# THE EFFECT OF COMPILER OPTIMIZATIONS ON AVAILABLE PARALLELISM IN SCALAR PROGRAMS

*Scott A. Mahlke* *Nancy J. Warter* *William Y. Chen* *Pohua P. Chang* *Wen-mei W. Hwu*

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

## Abstract

In this paper we analyze the effect of compiler optimizations on fine grain parallelism in scalar programs. We characterize three levels of optimization: classical, superscalar, and multiprocessor. We show that classical optimizations not only improve a program's efficiency but also its parallelism. Superscalar optimizations further improve the parallelism for moderately parallel machines. For highly parallel machines, however, they actually constrain available parallelism. The multiprocessor optimizations we consider are memory renaming and data migration.

## Introduction

Compiler optimizations are designed to reduce a program's execution time. Traditionally, these optimizations are customized for a given machine model. Classical optimizations are designed to improve the program's efficiency for a machine model which has one thread of execution and can issue one instruction per cycle. Superscalar optimizations are designed for a machine model with a single thread of execution and a limited instruction issue rate. Multiprocessors are built using either uniprocessors or superscalar processors and thus there is more than one machine model to optimize for. Therefore, it is important to understand the interactions of these optimizations and their effect on available parallelism and speedup.

There has been significant research done to analyze the available parallelism in numeric programs [1] [3]. Previous researchers have shown that for numeric programs the most parallelism can be found at either the instruction-level or the loop-level [1]. However for scalar programs, Larus suggests that there is not much loop-level parallelism available because the loops tend to be small and have few iterations [5]. Therefore, it may be better to exploit fine grain parallelism for scalar applications.

In this paper we analyze the effect of classical and superscalar optimizations on fine grain parallelism and speedup for scalar programs. Using code generated by the IMPACT-I C compiler, we study the effectiveness of these optimizations for a range of machines that can exploit increasing levels of fine grain parallelism. Furthermore, in order to study the effect of classical and superscalar optimizations on highly parallel code we simulate two powerful multiprocessor optimizations that have been shown to uncover large amounts of parallelism within an application program: memory renaming and data migration to high speed memory [2] [4].

## Compiler Optimizations

Compiler optimizations remove artificial constraints imposed by the programmer and the programming language, in order to increase the program's efficiency and expose its inherent parallelism. We have classified these optimizations into three levels: classical, superscalar, and multiprocessor.

### Classical Optimizations

Classical optimizations are made up of two components, local and global optimizations. Local optimizations are applied to instructions within a basic block, and use no knowledge of the program as a whole (e.g., data flow analysis) to make optimization decisions. The local optimizations considered in this paper are constant propagation, copy propagation, common subexpression elimination, redundant load/store elimination, constant folding, strength reduction, operation folding, constant combining, and code reordering. On the other hand, global optimizations are applied among operations within the same function. The global optimizations considered in this paper are constant propagation, copy propagation, common subexpression elimination, redundant load/store elimination, dead code removal, loop invariant code removal, loop induction variable strength reduction, loop induction variable elimination, and loop global variable migration. The goal of classical optimizations is to reduce the execution time of a program by eliminating redundant instructions and replacing a set of instructions with a more efficient set. The effect of these optimizations on the available parallelism is not clear.

Reducing the number of instructions will typically reduce the available parallelism within the program. Consider the following code before and after common subexpression elimination is applied:

```
Before              After
    $t_1 = 2 * i$          $t_1 = 2 * i$
    $x = a[t_1]$           $x = a[t_1]$
    $t_2 = 2 * i$          $b[t_1] = x$
    $b[t_2] = x$
```

Before optimization, since the first and third instructions can execute concurrently, the parallelism is 4/3 (for these examples we assume unit time delay unless otherwise specified). However, after optimization the parallelism is 3/3. Therefore, code removal reduces the available parallelism but the execution time remains the same in parallel processing systems and is reduced in a uniprocessor system.

On the other hand, loop induction variable strength reduction will typically increase the available parallelism by reducing the length of the critical path. For example, consider the following code before and after induction strength reduction:

```
Before                       After
                                  $t_1 = 3 * j$
$L1 : t_1 = 3 * j$           $L1 : t_2 = t_2 + a[t_1]$
      $t_2 = t_2 + a[t_1]$          $j = j - 1$
      $j = j - 1$                   $t_1 = t_1 - 3$
      $if\ j > c\ goto\ L1$         $if\ j > c\ goto\ L1$
```

Assuming that the multiply instruction takes six time units and the other instructions require one time unit, the parallelism within the loop before optimization is 4/8. After optimization, the parallelism within the loop becomes 4/2.

In addition, there are some optimizations such as loop invariant code removal which can either decrease or increase the parallelism depending on whether or not the optimization reduces the critical path.

## Superscalar Optimizations

Superscalar optimizations combine and enlarge basic blocks to expose more parallelism. The following superscalar optimizations are considered in this paper: superblock formation, loop unrolling, loop peeling, branch target expansion, induction variable expansion, memory disambiguation, and register renaming. A superblock is the basic scope for optimizations. Superblock formation consists of first combining basic blocks which tend to execute in sequence into a trace, and then performing code duplication to eliminate all side entrances from the trace.

Loop unrolling replicates the body of a superblock loop several times. Loop peeling fully unrolls loops with small numbers of iterations. Branch target expansion copies the target superblock of a frequently taken branch into its fall-through path. Induction variable expansion removes the dependencies between induction variables in unrolled copies of a loop body. Memory disambiguation and register renaming are used to remove artificial dependencies between instructions.

Superblock formation and superblock optimizations add additional bookkeeping instructions to the less frequently executed portions of a program. For superscalar architectures with a limited scheduling scope, these additional instructions do not have much impact on the resultant parallelism. However, for larger parallel machines, these added instructions may increase the critical path which effectively decreases the available parallelism.

To understand why the critical path may increase for multiprocessors consider the *doacross* loop shown in Figure 1a. The superscalar execution and multiprocessor execution traces before and after the superscalar transformations are shown in Figures 1b and 1c. In a superscalar machine, the restricted issue will limit the amount of overlap between a parallel section and the subsequent sequential section. Since this overlap is anticipated to be small, we do not include it in this example (e.g., $P_i$ does not overlap with $S_{i+1}$ in the superscalar execution). In a highly parallel machine, a parallel section $P_i$ cannot overlap the proceeding sequential section $S_i$ and the sequential sections cannot overlap. However, two parallel sections $P_i$ and $P_j$ may overlap.

After the superscalar optimization, the size of the sequential code, $S_i$, is increased to $S_i'$ and the size of the parallel code, $P_i$, is reduced to $P_i'$. The critical path of the superscalar code is

$$critical\ path_{superscalar} = \sum_{i=1}^{N} (S_i + P_i) \qquad (1)$$

If the superscalar optimizations reduce the sum of the parallel sections more than they increase the sum of the sequential sections, the critical path will be reduced. Formally stated, the optimizations are effective for superscalar machines if

$$\sum_{i=1}^{N} (S_i' - S_i) < \sum_{i=1}^{N} (P_i - P_i') \qquad (2)$$

The critical path of the multiprocessor code is

$$critical\ path_{multiprocessor} = \sum_{i=1}^{N} S_i + P_N \qquad (3)$$

The critical path will be longer if the increase in execution time of the sequential sections is greater than the decrease in the parallel section, $P_N$. Formally stated, the superscalar optimizations are effective for multiprocessors if

$$\sum_{i=1}^{N} (S_i' - S_i) < P_N - P_N' \qquad (4)$$

Comparing equations 2 and 4, it is clear that superscalar optimizations are much more likely to increase the critical path and thus reduce the parallelism on a highly parallel machine than on a superscalar machine.

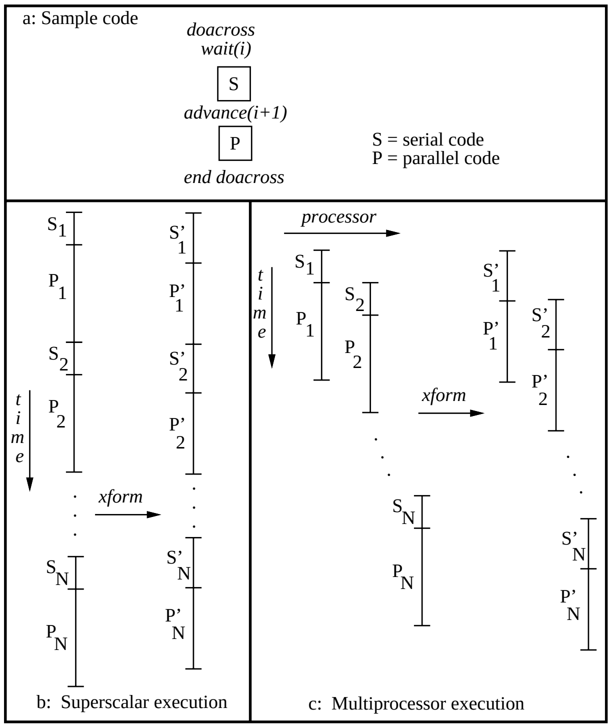


Figure 1: Superscalar versus multiprocessor execution.

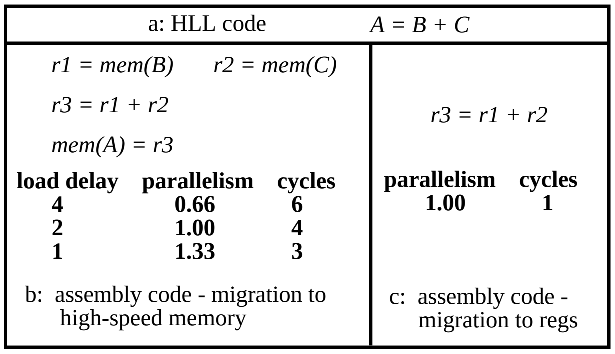
## Multiprocessor Optimizations

Memory renaming and data migration to high speed memory are powerful compiler optimizations that uncover the inherent parallelism within an application program. Memory renaming refers to renaming all memory variables such that they only hold one value. Since a memory variable is never written more than once, all memory output and anti-dependencies are removed. Data migration refers to loading frequently used memory variables into high speed memory such as registers.

It is obvious that memory renaming will improve the parallelism because it removes data dependencies. However, the effect of data migration on parallelism depends on the level of data migration. For example, consider the high level language (HLL) code statement in Figure 2a. The assembly language of the code statement is shown in Figure 2b for migration to high-speed memory. For this code segment, the parallelism and execution time is shown for load delays of 1, 2, and 4. It can be seen that reducing the load delay both increases the parallelism and decreases the execution time. Now, consider the assembly code in Figure 2c when the variables can be migrated to registers. The parallelism decreases with respect to load delay 1 but the execution time still improves. Therefore, parallelism and efficiency are not always mutually attainable goals.

## Method

We have developed IMPACT-I, a retargetable C compiler with classical and superscalar optimization capability. To


a: HLL code $A = B + C$

*r1 = mem(B)* *r2 = mem(C)*
*r3 = r1 + r2*
*mem(A) = r3*

| load delay | parallelism | cycles |
|---|---|---|
| 4 | 0.66 | 6 |
| 2 | 1.00 | 4 |
| 1 | 1.33 | 3 |

b: assembly code - migration to high-speed memory

*r3 = r1 + r2*

| parallelism | cycles |
|---|---|
| 1.00 | 1 |

c: assembly code - migration to regs

Figure 2: Effect of data migration on parallelism and efficiency.

calibrate the quality of the classical optimizations, we compare the execution times of the code generated by our compiler and the MIPS C compiler on a DEC 3100 workstation. Our benchmark set consists of five programs, *eqntott*, *espresso*, and *xlisp*, are from the SPEC benchmark set, and the others, *lex* and *yacc*, are commonly used scalar programs. Table 1 shows the speedup we obtain over the MIPS C compiler using its highest degree of optimization. For this study it is important to start with highly optimized

| *program* | *MIPS-O4* | *IMPACT* |
|---|---|---|
| eqntott | 1.0 | 1.04 |
| espresso | 1.0 | 1.02 |
| lex | 1.0 | 1.01 |
| xlisp | 1.0 | 1.13 |
| yacc | 1.0 | 1.00 |

Table 1: Speedup comparison.

code because naive code may contain redundant operations which show deceptive amounts of fine grain parallelism.

We simulate memory renaming by only preserving the memory data flow dependencies. Data migration is simulated by reducing the load operation latency. For load delay zero, the memory loads and stores are not counted in the instruction count. However, we were not able to remove the address calculations.

The machine is assumed to have infinite computational resources, perfect branch prediction, infinite branch lookahead, an infinite register file, and out-of-order execution. The basic processing element for all machines has deterministic operation latencies. All integer operations have a 1 cycle latency with the exception of multiply (6 cycles) and divide (12 cycles). The memory load latency is 2 cycles. Finally, all floating point operations have a 6 cycle latency with the exception of divide (12 cycles).

The level of parallelism the machine can exploit is varied by changing the window size. The smaller window sizes are used to model the behavior of superscalar machines, while the larger window sizes are used to model highly parallel

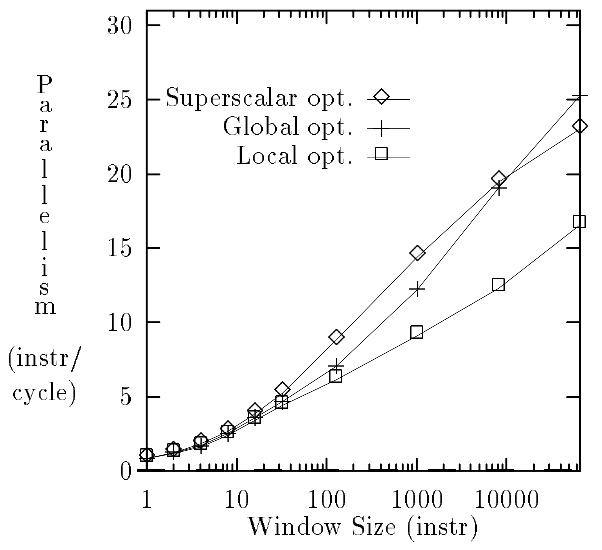


Figure 3: Effect of classical and superscalar optimizations.

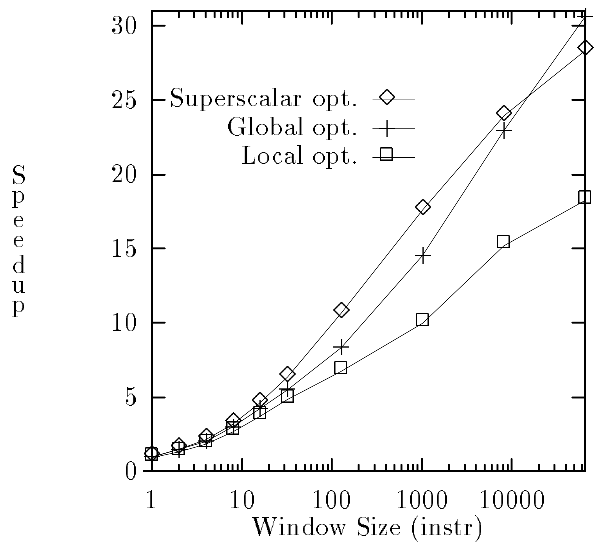


Figure 4: Effect of classical and superscalar optimizations.

multiprocessors. However, it is difficult to specify exact machine boundaries (e.g., a superscalar machine may have a low issue rate, but the compile-time scheduling window is actually much larger).

## Experiments

For the results presented in this section, each data point represents the harmonic mean of the parallelism or speedup of the five benchmarks in Table 1. The base machine for the speedup calculations has a window size of one with global optimization. For all of the results, each optimization level is applied to code optimized by the previous level (e.g., global includes local and superscalar includes global).

The effects of classical and superscalar optimizations on parallelism and speedup are shown in Figures 3 and 4. Figure 3 shows the importance of optimizing for a target machine. For small window sizes the optimizations have little effect on the available parallelism. For the intermediate range of window sizes the global optimizations reveal more parallelism than local, and superscalar optimizations have the highest available parallelism. However, for very large window sizes, the overhead of the superscalar optimizations actually decreases the available parallelism and speedup. This supports the analysis of superscalar optimizations for highly parallel machines discussed earlier in this paper.

Figures 5 and 6 show the interaction of classical, superscalar and multiprocessor optimizations with respect to parallelism and speedup for an infinite window. Note that the data migration results include memory renaming. For highly parallelized code, we see that global optimizations continue to perform significantly better than local and that

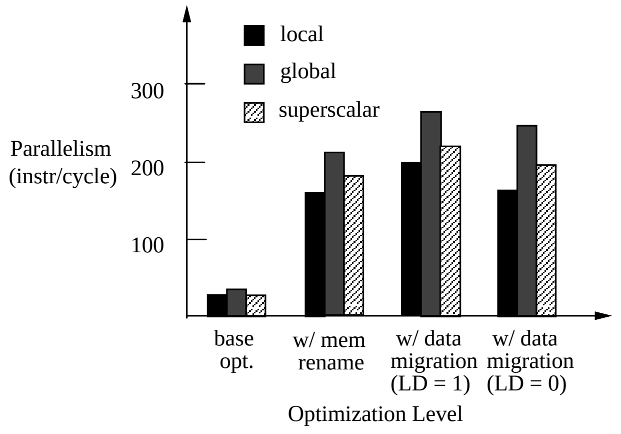


Figure 5: The interaction of classical, superscalar and multiprocessor optimizations for an $\infty$ window.

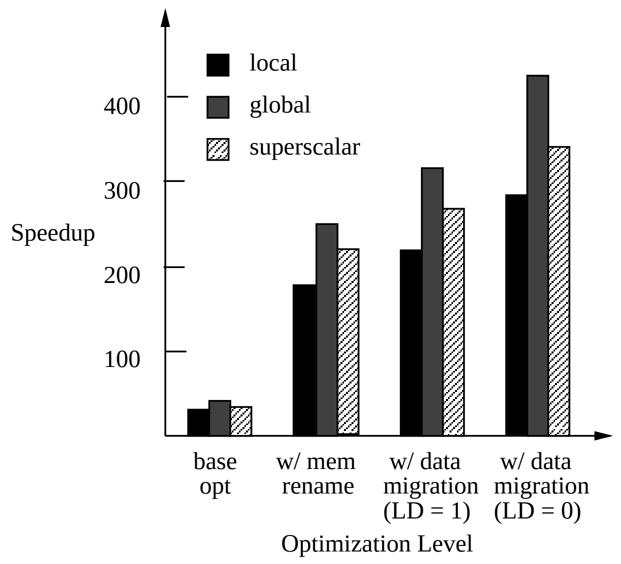


Figure 6: The interaction of classical, superscalar and multiprocessor optimizations for an $\infty$ window.

superscalar optimizations continue to have a negative effect on parallelism and speedup. Figure 5 shows that there is significant fine grain parallelism available in scalar programs, especially after memory renaming and data migration. Memory renaming is the most effective optimization for revealing parallelism. Data migration also improves the parallelism but it is most effective at improving the efficiency of programs (Figure 6). Note that as discussed earlier, data migration reduces the parallelism when the load delay is decreased from 1 to 0.

## Conclusions

In this paper we have shown that optimizations designed for a specific processor are not necessarily valid when this processor is embedded within a multiprocessor machine. For instance, for multiprocessors which can exploit a moderate amount of fine grain parallelism, superscalar optimizations expose significant parallelism. However, for multiprocessors which can exploit large amounts of fine grain parallelism, superscalar optimizations may actually degrade the parallelism. Furthermore, we have shown that for highly parallel machines, global optimizations which are designed to improve a program's efficiency for uniprocessors also reveal more parallelism than might be expected.

## Acknowledgements

The authors would like to thank Sadun Anik, Paul Chen, and all members of the IMPACT research group for their support, comments, and suggestions. This research has been supported by NSF under Grant MIP-8809478, NCR, AMD, NASA under contract NASA NAG 1-613 in cooperation with ICLASS, and ONR under contract N00014-88-K-0656.

## References

[1] D. -K. Chen, H. -M. Su, and P. -C. Yew, "The Impact of Synchronization and Granularity on Parallel Systems", *Proceedings of the 17th Annual International Symposium on Computer Architecture*, June 1990, pp. 239-248.

[2] R. Cytron and J. Ferrante, "What's in a Name? The Value of Renaming for Parallelism Detection and Storage Allocation", *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987, pp. 19-27.

[3] D. J. Kuck, Y. Muraoka, and S. -C. Chen, "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup", *IEEE Transactions on Computers*, vol. C-21, no. 12, December 1972, pp. 1293-1310.

[4] M. Kumar, "Effect of Storage Allocation/Reclamation Methods on Parallelism and Storage Requirements", *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987, pp. 197-205.

[5] J. R. Larus, "Parallelism in Numeric and Symbolic Programs", *Proceedings of the 1990 Irvine Workshop*, July 1990.