2
Weather sim 1	1	89	0	0	0	62	106	111
Weather sim 2	18	98	5	11	1	242	25	43
Weather sim 3	10	9	1	0	2	10	1	0
Average	19.4	42.0	4.0	5.7	4.0	114.2	50.2	22.1

one procedure. These numbers were generated with inline substitution turned off and with procedure cloning for the default case (integer variables that feed loop bounds or subscripts).

The global variable numbers demonstrate the importance of having a constant propagation algorithm that detects constant global variables as well as constant arguments. There are seven applications shown where more than 50 percent of the total propagated constants were global variables. There are a further three applications where between 25 and 50 percent of the propagated constants were global variables. As the fourth column of numbers shows, most applications did not statically initialize global variables that remained constant. In six applications, however, there were dozens and even hundreds of such variables.

5.6 Constant Values Propagated

Table VI breaks down the constants propagated by their value. The first column identifies the application. The next two columns give the total num-

ber of constants propagated to integer variables that had the values 0 or 1. The following two columns give the total number of constants propagated to floating-point variables that had the values 0.0 or 1.0. The next column gives the total number of constants propagated to integer variables that had values from −20 to −1. The next column gives the total number of constants propagated to integer variables that had values from 2 to 20. The following column provides the total number of constants propagated that had integer values outside the range −20 to 20. The last column gives the total number of constants propagated to floating-point variables that did *not* have the values 0.0 or 1.0. These numbers were generated with inline substitution turned off and with procedure cloning for the default case (integer variables that feed loop bounds or subscripts).

The values 0 and 1, both integer and floating point, are obviously a large fraction of the total set of unique values propagated. Small integer values, in the range −20 to +20, are also a large subset of the values propagated. The most popular integer constant outside this range was 80, which is the length of a punched card.

Seven applications had more than two dozen distinct propagated real values; however, most had none at all. This is partially the result of our cloning strategy, which targets values used as loop limits and array subscripts.

5.7 Effects on Procedural Constant Propagation

Interprocedural constant propagation does not improve the performance of applications by itself. It depends on procedural constant propagation to actually change the procedure. As the procedure representation is simplified, the optimizations we are actually targeting, such as dependence analysis and loop interchange, are made more effective. The purpose of the next table is to show how interprocedural constant propagation affects procedural constant propagation.

Table VII provides information on the increase in procedural constant propagation resulting from interprocedural constant propagation. The first column identifies the application. The second and third columns give the total number of variable uses changed to constants when interprocedural constant propagation is turned off or on, respectively. These uses are substituted after basic blocks have been converted to directed acyclic graphs and after redundant use elimination has been performed on a procedure basis. Both of these optimizations reduce the number of distinct uses of a variable. If we were simply substituting constant values for variables in the original program text, the numbers would be higher. The next column gives the total number of additional constants propagated within a procedure when interprocedural constant propagation is turned on. The last column gives the percent increase of constants propagated within a procedure when interprocedural constant propagation is turned on. These numbers were generated with inline substitution turned off and with procedure cloning for the default case (integer variables that feed loop bounds or subscripts).

Table VII. Increase in Procedural Constant Propagation

Application	Total uses changed to constants		Total increase	Percent increase
	ICP off	ICP on		
Ballistics 1	358	1,450	1,092	305
Ballistics 2	1,538	2,247	709	46
Circuit sim	3,131	3,746	615	20
Comp chem 1	2,038	2,126	88	4
Comp chem 2	2,296	2,813	517	23
Finite elem 1	13,220	19,105	5,885	45
Fluid dynam 1	127	336	209	165
Fluid dynam 2	749	1,024	275	37
Fluid dynam 3	2,004	2,082	78	4
Fluid dynam 4	369	387	18	5
Mech CAE 1	803	928	125	16
Mech CAE 2	707	1,106	399	56
Mech CAE 3	2,667	3,124	457	17
Mole model 1	2,904	4,446	1,542	53
Mole model 2	285	360	75	26
Nuclear sim	2,436	2,747	311	13
Oper research	388	509	121	31
Reserv model 1	590	1,055	465	79
Reserv model 2	5,409	7,650	2,241	41
Reserv model 3	4,120	6,147	2,027	49
Reserv model 4	3,974	4,916	942	24
Signal proc	832	1,667	835	100
Simulation 1	361	384	23	6
Simulation 2	488	537	49	10
Struct mech 1	1,613	1,822	209	13
Struct mech 2	3,590	5,200	1,610	45
Weather sim 1	1,489	3,184	1,695	114
Weather sim 2	2,858	4,933	2,075	73
Weather sim 3	1,287	1,469	182	14
Average	2,159.7	3,017.2	857.6	39.7

5.8 Procedure Cloning

Procedure cloning is a special interprocedural optimization performed solely for the purpose of increasing the number of propagated constants. Procedures are cloned when constant propagation determines that the value of an interface variable is constant at a given call site, but that it has a different constant value or that it has a variable value at another call site to the same called procedure. When such a condition occurs and when the situation is determined to be profitable, a complete copy of the called procedure is made, the call site is changed, and the constant is propagated into the clone.

Procedure cloning has the potential to cause an explosion in the size of the generated code. For this reason, it is only considered profitable when there is a potential to propagate constants that will contribute to other optimizations. The compiler currently creates a clone when it identifies the potential to propagate a constant into a variable that is one of the following:

Table VIII. Procedure Cloning

Application	Total clones created	Total cloned procedures	Clones of procedure		Percent procedures cloned	Percent procedures increased
			Average	Maximum		
Ballistics 1	30	10	3.0	11	4.5	14
Ballistics 2	13	6	2.2	7	2.9	6
Circuit sim	59	9	6.6	45	3.8	28
Comp chem 1	13	7	1.9	6	4.4	8
Comp chem 2	91	17	5.4	41	5.0	27
Finite elem 1	951	117	8.1	119	8.4	68
Fluid dynam 1	4	2	2.0	3	5.3	11
Fluid dynam 2	51	20	2.0	17	3.2	8
Fluid dynam 3	5	5	1.0	1	1.9	2
Fluid dynam 4	1	1	1.0	1	3.2	3
Mech CAE 1	10	2	5.0	5	4.7	23
Mech CAE 2	58	12	4.8	18	5.4	26
Mech CAE 3	57	17	3.4	19	4.0	14
Mole model 1	62	30	2.0	12	5.2	11
Mole model 2	3	2	1.5	2	3.4	5
Nuclear sim	25	11	2.3	6	2.9	7
Oper research	19	1	19.0	19	1.0	19
Reserv model 1	59	7	8.4	26	6.1	51
Reserv model 2	226	84	2.7	16	7.9	21
Reserv model 3	112	28	4.0	16	4.1	16
Reserv model 4	30	17	1.9	11	5.8	10
Signal proc	158	40	4.0	18	13.0	51
Simulation 1	1	1	1.0	1	1.2	1
Simulation 2	7	5	1.4	3	2.3	3
Struct mech 1	92	12	7.7	27	1.9	14
Struct mech 2	193	80	2.4	12	8.1	20
Weather sim 1	33	11	3.0	9	20.4	61
Weather sim 2	31	20	1.6	5	5.0	8
Weather sim 3	7	5	1.4	3	6.5	9
Average	80.0	19.3	3.7	16.0	5.6	23.4

—a formal argument or global variable that is used directly or indirectly to control a loop (start, stop, or step value),

—a formal argument or global variable that is used directly or indirectly as a part of an array subscript expression, or

—a formal argument that is used in a comparison in a conditional statement.

The first two criteria were chosen to increase the number of constants propagated that would be used by dependency analysis and loop interchange. The third criterion was chosen to increase the possibilities for dead-code elimination.

Table VIII provides more information on procedure cloning. The first column identifies the application. The second column gives the total number of clones that were created. The third column gives the total number of procedures that were cloned. The next two columns give the mean and maximum number of clones per procedure. The next column gives the per-

centage of all procedures in the application that were cloned. The last column gives the percent increase in the number of procedures in the application after cloning was performed. These numbers were generated with inline substitution turned off and with procedure cloning for the default case (integer variables that feed loop bounds or subscripts).

The percentage of procedures cloned range from 0.98 to 20.37 percent, with a median value of 4.5 percent. All but two of the applications had less than 9 percent of their procedures cloned. We believe these low figures indicate the effectiveness of our targeted approach to cloning and constant propagation.

5.9 Procedure Cloning Choice Effectiveness

The effects of procedure cloning and inline substitution on procedural constant propagation are very similar. The question then arises as to how effective the Application Compiler is at choosing procedures to clone. It prefers inlining to cloning in general, since inlining eliminates call overhead. The Application Compiler first analyzes calls for potential inlining and then analyzes the remaining calls for potential retargeting to clones.

Analyzing the relationship between calls and clones can help determine whether this simplistic sequencing is sufficient or whether a more complicated sequencing is desired. One alternative would be iterating between cloning and inlining until the solution stabilizes. Clones that have only one call retargeted to them might be better off inlined instead. A high percentage of such clones might suggest that a more complicated sequence could do better by inlining more of these calls instead.

Table IX shows the relationship between calls and clones. The values were computed with automatic inlining turned on. The first column identifies the application. The second column gives the total number of clones created. The third column gives the number of calls that were retargeted to call a clone instead of the original routine. The next column gives the average number of calls that were retargeted to one of its clones for each cloned procedure. The following column lists the total number of clones that had only one call retargeted to them, referred to as "single" clones. The final column shows what percentage of the total clones were "single" clones.

As can be seen in the figures at the bottom of the table, on average, 21.8 percent of all clones were single clones. This suggests that the Application Compiler could do better by inlining those calls instead of retargeting them. On the other hand, the average number of calls retargeted to a clone was 4.14. This indicates that the cloning algorithm is effective, on average, since inlining each of those calls instead would have greatly increased the program size.

5.10 Benchmark Speedups

The applications analyzed in this paper are proprietary to CONVEX customers and third-party software vendors. Thus, our results cannot be reproduced by others. Other researchers can compare their results with ours using Table X. This table shows the relationship between constant propagation and

Table IX. Clone Choice Effectiveness

Application	Total clones created	Total calls to clone	Average calls to clone	Single clones	Percent single clones
Ballistics 1	23	122	5.3	8	35
Ballistics 2	13	74	5.7	6	46
Circuit sim	56	568	10.1	19	34
Comp chem 1	12	92	7.7	5	42
Comp chem 2	86	397	4.6	25	29
Finite elem 1	327	1,473	4.5	125	38
Fluid dynam 1	4	4	1.0	4	100
Fluid dynam 2	12	23	1.9	7	58
Fluid dynam 3	5	12	2.4	4	80
Fluid dynam 4	1	4	4.0	0	0
Mech CAE 1	5	12	2.4	2	40
Mech CAE 2	44	126	2.9	24	55
Mech CAE 3	42	335	8.0	13	31
Mole model 1	42	146	3.5	21	50
Mole model 2	2	10	5.0	0	0
Nuclear sim	24	31	1.3	21	88
Oper research	19	39	2.0	9	47
Reserv model 1	8	10	1.3	6	75
Reserv model 2	185	2,837	15.3	94	51
Reserv model 3	101	371	3.7	66	65
Reserv model 4	12	45	3.8	5	42
Signal proc	107	323	3.0	50	47
Simulation 1	1	1	1.0	1	100
Simulation 2	6	10	1.7	4	67
Struct mech 1	65	173	2.7	25	38
Struct mech 2	152	464	3.1	77	51
Weather sim 1	21	79	3.8	12	57
Weather sim 2	26	169	6.5	17	65
Weather sim 3	7	16	2.3	4	57
Average	46.9	265.5	4.1	21.8	46.4

speedup on selected benchmark codes from the Perfect Club and SPEC92 benchmark suites.

The first column identifies the application. The second column gives the number of constants propagated. The third column gives the number of clones created. The last column gives the percent speedup over the same code compiled with constant propagation and procedure cloning turned off. Procedure inlining was turned off in both cases. The codes were run on a CONVEX C3240 vector processor.

These codes are small and offer relatively little opportunity for interprocedural optimization. The results understate the benefits of these optimizations as seen in real applications. In those cases where the percent change was less than 1 percent, it is listed as 0.

6. CONCLUSIONS

We have compiled nearly 2,000,000 lines of scientific and engineering FORTRAN code and analyzed the availability of interprocedurally propa-

Table X. Benchmark Speedup

Application	Total constants propagated	Total clones created	Percent speedup
Perfect AP	29	6	0
Perfect LG	13	5	0
Perfect LW	127	1	3
Perfect MT	24	1	0
Perfect NA	26	6	2
Perfect OC	4	1	0
Perfect SD	71	8	0
Perfect SM	151	13	0
Perfect SR	34	2	0
Perfect TF	24	1	0
Perfect WS	110	18	13.4
SPEC92 doduc	226	10	2.3
SPEC92 fpppp	75	2	0
SPEC92 hydro2d	70	3	0
SPEC92 mdljdp2	138	35	0
SPEC92 mdljsp2	64	36	0
SPEC92 su2cor	299	30	0
SPEC92 wave5	403	13	0

gated constants. We have shown that an aggressive constant propagation algorithm, coupled with procedure cloning, will expose many global variables and arguments that are constant on entry to a procedure. Application developers can write highly modular systems that refer to values symbolically, rather than literally, confident in the knowledge that compiler technology can optimize their code effectively.

ACKNOWLEDGMENTS

Ken Kennedy of Rice University pioneered this field and gave us helpful advice. Randall Mercer developed the initial design for the interprocedural optimizer. The final design and implementation of the interprocedural optimizer were done by the authors, Jon Loeliger, Mark Seligman, and Matthew Diaz. Henry Baker of Nimble Computer Corporation suggested a number of ways to improve the paper, as did the anonymous referees.

REFERENCES

BURKE, M., AND CYTRON, R. 1986. Interprocedural dependence analysis and parallelization. In *Proceedings of SIGPLAN 86 Symposium on Compiler Construction.* ACM, New York, 162–175.

CALLAHAN, D., COOPER, K. D., KENNEDY, K. W., AND TORCZON, L. M. 1986. Interprocedural constant propagation. In *Proceedings of SIGPLAN 86 Symposium on Compiler Construction.* ACM, New York, 152–161.

TORCZON, L. 1985. Compilation dependences in an ambitious optimizing compiler. Tech. Rep. TR85-21, Rice Univ., Houston, Tex., 59–79.

WEGMAN, M. N., AND ZADECK, F. K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13,* 2, 181–210.

Received August 1991; revised January and October 1993; accepted November 1993