AD-A277 662

Machine Assisted Implementation of Complex Algorithms and Labor Intensive Software - Closing Report

DTIC
ELECTE
MAR 29 1994
S E D

Principal Investigator Name: Robert A. Paige
PI Institution: New York University/Courant Institute
PI Phone Number: 212-998-3512
PI E-mail Address: paige@cs.nyu.edu
Grant Title: Machine Assisted Implementation of Complex Algorithms and Labor Intensive Software
Grant Number: N00014-90-J-1890
ONR Scientific Officer: Ralph Wachter
Grant Period: Mar. 1, 1990 - Sept. 10, 1993

1. DESCRIPTION AND OBJECTIVES OF RESEARCH AND SIGNIFICANT RESULTS DURING THE GRANT PERIOD

Our project stated the following broad goals:

a to design a practical tool capable of automating major aspects of programming - essentially, a generalization of YACC [26] and MACSYMA [28] to facilitate implementation of a wide class of complex nonnumerical algorithms (in addition to parsing);

b to design and implement complexity based specification languages;

c to integrate problem specification, program design, verification, and analysis within a single unified framework;

d to design and efficiently implement pattern directed rule systems for semantic analysis;

e to make it easier to teach and understand algorithms and software engineering.

All of the goals stated in the original proposal were fulfilled. The specific achievements during the grant period may be divided into four categories described below.

1.1. Transformational Methodology

Underlying our transformational methodology is the hypothesis that much of the difficulty involved in implementing complex nonnumerical algorithms is due to human factors that are unrelated to any inherent problem complexity and can be solved by specific transformations implemented in APTS. These factors stem largely from the extensive level of precise detail needed (1) to craft efficient program loops (2) to write the essential imperative bookkeeping operations that serve to maintain program invariants, and (3) to implement tedious but necessary physical data structures that implement conceptually simpler set and map operations. Although this crafting is error prone if done by hand, our transformations allow us to avoid having to

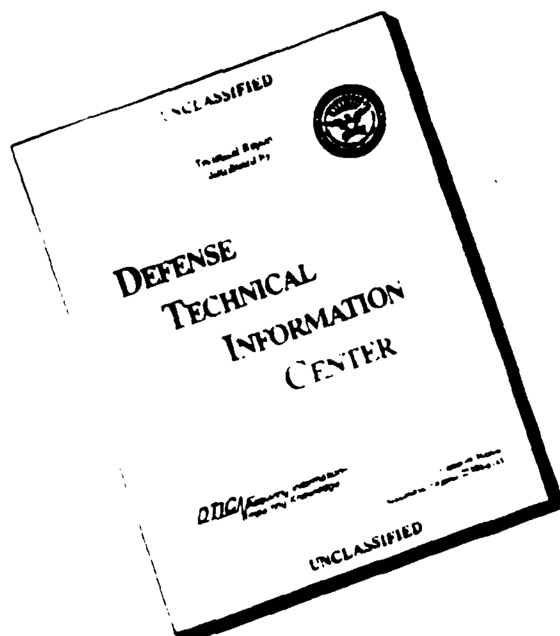
approved for public release

94 3 28 041

94-09475

DTIC QUALITY INSPECTED

DISCLAIMER NOTICE



THIS REPORT IS INCOMPLETE BUT IS THE BEST AVAILABLE COPY FURNISHED TO THE CENTER. THERE ARE MULTIPLE MISSING PAGES. ALL ATTEMPTS TO DATE TO OBTAIN THE MISSING PAGES HAVE BEEN UNSUCCESSFUL.

either pointer or cursor access. These methods show the surprising fact that most of the algorithms that seem to rely on arrays and array access in the main algorithm texts can be realized with the same theoretical performance using just pointers and pointer access. These techniques also have substantial applications to software design and high level language implementation that could, for example, facilitate an efficient implementation of Willard's RCS database language [45,46,47].

Although the linear time language mentioned earlier is a broad based automatic method for producing software, it has sharp ramifications to algorithm design as well. For example, we showed that the very basic problem of Propositional Horn Clause Satisfiability can be expressed in our linear time language, thereby ensuring a new linear time pointer machine solution.

Our two articles with Tarjan on applications and improvements to partition refinement strategies [34,35] and our more recent article with Bloom [4] exhibit the power of the dominated convergence argument for computing fixed points. One of these applications remains the best solution to the basic problem of lexicographic sorting. Two of the other applications to ready simulation and strong bisimulation have had an impact on the field of distributed computing.

Our recent work with Chang on processing regular expressions [14] exhibits our differential transformations applied in a new functional language context. In CPM '92 we gave theoretically and computationally superior solutions for the classical problems of (1) transforming regular expressions into nondeterministic finite state automata, (2) turning nondeterministic finite state automata into deterministic finite state automata, (3) acceptance testing in nondeterministic finite state automata, and (4) regular expression search. These problems are important, because of the many practical applications, including compilation of communicating processes, string pattern matching, model checking, lexical scanning, and VLSI layout design.

1.3. Transformational System

In order to mechanize the transformational methodology, we decided to design a transformational programming system with bottom-up tree pattern matching as a fundamental operation. Since Hoffmann and O'Donnell's seminal work in 1982 [25], bottom-up tree matching has been regarded as important but very difficult, with no obvious way to make either theoretically or computationally tractable.

In CAAP '90 Cai, myself, and Tarjan presented the first theoretical improvement to Hoffmann and O'Donnell, and our paper was selected for publication in a special issue of Theoretical Computer Science on best papers from the conference [11]. This paper included an algorithm for pattern matching, where a newly proposed on-line algorithm outperforms a batch algorithm that was its direct predecessor.

In 1991 we completed the APTS transformational system at the University of Wisconsin. We used APTS to build two compilers for conducting experiments to test the feasibility of these techniques. One of these compilers translates a major fragment of SETL2 into C. The other translates SQ2+ (a strongly typed functional subset of SETL2 augmented with fixed point expressions) into C.

The APTS system comprises about 15,000 lines of SETL2, a nice manageable size encouraging further development. SETL2 is written in C, which makes APTS portable to more

that this is greatly improvable.

These experimental results are highly encouraging. The latest experiments compared benchmarks for implementations of 10 different algorithms. For nine of these algorithms benchmarks were made comparing executable low level SETL2 specifications with C codes automatically generated from them (see Fig. 1). The sizes of the SETL2 specifications ranged from 25 lines for a maximal independent set algorithm to 103 lines for Floyd's implementation [21] of Dijkstra's single source shortest path algorithm [18] with 2-heaps (see Fig. 2). The Generated C programs ranged from 133 lines for graph reachability to 766 lines for Floyd's algorithm. All operations were explicitly written into the source code (e.g. heap operations were written explicitly into the SETL2 specification of Floyd); there were no hidden library routines. Source lines were measured from prettyprinted text (free from comments) using the UNIX wc function.

	SETL2-to-C		SQ2+-to-C	
	C lines / SETL2 lines	SETL2 time / C time	C lines / SQ2+ lines	SETL2 time / C time
reach	5.1	13 → 57	66	23 → 38
maxind	7.6	3700 → 24914		
dijkstra1	9.9	43 → 32		
cycle	6.6	926 → 6550	80	124 → 2183
center	5.6	17 → 27		
aclose	7.6	15 → 38	87	15 → 38
topsort	6.4	600 → 4533		
dijkstra2	7.4	36 → 95		
orbit	9.2			

Figure 1. Ratio Summary

	SETL2	C	SQ2+	C
reach	26	133	4	265
maxind	25	189		
dijkstra1	42	418		
cycle	36	237	5	402
center	42	234		
aclose	54	409	5	435
topsort	46	295		
dijkstra2	103	766		
orbit	26	238		

Figure 2. lines of Source

Accesion For
NTIS CRA&I X
DTIC TAB
Unannounced
Justification per ltr
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

Based on assumptions described in [13], we measure the increase in productivity by the number of source lines of the compiled C program divided by the number of source lines of

faster than the SETL2 program generated from it, and has 75 times more source lines than the specification. It is interesting to note from Fig. 1 that the attribute closure C program generated by our SETL-to-C translator from a much more detailed low level SETL specification runs no faster than the C program generated from the SQ2+ specification. It is also interesting to note that if Kirk Snyder's and our estimates that his SETL2 is about 30 times slower than hand coded C is accurate, then the performance of our generated C program would be comparable to good hand code.

Finally, an experiment was performed to test the potential for scaled up applications. Raytheon, an industrial affiliate of the DOD-sponsored prototech projects, provided the investigators with a moderate sized prototype software package to generate tracks from input radar plot data using an alpha-beta filter with constant scalar coefficients. "The program processes a file of radar plot data and produces a Postscript output file which shows the original plots and program generated radar tracks superimposed on a plan of the radar site." The Raytheon code, which comprises 9 packages and classes and over 1000 lines of source, makes use of every aspect of SETL2. The experiment involved translating the whole main SETL2 module into C, which would interoperate with the remaining SETL2 modules. The productivity improvement (i.e., 477 lines of generated C plus SETL2 interoperability code divided by 64 lines in the SETL2 main module) was 7.7. The overall speedup was 2.6, which was surprisingly good, considering the tremendous overhead in the way that SETL2 currently implements interoperability with C. Overhead is due to added levels of procedure calls, and massive datatype conversion, because only strings are currently allowed to pass between C and SETL2. Recall also, that only one out of the 9 modules was translated into C.

1.5. Related Work

The SETL research program directed by Jack Schwartz from about 1971 until 1989 has had the most positive influence on our work. Schwartz effectively demonstrated the conceptual simplicity of the SETL language with its rich repertoire of universal set theoretic dictions [38], and introduced many important concepts in an attempt to solve the formidable task of implementation and optimization. SETL's mathematical value semantics made costly hidden copy operations a problem. Dynamically allocated aggregate datatypes with arbitrary levels of nesting made memory management difficult. Dynamic weak typing, massive overloading, and the prevalence of operations that relied on associative access compounded the run-time costs and the complexity of an efficient implementation.

The SETL optimizer [37,23] dealt with the hidden copy problem by making a conservative estimate of when a set could be updated destructively or not, and by using a run-time bit (that could be turned on but not off) to indicate this fact. Snyder's SETL2 implementation uses reference counts for the same purpose. The SETL system relied on compacting garbage collection for dynamic allocation, whereas SETL2 uses collections of stacks. SETL used an elaborate hashing mechanism as a default implemention of associative access for arbitrary datatypes. Its optimized data structures combined hashing with bit vectors and plex structures to reduce the amount of replicated data and the expense of hashing [39,17,37]. SETL2 uses tries, and attempts no compile-time optimization. SETL attempted to generate more efficient code using type analysis with an accuracy up to three levels of nesting [44], for sets, maps, and tuples (in order to guarantee

Finally, there were conceptual as well as engineering problems with the SETL data structure optimization. The fundamental data structure that formed the spine of a data structure aggregate was the hash table. Hashing was used heavily in inputting data, in forming the initial optimized data structures, and was used to implement any set that was iterated over. The heuristic nature of the SETL optimizations and the unpredictability of expensive hidden operations was observed by both Shields and Straub.

Our real-time simulation provides new theoretically sound conceptual underpinnings to data structure selection with precise guarantees of speedup. Our transformation is grounded in basic algorithm theory by forming data structure aggregates from arrays and linked lists instead of hash tables. Iteration over a SETL2 set does not force us to resort to hashing. Our routine to input data and to form initial data structures is highly efficient without making use of hashing.

References

1. Bancilhon, F., "Naive Evaluation of Recursively defined Relations," in *On Knowledge-Base Management Systems*, ed. Brodie, M and Mylopoulos, J, pp. 165-178, 1986.
2. Bayer, R., *Query evaluation and recursion in deductive database system*, 1985. unpublished manuscript
3. Beeri, C. and Bernstein, P., "Computational Problems Related to the Design of Normal Form Relation Schemes," *ACM TODS*, vol. 4, no. 1, pp. 30-59, 1979.
4. Bloom, B. and Paige, R., "Computing Ready Simulations Efficiently," in *Proc. First North American Process Algebra Workshop*, ed. A. Zwarico and S. Purushothaman eds., Workshops in Computer Science Series, pp. 119 - 134, Springer-Verlag, 1992.
5. Bouma, Cai, J., Fudos, Hoffmann, C., and Paige, R., "2D Sketcher and Constraint Solver," *accepted for CAD*, 1993.
6. Cai, J. and Paige, R., "Program Derivation by Fixed Point Computation," *Science of Computer Programming*, vol. 11, pp. 197-261, 1988/89.
7. Cai, J., Facon, P., Henglein, F., Paige, R., and Schonberg, E., "Type Transformation and Data Structure Choice," in *Constructing Programs From Specifications*, ed. B. Moeller, pp. 126 - 124, North-Holland, 1991.
8. Cai, J. and Paige, R., "Binding Performance at Language Design Time," in *ACM POPL*, pp. 85 - 97, Jan, 1987.
9. Cai, J. and Paige, R., "Languages Polynomial in the Input Plus Output," in *Algebraic Methodology and Software Technology*, ed. M. Nivat, C. Rattray, T. Rus, and G. Scollo, Workshops in Computing Series, pp. 287 - 302, Springer-Verlag, 1992.
10. Cai, J. and Paige, R., ""Look Ma, No Hashing, And No Arrays Neither"," in *ACM POPL*, pp. 143 - 154, Jan, 1991.
11. Cai, J., Paige, R., and Tarjan, R., "More Efficient Bottom-Up Multi-Pattern Matching In Trees," *Theoretical Computer Science*, vol. 106, no. 1, pp. 21 - 60, Nov. 1992.
12. Cai, J., *A Language for Semantic Analysis*, New York University, 1993. Technical Report
13. Cai, J. and Paige, R., "Towards Increased Productivity of Algorithm Implementation," in *Proc. ACM SIGSOFT*, pp. 71 - 78, Dec. 1993.
14. Chang, C.-H. and Paige, R., "From Regular Expressions to DFA's Using Compressed NFA's," in *Proc. CPM '92*, ed. A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, Lecture Notes in Computer Science, vol. 644, pp. 88 - 108, Springer-Verlag, 1992.
15. Cocke, J. and Kennedy, K., "An Algorithm for Reduction of Operator Strength," *CACM*, vol. 20, no. 11, pp. 850-856, Nov., 1977.
16. Cousot, P. and Cousot, R., "Constructive versions of Tarski's fixed point theorems," *Pacific J. Math.*, vol. 82, no. 1, pp. 43-57, 1979.
17. Dewar, R., Grand, A., Liu S. C., Schwartz, J. T., and Schonberg, E., "Program by Refinement as Exemplified by the SETL Representation Sublanguage," *TOPLAS*, vol. 1, no. 1, pp. 27-49, July, 1979.
18. Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 5, pp. 269-271, 1959.