REDUCTION IN RUN TIME USING TRAP ANALYSIS

[1]Prof. K.V.N.Sunitha **[2]Dr V. Vijay Kumar**

[1]Professor & Head, CSE Dept, G.Narayanamma Inst.of Tech. & Science, Shaikpet, Hyderabad, India.

[2]Dr V. Vijay Kumar Professor & Head, CSE Dept, R.G.M.I.T, Nandyal, Kurnool, India E-mail:

[1]sunitha_kvn@rediffmail.com, [2]vijayvakula@yahoo.com

ABSTRACT

"Reduction in Runtime using trap analysis " concentrates on finding the procedure for inserting traps at required places and optimizing runtime by moving the traps to low frequency regions. Using this, one can benefit in two ways: (i) being able to generate efficient code (ii) being able to produce more reliable programs by careful examination of errors in the programs. Traps are placed within a loop so as to avoid abrupt termination on bound violations. Since these traps are placed in high frequency regions, apply frequency reduction optimization technique to move this trap to low frequency regions. The results clearly indicates significant reduction in number of instructions, number of machine cycles and execution times for array bound checks applied on sorting techniques for inner loops as well as outer loops. In the present paper, the trap analysis is performed on all sorting techniques which have nested loops of depth two. However insertion of traps increases the run time, hence to overcome this disadvantage,an innovative method of optimization has been proposed in the present paper which move the checking code out of loops.

1. INTRODUCTION

Many compilers generate runtime checks to detect errors due to array bound violations dynamically which aid in debugging of programs. The overhead of these checks are quite high, resulting in inefficient code with high execution times. Earlier investigations indicate that execution times for programs may double if runtime checks are performed [1]. This is true for both optimized and unoptimized code because traditional optimizations are inefficient in reducing the overhead due to range checks. When the XL family of compilers was first released for the IBM RS/6000, they were coded in PL.8. Even though the compilers were producing code, range checking remained an integral part of the running code. The net cost of the checking was in the order of 1-2%. The aggressive optimization of range checks decreases the costs at an acceptable level[2]. What was found 20 years ago has become common place for the latest research area. The basis of our work is the work which is actually implemented for inner loops and accepted as novel for IBM XL family of compilers[3]. In the present study an innovative extension based on the above work is implemented for outer loops and it is been implemented on some of the sorting techniques and good optimization is obtained[6,7,8,9].

2. METHODOLOGY

The simplest and by far the most common case of optimizable range checking code is a range check embedded in a loop. For correctness following assumptions are made:

i. The loop should have an iteration variable 'i' with an initial value 'a' and

$0 \leq i \leq j$.

ii. 'i' and 'j' must be positive integers where $j \geq i$ and in every iteration 'i' can be increased by 'b' where $b \geq 1$, implies incremental factor is i+b.

By encoding the range check as a trap instruction, the trap has the property of not introducing program flow under normal conditions, much in the same way as the divide instruction does not introduce flow under normal conditions. That is to say, if the arguments of the trap do not indicate a range violation, the next sequential instruction is executed. If the divisor is non-zero, the division completes, and the next sequential instruction is executed. Otherwise, a divide check interrupt (and program termination) occurs. The trap instruction can then be subjected to code motion and common sub expression elimination much like any other instruction.

Consider the trap instruction as having its operand x compared to a region constant C, and the loop ending test which compares the induction variable 'i' to another region constant C′. Trap analysis s performed only if $i\text{-}x = r$, where r is some region constant. It is conceivable that i might be modified between the instruction which compares 'i' to C′ and then branches to an

instruction or exit point from the loop. The induction variable is not incremented between the test and the branch instructions.
Let ‘x’ represent the difference between the trap operand t, and the induction variable value at the loop entry point. If ‘i‘ is the induction variable at the Block entry point then t will have the value i+x for the next iteration (Fig. 2.1). To facilitate the remainder of the analysis, which is inductive in nature, the present study assumes an increasing induction variable. Then trap instructions (and any preceding instructions which are used to compute the operands for the trap instruction) are copied into the predecessor node. If a range violation is detected after the first iteration, the present study has taken care such that it will not occur in subsequent iterations.

```
Predecessor node
________________________
Block entry point
t=i+x
t ≤C        trap instruction
i≤ C′         loop termination test
________________________
        Loop exit point
```

Fig. 2.1 Bound Checking with Traps.

In order to remove the trap instruction from the loop altogether an adjustment must be made to C′ to guarantee that i and consequently t will not increase to a value which will cause a trap to terminate the program in subsequent iterations. The trap condition for staying in the loop is given above by the following condition

$t \leq C$

Since

$t = i + x$

then the condition for the trap not terminating the program on the next iteration is

$i \leq C - x$

While the original condition for staying in the loop had been $i \leq C'$,
replace the loop ending test with:

$$i \leq \min(C', C-x).$$

The principal advantage of above function is that the first argument of the minimum function guarantees that i will not attain larger values than it would have with the original loop ending condition and the second argument guarantees that in the subsequent iteration the trap will not terminate the program.

It has been shown that copying a trap instruction inside a loop cannot cause a program to terminate during execution of the loop body. The new loop end condition guarantees that trap cannot cause termination in subsequent iterations since loop end condition ensures no abrupt termination in subsequent iteration. The present method removes the trap placed inside the loop if it no longer serves any function. Since check is done at the end of the loop which forces the loop to be executed even on boundary violation, the trap instruction is placed in a predecessor node which checks the boundary violations. By this the loop is executed within the valid boundaries which ensures that program gets terminated when $C' \neq \min(C', C-x)$.

```
Predecessor node
t=i+x
t ≤C        trap instruction
________________________
                          Block
entry point
i≤ min (C′, C-x)      loop termination test
________________________
i > C′          loop exit trap
```

Fig. 2.2 Bound Checking optimization with Traps.

In contrast to the approach of [82] and [67], which implements a system that uses theorem proving techniques to verify the absence of array range violation in program, present approach can reduce the run-time overhead due to bound checks that can be eliminated at compile time. There are two limitations of program verification approach:(i) It often requires the programmer to supply assertions to aid the verification proofs and (ii) It is restricted to programs written in a structured manner(without go to). Hence this approach is not directly applicable to problem of automatic range check optimization of arbitrary programs.

3. EXPERIMENTAL RESULTS

This technique is tested on different sorting techniques and other programs using IBM Rational Quantify tool on Pentium IV 2.5 GHz Processor using three different options:

Case 1: No range checking code.
Case 2: Range checking code with Traps
Case 3: Optimized range checking code with Traps.

The above three cases are studied in depth and as an experimental model, they are applied on bubble sort, insertion sort, quick sort and selection sort techniques. The results are evaluated and compared not only in terms of execution times but also in terms of machine cycles and number of assembly instructions for inner and outer loops.

Table 2.1 describes optimization in terms of number of instructions, Table 2.2 describes optimization in terms of execution times and Table 2.3 describes optimization in terms of memory cycles. The results obtained are shown as bar graphs in Fig.2.3, Fig.2.4 and Fig.2.5. For better understanding, the same results are plotted as line graphs, shown in Fig.2.6, Fig.2.7 and Fig.2.8 respectively.

Inner Loop- Number of instructions			
Sorting Technique	Case 1	Case 2	Case 3
Selection Sort	463	515	505
Insertion Sort	443	489	479
Bubble Sort	497	521	512
Quick Sort	541	564	552

Fig. 2.3(a) Results of optimization in terms of number of instructions for inner loop.

Outer Loop- Number of instructions			
Sorting Technique	Case 1	Case 2	Case 3
Selection Sort	510	528	514
Insertion Sort	456	468	459
Bubble Sort	553	676	562
Quick Sort	599	621	602

Fig. 2.3(b) Results of optimization in terms of number of instructions for outer loop.

Inner Loop-Execution time in Micro seconds			
Sorting Technique	Case 1	Case 2	Case 3
Bubble sort	27.38	31.88	30.07

Table 2.1(a) optimization in terms of number of instructions- inner loop .

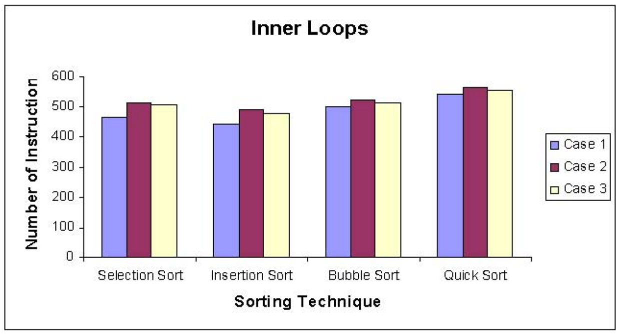


Table 2.1(b).Optimization in terms of number of instructions for Outer loop.

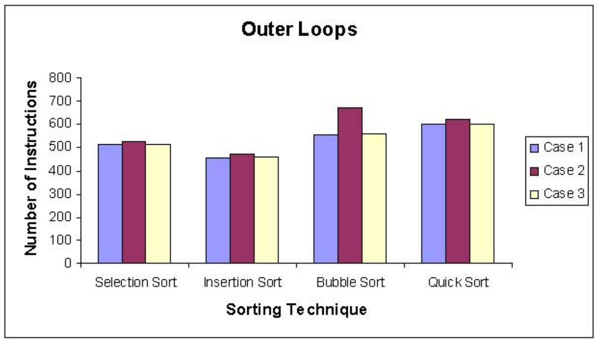


Table 2.2(a) Optimization in terms of Execution time for inner loop.

Insertion sort	12.72	13.33	12.29
Selection sort	14.16	15.75	14.63
Quick sort	13.04	13.65	13.05

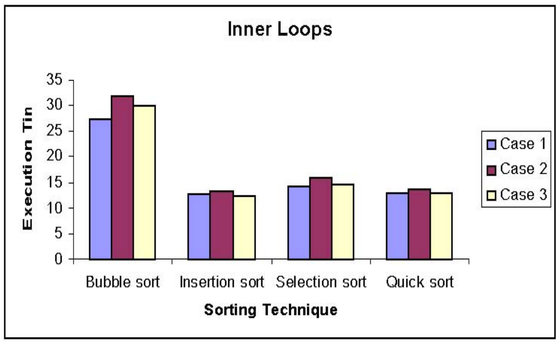


Fig. 2.4(a) Results of optimization in terms of Execution times for Inner loop.

Outer Loop - Execution Time in Micro seconds.			
	Case 1	Case 2	Case 3
Bubble sort	32.06	34.77	32.82
Insertion sort	13.27	14.38	13.41
Selection sort	15.16	16.88	15.68
Quick sort	14.83	15.63	14.93

Table 2.3(b) Optimization in terms of Execution time for outer loop.

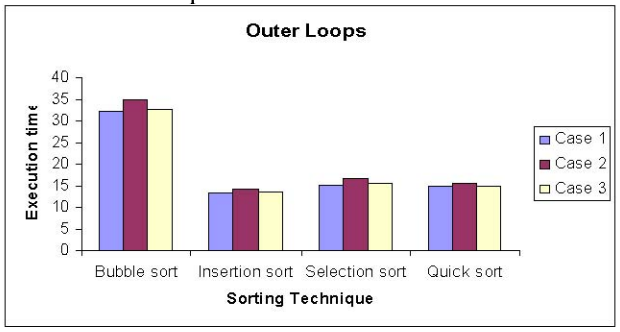


Fig. 2.4(b) Results of Optimization in terms of Execution times for Outer loops.

Inner Loop- Machine cycles.			
	Case 1	Case 2	Case 3
Bubble sort	80293	93503	93055
Insertion sort	30316	30436	30338
Selection sort	38600	40030	39734
Quick sort	34684	37033	36983

Table 2.3(a) Optimization in terms of Machine Cycles for inner loop.

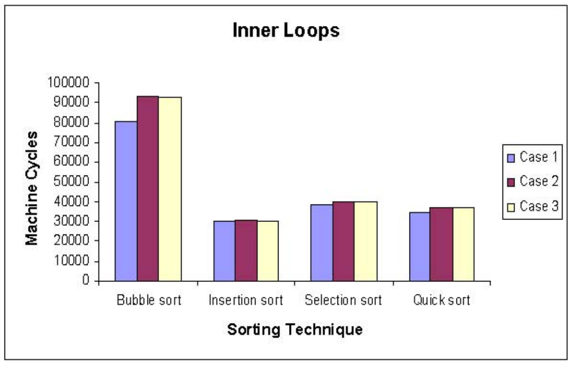


Fig. 2.5(a) Results of optimization in terms of Machine Cycles for inner loops.

Outer Loop- Machine cycles			
	Case 1	Case 2	Case 3
Bubble sort	80393	94350	94938
Insertion sort	30436	31836	30538
Selection sort	38797	41506	40210
Quick sort	35704	39033	38983

Fig. 2.5(b) Results of Optimization in terms of

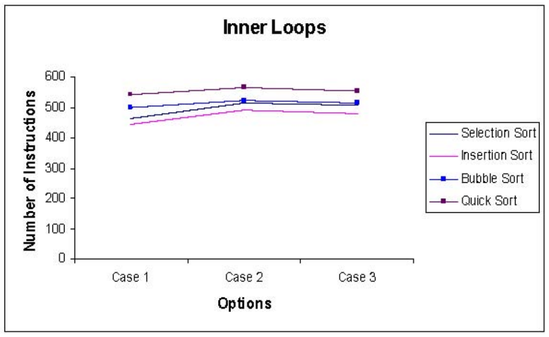


Fig. 2.6 Results of Optimization in terms of

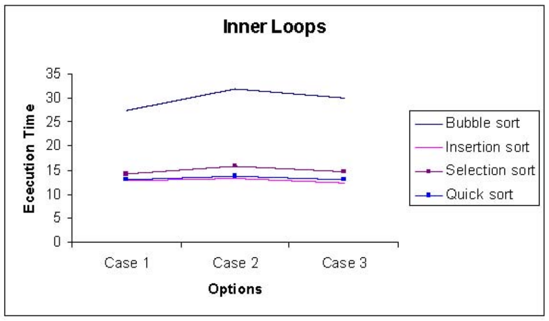
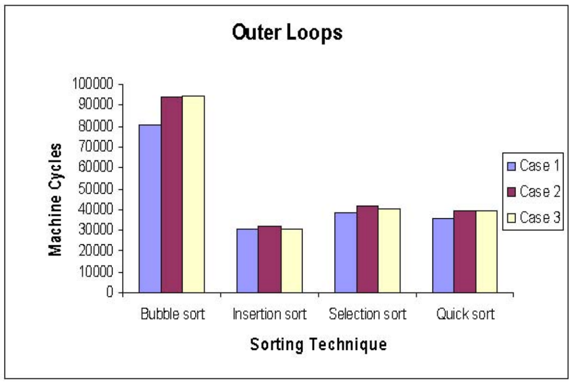
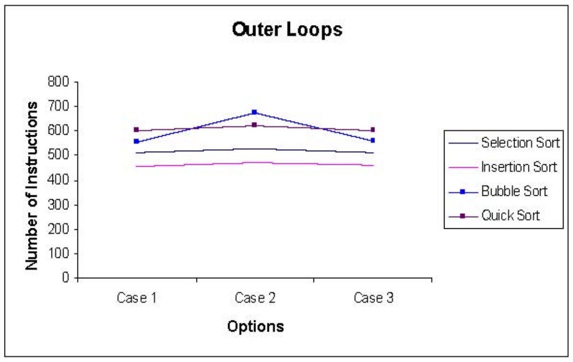


Table 2.3(b) Optimization in terms of Machine Cycles for outer loop.



Machine Cycles for Outer loops.



number of instructions**.**

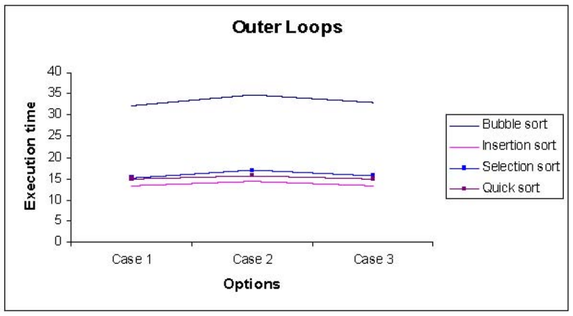
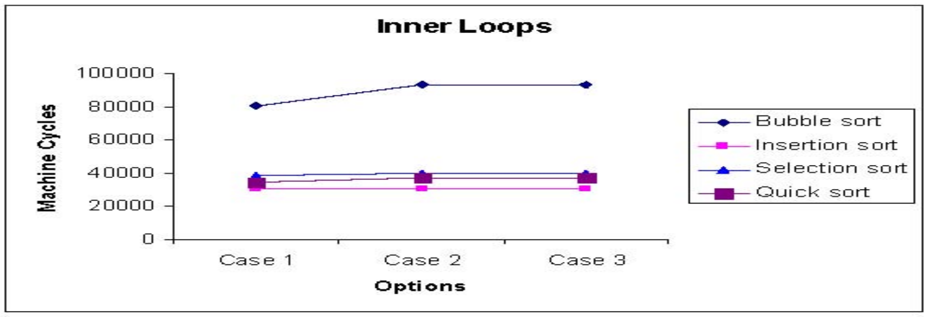


Fig. 2.7 Results of Optimization in terms of Execution times for outer loop.



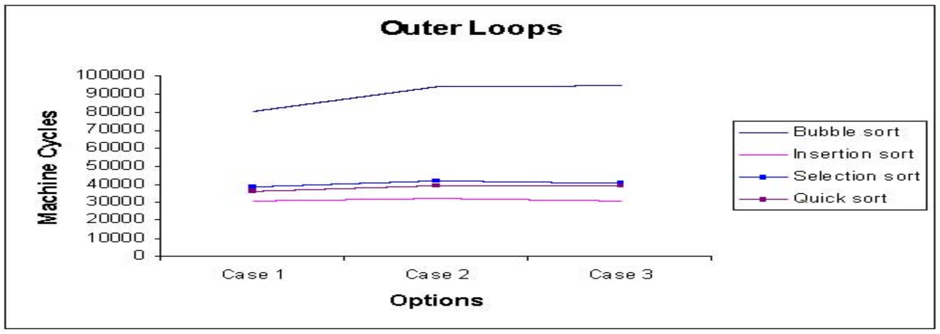


Fig. 2.8.Results of Optimization in terms of Machine Cycles for outer loop.

4. CONCLUSION

The above results clearly indicate significant reduction in number of instructions, number of machine cycles and execution times for array bound checks applied on sorting techniques for inner loops as well as outer loops when case 2 is compared with case 3.

In the present study, the trap analysis is performed on all sorting techniques which have nested loops of depth two. Had this analysis been carried out for the regions of high execution frequency, the results would have been capable of yielding impressive gains. Since the insertion of traps increases the run time, there is scope for applying other optimizations like redundancy elimination, common sub expression elimination.

REFERENCES:

[1]. Chow.F 1983. A portable machine-independent global optimizer Design and measurements.Tech. Rep. 83-254, Ph.D. thesis, Computer Systems Labs, Stanford Univ., Calif.

[2]. *XL Fortran for AIX Users' Guide*, IBM Corp. Form SC09-4946

[3]. Kevin O'Brien, Bill Hay, Joanne Minish, Hartmann Schaffer, Bob Schloss, Arvin Shepherd, Matthew Zaleski, Advanced Compiler Technology for the RISC System/6000 Architecture, IBM RISC System/6000 Technology, IBM

[4]. Suzuki. N. and Ishihata. K. 1977. Implementation of array bound checker. In Proceedings 4th ACM Symposium on Principles of Programming Languages. ACM, New York, 132-143.

[5]. S.M.German., Automating proofs of absence of common run time errors. Conference Record of the 5 th ACM Symposium on principles of programming languages, 105- 116, January, 1978.

[6]. Sunitha .K.V.N.and Vijay Kumar.V, Study of optimization on bubble sort using trap analysis. Published in the proceedings of International Conference on Systemics, Cybernetics and Informatics (ICSCI), Dec 2005, Hyderabad, January 2006.

[7]. Sunitha .K.V.N.and Vijay Kumar.V, Optimizing array range checks. Published in proceedings of International

conference on current trends of information Technology, MERG, S.R.K.R. Engg. College, Bhimavaram, Oct 2005.

[8]. Sunitha .K.V.N.and Vijay Kumar.V, Study of optimization on different sorting Techniques using trap analysis. Published in the proceedings of National Conference, Erode , 20-23rd December 2005.

[9]. Sunitha .K.V.N.and Vijay Kumar.V, Study of optimization on quick sort using trap analysis. Published in the proceedings of National Conference