The Performance of Scalar Replacement on the HP 715/50

Steve Carr
Qunyan Wu
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295

1 Introduction

It has been shown that the allocation of array values to registers can significantly improve the memory performance of scientific programs [3][5]. However, in many product compilers array references are left as references to main memory rather than promoted to references to registers because the data-flow analysis used by the compiler is not powerful enough to recognize most opportunities for reuse in subscripted variables. Arrays are often treated in a particularly naive fashion, if at all, making it impossible to determine when a specific element might be reused. This, however, need not be the case. In the code shown below,

```
    DO 1 I = 2, N
1     A(I) = A(I-1) + B(I)
```

the value accessed by `A(I-1)` is defined on the previous iteration of the `I`-loop. Using this knowledge, obtained via dependence analysis, the flow of values between the definition of `A(I)` and the use of `A(I-1)` can be expressed with temporaries as follows.

```
    T = A(I)
    DO 1 I = 2, N
      T = T + B(I)
1     A(I) = T
```

Since global register allocation will most likely put scalar quantities in registers, we have removed the load of `A(I-1)`. This transformation is called *scalar replacement* and can be applied to the reuse of array values across and within iterations of an innermost loop [2][3][5].

This report discusses an experiment with the effectiveness of scalar replacement on a number of scientific benchmarks run on the Hewlett-Packard 715/50. For a more detailed description of the scalar replacement algorithm, see Carr and Kennedy [5].

2 Experiment

A source-to-source translator, called Memoria, that replaces subscripted variables with scalars using the algorithm described in Carr and Kennedy has been developed in the ParaScope programming environment [1][4]. In this particular experiment, Memoria serves as a preprocessor to the HP product compiler, rewriting Fortran programs to improve register allocation of array values.

The target architecture for the experiment is the Hewlett-Packard 715/50. Each benchmark is compiled with version 9.16 of the HP Fortran compiler using optimization level 2 (-O). Experimentation with this compiler reveals that portions of the scalar replacement algorithm have already been implemented in the HP product compiler. Essentially any invariant reuse in the innermost loop that does not require sophisticated dependence analysis to detect and any reuse within an iteration of the innermost loop (loop-independent reuse) is detected by the HP compiler. Therefore, this study attempts to reveal what can be gained by including loop-carried reuse (not invariant),

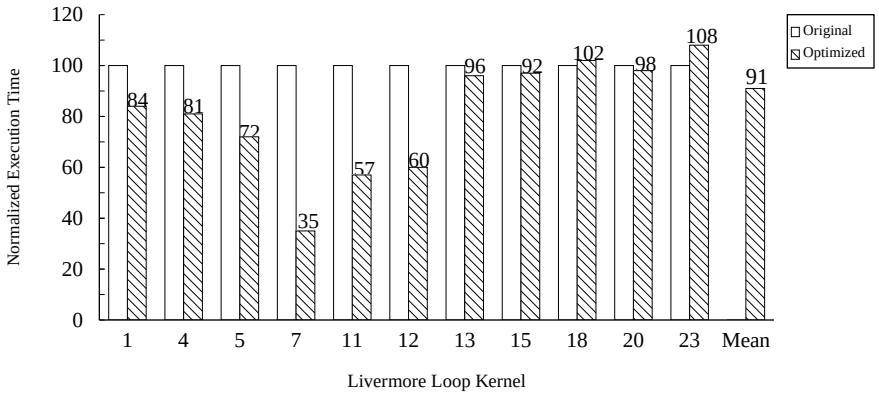

FIGURE 1. Performance of Livermore Loops

invariant reuse that requires sophisticated dependence analysis and reuse that exists only on some paths through the loop (partially available values).

The test suite for this experiment consists of the Livermore Loops, a number of linear algebra kernels and programs from the SPEC, Perfect and NAS benchmark suites. The rest of this reports details our examination of scalar replacement on this suite of programs and kernels.

2.1 Livermore Loops

The first benchmark set that we examine is the 24 Livermore Loops. The performance of the loops is detailed in Figure 1. Performance is shown in normalized execution time where the base time of 100 is not labeled. Any loop not detailed in Figure 1 does not improve or degrade in performance due to scalar replacement.

All loops in Figure 1 that show improvement have loop-carried dependences that are amenable to scalar replacement. The loops with larger run-time improvements have a larger reduction in the ratio of memory reference to floating-point operations than do the loops with smaller improvements. Loop 7 improves by such a large degree because 5 of 10 array references are removed from the loop, leaving the loop no longer bound by memory accesses.

Two of the Livermore Loops show a degradation in performance after scalar replacement. In each case, the software pipelining algorithm used by the HP compiler is inhibited by scalar replacement. Thus, the original loop is software pipelined, but the scalar-replaced loop is not software pipelined. Software pipelining fails on loop 18 after scalar replacement due to register pressure. On loop 23, software pipelining fails to find a schedule.

2.2 Linear Algebra and Miscellaneous Kernels

The next benchmark set that we examine is a collection of kernels commonly used in linear algebra and other scientific applications. Table 1 gives a description of the kernels used in this study.

Linear Algebra Kernel	Description
VM	Vector-Matrix Multiply
MMk	Matrix Multiply reduction order
MMi	Matrix Multiply cache order
LU	LU Decomposition
LUP	LU Decomposition w/pivoting

TABLE 1. Linear Algebra Kernel Descriptions

Linear Algebra Kernel	Description
BLU	Block LU
BLUP	Block LUP
Chol	Cholesky Decomposition
Afold	Adjoint Convolution
Fold	Convolution
Seval	Spline Evaluation
Sor	Successive Over Relaxation

TABLE 1. Linear Algebra Kernel Descriptions

A common property of the linear algebra kernels is that they compute reductions that contain array references that are invariant with respect to the innermost loop. As mentioned previously, these types of references are scalar replaced by the current version of the HP optimizer unless sophisticated dependence analysis is needed to detect the invariant reuse. Figure 2 details the performance of the kernels in normalized execution time.

Each of the kernels that show no improvement contains only invariant references that are amenable to scalar replacement and those references are caught by the HP compiler. LU decomposition (with and without pivoting), block LU decomposition (with and without pivoting) and Cholesky decomposition all require some form of triangular loop dependence testing to detect the invariant array references. Although, in general, this can be a reasonably complex dependence test for the back end of a compiler, we believe many common cases can be detected with simple pattern matching.

Of the kernels that show improvement, the performance of the block LU decomposition algorithms improves more than other kernels because the block algorithms contain an innermost-loop reduction that allows scalar replacement to remove both a load and a store. The point versions of LU decomposition (with and without pivoting) only have an invariant load removed from their main loop. Cholesky decomposition contains an inner loop reduction just at the block LU algorithms, but the kernel contains other loops that do not contain opportunities for scalar replacement. Thus, Cholesky's improvement is less. Seval and Sor are two kernels that contain variant loop-carried reuse. Sor improves more because it has a higher percentage of references removed.

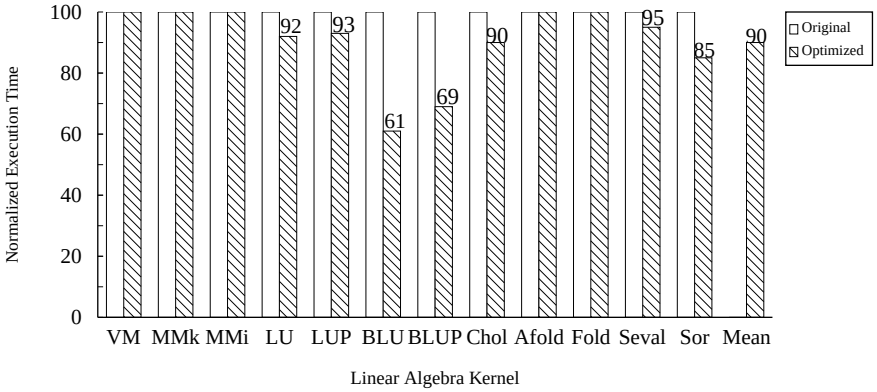

FIGURE 2. Performance of Linear Algebra Kernels

2.3 Benchmark Suites

This section details the performance of scalar replacement on a number of benchmark suites. The experiment reveals that the effects of scalar replacement are disappointing on whole programs. The lack of improvement occurs for three major reasons:

1. The loops on which scalar replacement is performed may not contribute a significant amount of execution time to the whole program.
2. A benchmark may not have many loop-carried dependences that Memoria's algorithm handles and HP's compiler does not handle.
3. The scalar replacement performed by Memoria may disable the software pipelining algorithm in the HP compiler on some loops and not others, causing a net improvement at or below zero. Software pipelining can be inhibited in the HP product compiler by high register pressure, long loop bodies or failure to find a schedule. Each of these effects occurs in our test suite.

In the following section, we present the performance of the programs in SPEC, Perfect, NAS and a group of miscellaneous benchmarks. We also discuss the reasons for performance gains, degradations or unvarying related to the previous observations. The third of the above observations turns out to be the most significant one in this experiment.

2.3.1 SPEC Benchmarks

The first benchmark suite that we examine is SPEC. Our experiment uses programs from both the SPEC89 and SPEC92 Fortran suites. Table 2 gives the list of Fortran programs and their corresponding key used in Figure 3.

Key	Benchmark Name
d7	dnasa7
dd	doduc
fp	fpppp
h2d	hydro2d
m300	matrix300
md2	mdljdp2
ms2	mdljsp2
ora	ora
s2r	su2cor
tv	tomcatv
w5	wave5

TABLE 2. Spec Benchmark Name Key

Dnasa7, fpppp, matrix300, mdljdp2, mdljsp2, ora and su2cor do not have a sufficient number of loop carried dependences to result in run-time improvements after scalar replacement. Of those loop carried dependence that do exist in these programs, 90-100% are loop invariant. Swm256 has loop-carried dependences, but there are also loop independent dependences between most of the references and only a small percentage of references are removed by going across the loop iterations. In tomcatv, although there are loop-carried dependences, the register pressure is too high to allocate registers across loop iterations. Finally, wave5 shows improvement only in those routines that do not contribute significantly to the running time of the program.

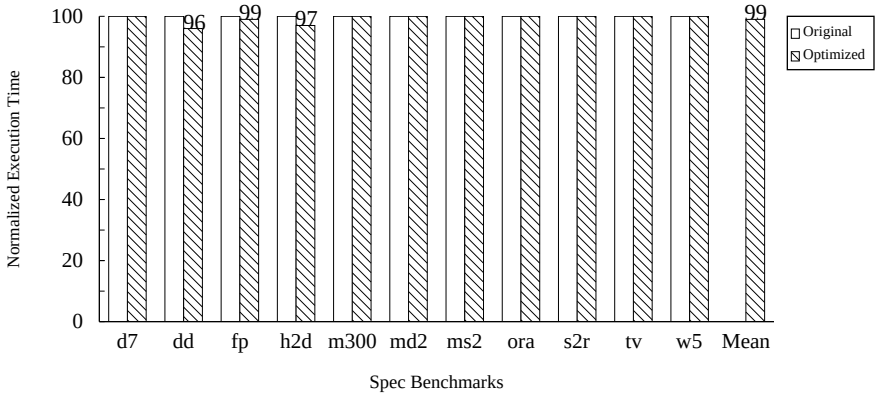

FIGURE 3. Performance of Spec Benchmarks

2.3.2 Perfect Benchmarks

The next benchmark suite that we examine is the Perfect Club. Unfortunately, we cannot get all of the programs in the suite to run on the HP 715/50. We do not have results for bdna, mdg, mg3d, ocean and spec77. Table 3 gives the names of the benchmarks from the Perfect Club that we are able to test.

Key	Benchmark Name
adm	adm
a2d	arc2d
dyf	dyfesm
f52	flo52
qcd	qcd
track	track
trfd	trfd

TABLE 3. Perfect Club Name Key

Dyfesm, qcd, track and trfd have only one combined reference between them that our scalar replacement algorithm is able to handle and the HP optimizer is not able to handle (this reference appears in dyfesm). Therefore, no improvement is possible with Memoria. Of those programs that we cannot successfully run on the 715/50, only

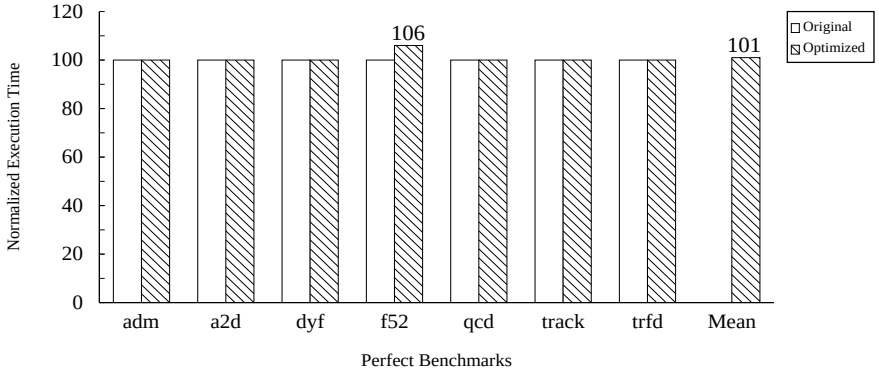

FIGURE 4. Performance of the Perfect Club

bdna and spec77 have any opportunities for Memoria. Due to the unpredictability of the interaction between scalar replacement and software pipelining (since we do not account for software pipelining in Memoria), the results of bdna and spec77 cannot be conjectured.

For the benchmark adm, many loops improve in execution time while others degrade. One of most serious degradation (6%) occurs on the third most computationally expensive loop in the program and is caused by an inhibition of software pipelining. In adm, more loops improve in performance after scalar replacement than degrade in performance. However, performance degradation occurs in the more computationally expensive loops and overall, the effect is to nullify any improvements gained by scalar replacement. Arc2d has only 8 additional references that Memoria can replace and, therefore, no improvement results. Finally, we cannot determine exactly why Flo52 degrades in performance. The degradation occurs in one loop nest where we suspect that our estimation of register pressure is too low and scalar replacement causes additional spill code. Our algorithm only looks at register pressure on a per loop basis rather than on a global basis. Thus, any values live across a loop body are not taken into account.

2.3.3 NAS Benchmarks

The next benchmark suite that we examine is the NAS benchmarks. The programs in this suite are listed in Table 4.

Key	Benchmark Name
abt	appbt
alu	applu
asp	appsp
buk	buk
emb	embar

TABLE 4. NAS Benchmark Key

Three programs from this suite (cgm, fftpde, mgrid) are not listed because we could not get them to run on the 715/50. None of these three programs have scalar replacement opportunities for Memoria, so no improvement can be expected.

Of the remaining programs, appbt, applu and appsp have opportunities for Memoria. Buk and embar have no opportunities for Memoria. Figure 5 gives the performance of these programs. Of the programs that have opportunities for scalar replacement, only appbt shows any improvement. Each of appbt, applu and appsp suffers from the inhibition of software pipelining to counteract any gains made.

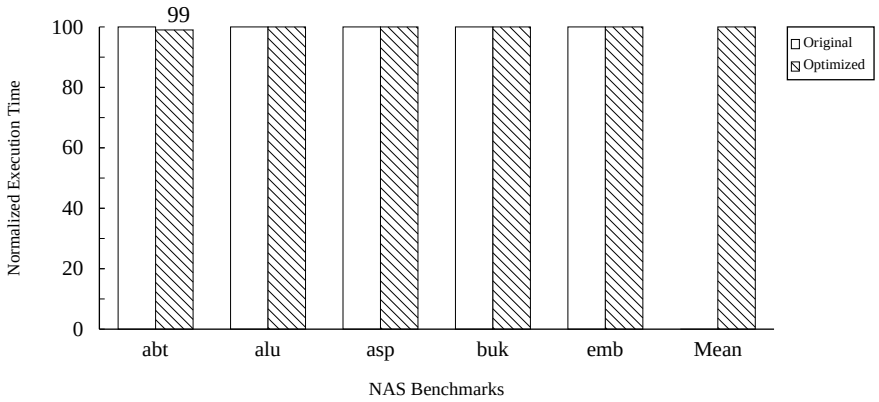

FIGURE 5. Performance of NAS Benchmarks

2.3.4 Miscellaneous Benchmarks

To conclude our study, we look at the performance of three benchmarks that do not fall into any of the standard benchmark suites. Table 5 gives the names and descriptions of those benchmarks.

Benchmark	Description
Simple	2D Hydrodynamics
Erlebacher	3D ADI Integration
Linpackd	Linear Algebra Package

TABLE 5. Miscellaneous Benchmark Descriptions

Figure 6 details the performance of the three benchmarks. Both Erlebacher and Linpackd show no improvement over the HP compiler because all or nearly all of the loop-carried dependences in each benchmark are invariant with respect to the innermost loop. Simple shows a significant performance degradation. Simple has one computational loop that encompasses 50% of the execution time of the entire program. This loop consists of a single outer loop containing four inner loops. When scalar replacement is performed on the inner loops, software pipelining is inhibited in three of those four loops. This causes a significant performance degradation that is actually more than that seen over the entire benchmark. In other loops in the program, scalar replacement improves performance enough to counteract some of the effects in the main computational loop.

2.4 Scalar Replacement Statistics

As part of this study, we collected a number of statistics on the loops in our benchmark suites. There were 2707 loops in our suite of which 2586 were scalar replaced. The loops that were not replaced did not contain array references. Of the 2586 replaced loops 1731 did not have any opportunities for scalar replacement. It is highly probable that forward expression propagation and auxiliary induction variable recognition would increase the number of loops that have opportunities for scalar replacement. In the loops that had opportunities for scalar replacement 4725 array references were removed. Of the replaced references, 70% had their value generated on the same loop iteration, 13% of the references were invariant with respect to the inner loop and the remaining 17% had their value generated on a previous loop iteration. Less than 1% of the reference replaced required partial redundancy elimination to ensure a value was generated on all paths through the loop [5].

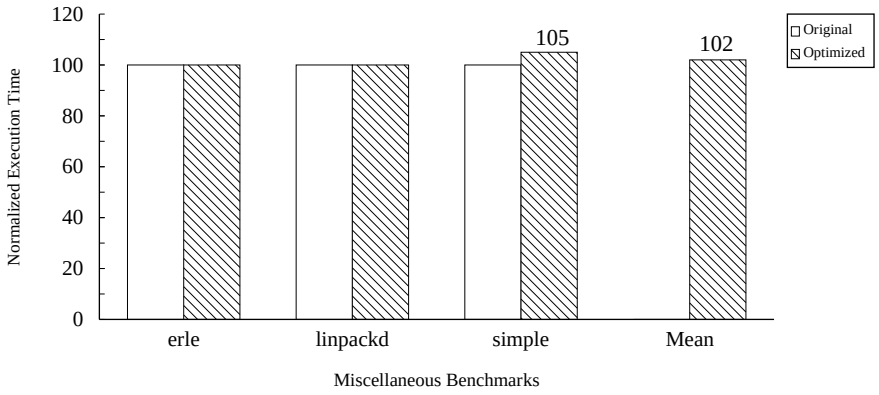

FIGURE 6. Performance of Miscellaneous Benchmarks

3 Conclusions

In this paper, we have reported on an experiment with scalar replacement on the HP 715/50. Our results have been mixed. While there is room for improvement over the current HP optimizer, we believe that intimate knowledge of the back end of the compiler is necessary to ensure that scheduling and register allocation are not adversely affected by scalar replacement. Since we are unable to determine adequately how beneficial more aggressive scalar replacement techniques might be to the HP compiler, we suggest the following study to provide that information.

- Modify Memoria to allow different levels of aggressiveness with scalar replacement. This will allow us to look at the effect of replacing different reference subsets.
- Modify Memoria to perform different sets of dependence tests. This will allow us to determine the effects of more sophisticated dependence analysis.
- Gather static statistics from the scalar replacement algorithm on the number of references replaced based on the aggressiveness of scalar replacement and dependence analysis.
- Gather dynamic statistics on the number of loads and stores removed based on the aggressiveness of scalar replacement and dependence analysis. This will reveal the potential for scalar replacement to improve performance.

We believe that this study will help HP determine whether a more aggressive scalar replacement algorithm in combination with better knowledge of the instruction scheduler will be beneficial to its product compiler.

References

1. D. Callahan, K. Cooper, R. Hood, K. Kennedy and L. Torczon. ParaScope: a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, June 1987.

2. D. Callahan, J. Cocke, K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5:334-358, 1988.

3. D. Callahan, S. Carr and K. Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices* 25(6):53-65, 1990. *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation.*

4. S. Carr. *Memory-Hierarchy Management.* Ph.D. Thesis, Rice University, Department of Computer Science, 1993.

5. S. Carr and K. Kennedy. Scalar Replacement in the Presence of Conditional Control Flow. *Software - Practice & Experience*, volume 24, number 1, pp. 55-71, January 1994.