Performance Characterization of Optimizing Compilers

USC-CS-92-525

Rafael H. Saavedra†
Alan J. Smith‡

† Department of Computer Science
University of Southern California
Los Angeles, CA 90089-0781
saavedra@usc.edu

‡ Computer Science Division
University of California
Berkeley, CA 94720

August 15, 1992
Performance Characterization of Optimizing Compilers

Rafael H. Saavedra
Alan Jay Smith

ABSTRACT

Optimizing compilers have become an essential component in achieving high levels of performance. Various simple and sophisticated optimizations are implemented at different stages of compilation to yield significant improvements, but little work has been done in characterizing the effectiveness of optimizers, or in understanding where most of this improvement comes from.

In this paper we study the performance impact of optimization in the context of our methodology for CPU performance characterization based on the abstract machine model. The abstract machine model considers all machines to be different implementations of the same high level language machine; in previous research, we have used this model as a basis to analyze machine and benchmark performance. In this paper, we: 1) show that our model can be extended to characterize the performance improvement provided by optimizers and to predict the run time of optimized programs; 2) measure the effectiveness of several optimizing compilers in implementing different optimization techniques; and 3) analyze the optimization opportunities present in the Fortran SPEC benchmarks and other benchmarks.

1. Introduction

Recent work in machine performance evaluation has focused on assembling large suites of realistic applications to be used as benchmarks, and in developing a more formal and systematic approach to benchmarking [SPEC89, Cybe90]. Computer manufacturers are using these suites to evaluate the overall performance of machines and to improve the designs of future machines and compilers. By concentrating on the performance of the whole system, however, it is not possible to explain why machines perform well on some benchmarks but badly on others, or to predict how they behave on
programs not included in the suites. Observed CPU performance is the result of the interactions between many hardware and software components, i.e., integer, floating point, and branch units, memory system, applications, libraries and optimizing compilers, and a comprehensive performance evaluation should characterize their respective contributions [Lind86]. Our research has focused in developing a methodology that addresses two problems: how to compare machines with different architectures in a meaningful way, and how to explain in detail performance results in terms of the different components of the system [Saav89, 92a, 92b].

The basis for our research has been to model all computers as machines that execute Fortran. By measuring the execution time for primitive Fortran operations on a given machine, and by counting the frequency of occurrence of the various operations in each program of interest, we have been able to predict with good accuracy the running time. We refer to this as our abstract machine model.

In this paper we focus on two problems, characterizing the performance improvement due to compiler optimization and extending our performance methodology to include the effects of optimization. We do this by addressing three different subproblems: 1) extending the abstract machine model to include optimization and using this new model to quantify and predict the execution time of optimized programs; 2) evaluating the effectiveness of different optimizing compilers in their ability to apply standard optimizations; and 3) evaluating the amount of optimization found in the SPEC suite and identifying distinctive features in the benchmarks which can be exploited by good optimizing compilers.

In Section 2 we begin by discussing the relevant work done with respect to evaluating the effectiveness of optimizing compilers. We then give a brief description of our methodology for CPU performance characterization, summarize our previous work, and discuss the inherent limitations of our model with respect to compiler optimization.

We then proceed in Section 3 by extending our methodology to account for the performance improvements due to optimization by using the concept of invariant optimizations. An optimization is invariant with respect to our abstract machine model if it is still possible to abstract from the optimized sequence of machine instructions the original operations embodied in the source code. This approach avoids the extremely difficult problem of having to predict how an arbitrary program will be modified by different optimizers. It assumes that the effect of optimization is now to cause the execution time of a given primitive operation to be reduced; in effect, optimization modifies the machine performance, not the program. We have found this approach to be quite successful in allowing us to predict the running times of optimized code.

In Section 4 we address the problem of characterizing and comparing different optimizing compilers in their ability to apply standard optimizations. We use a special benchmark consisting of a set of small kernels, each containing a single optimization, which detect the set of optimizations that optimizers can apply and the context in which they are detected. We show that even when most optimizers attempt to apply the same set of optimizations, there are some differences in their relative effectiveness, and these differences can significantly affect the performance improvement obtained on some programs.
Finally, in Section 5, we analyze the potential optimizations present in the SPEC Fortran benchmarks and how well current optimizers can detect them. We also discuss some problems in the benchmarks which can exploited by smart compilers to artificially improve the performance of the machine.

2. Previous Work and Background Material

In this section we review some of the work done in evaluating the effectiveness of optimizing compilers, and then give a brief description of our methodology for CPU performance evaluation. We note that most performance studies about optimizing compilers have focused on showing that they actually improve the execution time of programs, but have ignored other important aspects like evaluating their effectiveness in detecting optimizations or how often these optimizations occur in real applications.

The second part of this section introduces our methodology for performance evaluation, in particular it reviews the abstract machine execution model and discusses some of our previous results. In the following section we discuss the limitation of the model with respect to compiler optimization and how they can be overcome using the concept of invariant optimizations.

2.1. Previous Performance Studies in Compiler Optimization

Knuth, in 1971, was the first who attempted to quantify the potential improvement due to optimization [Knut71]. He statically and dynamically analyzed a large number of Fortran programs and measured the speedup that could be obtained by hand-optimizing them. He found that on the average a program could be improved by as much as a factor of $4^1$.

Papers reporting on the effectiveness of real optimizers were not published until the beginning of the eighties [Cock80, Chow83, BalH86, John86, Wolf85, Much86, Jaza86]. Most of these studies describe the set of optimizations that can be detected by the optimizers, but without specifying if they are detected on all data types or only on a small subset. As we will see in §4.2, very few optimizers are able to detect optimizations on all data types; this can result in a significant loss of potential improvement as a result of changing the precision or type in the declaration of variables. Another problem is the programs used to evaluate optimization in these studies tend to be small and thus the results may not be representative of real applications.

The performance of IBM’s PL/1L experimental compiler is evaluated in [Cock80]. The compiler has 3 levels of optimization. Although the paper describes which optimizations are carried out at each level, only the aggregate speedup is reported. On four programs, the amount of speedup obtained at the maximum level of optimization was 1.312. Chow [Chow83] who wrote the Uopt portable global optimizer at Stanford, gives

---

1 In the rest of this paper we quantify the improvement produced by an optimizer in terms of the speedup experienced by the program, i.e., the ratio between the unoptimized execution time to the optimized time. When reported, the overall speedup on all benchmarks is computed by taking the geometric mean of the individual speedups. In order to be consistent, we also follow these two rules when we describe work done by others. Because of this some of the numbers we quote here are not the same as those found in the original papers.
statistics about the number of times that each optimization was detected and for some of the optimizations he reports the amount of improvement produced. On 13 small Pascal programs, with no program larger than 160 lines, the average speedup was 1.705. He also found that the most effective optimizations were register allocation and backward code motion with speedups of 1.423 and 1.431 respectively when applied individually. Bal and Tanenbaum [BalH86] found using the Amsterdam Compiler Kit optimizer that the speedup produced on toy programs was 1.851, while the speedup on larger programs was only 1.220. Because the larger programs consisted of modules taken from a single application and were all written by the same people, it is not clear whether the difference in speedups can be attributed to the complexity of the programs or the ability of the programmers. A performance study based on the HP Precision Architecture global optimizer [John86] found that on the same programs used by Chow the average speedup was 1.381.

There had been other studies dealing with other aspects of optimization. Arnold [Arno83] reports on the effectiveness of the CYBER 205 vectorizing compiler in producing either vector or scalar versions of a loop as a function of the number of iterations. Richardson and Ganapathi [Rich89] have shown that certain types of interprocedural data flow analysis provides only marginal improvement on most of the programs in their suite. Callahan, Dongarra, and Levine have collected a large suite of tests for vectorizing compilers and have evaluated a large number of compilers [Call88]. Most commercial vectorizing compilers are based either on the VAST or KAP pre-compilers developed at Pacific Sierra Research and Kuck and Associates, which are compared in [Bras88]. Singh and Hennessy [Sing91] are studying the potential and limitations of automatic parallelization.

Most of the effectiveness of vectorizing compilers comes from making a dependence analysis of the program. People have realized that dependency analysis techniques [Bane88] can also be used to improve the performance of scalar machines. Recent research have proposed using this information to improve register allocation of subscript variables [Call90], increase locality inside nested loops [Port89, Ferr91, Wolf91], and reduce memory latency by using software prefetching in conjunction with lockup-free caches [Call91]. Although it will take some time before these techniques are incorporated into commercial compilers, they appear to be the best way of improving the performance of scientific programs. As optimizers become more and more powerful and complex, the need to evaluate their effectiveness and characterize how they affect real programs becomes increasingly important.

2.1.1. Factoring Out the Effect of Languages and Architectures

The programs used in the above studies have been written in C, Pascal, or a dialect of PL/1 and the speedups observed were in all cases smaller than two. We have found that Fortran compilers produce significantly larger speedups; we believe that this is because Fortran is inherently easier to optimize. First, Fortran programs appear to offer more opportunities for optimization as most of the work is inside highly nested loops.

---

2 The product of the individual speedups can be larger than the overall speedup because in some cases one optimization prevents the application of the other.
Performing an optimization inside one of these loops tends to significantly reduce the execution time. Second, Fortran programs tend to use large amounts of data stored in arrays, and the access to this data tends to follow simple regular patterns which can often be easily optimized. Third, subroutines in Fortran programs tend to be larger than for other languages, so there is more opportunity to find optimizations. Fourth, typical Fortran code inside basic blocks contains long sequences of arithmetic statements, rather than complicated sequences containing procedure calls, case statements, and other program structuring constructs, which impede optimization. Lastly, there are no pointers in Fortran, so detecting aliases is no problem for the optimizer. (Aliases do exist, because of common and equivalence statements, and subroutine parameters, but may be readily identified.) Normally, in the presence of aliases, optimizers are forced to make worst case assumptions.

One problem when comparing the improvements produced by different optimizers is how to factor out from the results the quality of the unoptimized code. It is clear that it is always possible to increase the speedup produced by an optimizer by generating worse unoptimized code.

Another factor that has to be taken into account is the effect of the architecture. Some machines are easier to generate code for than others. One of the arguments in favor of the RISC movement [Patt85] was that a simple architecture makes it easier to write better optimizers, as the number of combinations to consider is significantly smaller. An attempt to evaluate the effect of optimization on different architectures using again the Uopt optimizer is reported in [Cude89]. This study found that register architectures tend to benefit the most from optimization, as the optimization process introduces large numbers of temporaries which get assigned to registers, effectively eliminating many load and store instructions.

2.2. The Abstract Machine Performance Model

We call the approach we have used for performance evaluation the abstract machine performance model. The idea is that every machine is modeled as and is considered to be a high level language machine that executes the primitive operations of Fortran. We have used Fortran for three reasons: (a) Most standard benchmarks and large scientific programs are written in Fortran; (b) Fortran is relatively simple to work with; (c) Our work is funded by NASA, which is principally concerned with the performance of high end machines running large scientific programs written in Fortran. Our methodology could be straightforwardly used for other similar high level languages such as C and Pascal.

There are three basic parts to our methodology. In the first part, we analyze each physical machine by measuring the execution time of each primitive Fortran operation on that machine. Primitive operations include things like add-real-single-precision, store-single-precision, etc; the full set of operations is defined in [Saav89, 92a]. Measurements are made by using timing loops with and without the operation to be measured. Such measurements are complicated by the fact that some operations are not separable from other operations (e.g. store), and that it is very difficult to get precise values in the presence of noise (e.g. cache misses, task switching) and low resolution clocks [Saav89, 92a]. We have also called this machine analysis phase narrow spectrum benchmarking. This approach, of using the abstract machine model, is extremely powerful, since it saves
us from considering the peculiarities of each machine, as would be done in an analysis at the machine instruction level [Puet77].

The second part of our methodology is to analyze Fortran programs. This analysis has two parts. In the first, we do a static parsing of the source program, and count the number of primitive operations per line. In the second, we execute the program and count the number of times each line is executed. From those two sets of measurements, we can determine the number of times each primitive operation is executed in an execution of the entire program.

The third part of our methodology is to combine the operation times and the operation frequencies to predict the running time of a given program on a given machine without having run that program on that machine. As part of this process, we can determine which operations account for most of the running time, which parts of the program account for most of the running time, etc. In general, we have found our run time predictions to be remarkably accurate [Saav92a, 92b].

It is very important to note and explain that we separately measure machines and programs, and then combine the two as a linear model. We do not do any curve fitting to improve our predictions. The feedback between prediction errors and model improvements is limited to improvements in the accuracy of measurements of specific parameters, and to the creation of new parameters when the lumping of different operations as one parameter were found to cause unacceptable errors. The curve fitting approach has been used and has been observed to be of limited accuracy [Pond90]. The main problems with curve-fitting is that the parameters produced by the fit have no relation to the machine and program characteristics, and they tend to vary widely with changes in the input data.

In [Saav89] we presented a CPU Fortran abstract machine model consisting of approximately 100 abstract operations and showed that it was possible to use it to characterize the raw performance of a wide range of machines ranging from workstations to supercomputers. We used these characterizations to define and compare a set of reduced parameters synthesized from the abstract operations which represents the performance of different aspects of the machine. These reduced parameters makes it easier to make a direct comparison between machines as the parameters can be identified with specific subunits in the machine. We also introduced the notion of a performance shape, which represents graphically the overall performance of a machine; we also defined a metric of machine similarity which identifies machines with similar distribution of performance over the parameters. We showed that this metric is related to the amount of variance found in the relative results between pairs of machines.

In [Saav92a, 92b] we studied the characteristics of the SPEC and Perfect Club benchmarks using the same abstract machine model and showed that it is possible to predict the execution time of arbitrary programs on a large number of machines. Both of these studies assumed that programs were compiled and executed without optimization. In the next section we discuss how optimization can invalidate some of our assumptions and how it is possible to extend the model to remedy this situation.
2.3. Limitation of Our Model in the Presence of Optimization

An apparent limitation of our linear model is that it does not account for the program transformations induced by optimization. To state this formally, we describe our methodology with this equation:

\[ T_{A,M} = \sum_{i=1}^{n} C_{i,A} P_{i,M} = C_{A} P_{M}. \]  

(1)

Here \( C_{i,A} \) is the number of abstract operations of type \( i \) that program \( A \) executes, and \( P_{i,M} \) is the execution time of operation \( i \) on machine \( M \). In general, when we include optimization, both the decomposition of the program in terms of the abstract model \( (C_A) \) and the performance of the abstract operations \( (P_M) \) may change. \( C_A \) changes when the optimizer eliminates some part of the computation. The raw performance measurements represented by \( P_M \) change, because the compiler generates different sequences of machine instructions at different levels of optimization. Therefore, in general, the execution time equation when using an optimizing compiler should be

\[ T_{A,M,O} = \sum_{i=1}^{n} C_{i,A,O} P_{i,M,O} = C_{A,O} P_{M,O} \]  

(2)

Our problem here is to obtain \( C_{A,O} \) and \( P_{M,O} \) by only making an analysis of the program and running experiments with optimization enabled.

3. Extending the Abstract Model to Include Optimization

From the discussion of the preceding subsection we can proceed to classify optimizations according to how they affect eq. (1). In the first class (type I) we have optimizations which change the program’s distribution of abstract operations, either by removing or replacing some amount of code. Common subexpression elimination is one example of this type of optimization. Here all the abstract operations forming the subexpression are eliminated and replaced by a reference to the previously computed result. Applying a type I optimization has the effect, on equation (1), of changing \( C_A \), but without affecting \( P_M \). The difficulty in characterizing the performance improvement due to these optimizations is that we need to know how \( C_A \) changes, but without having any information about how an arbitrary optimizer works.

In the second class (type II) we have optimizations which only improve the sequence of machine instructions generated by the compiler to implement an abstract operation, but do not remove any abstract operations. This class not only includes improved machine code sequences, but also strength reduction, as explained below. Here one or several slow operations are replaced by a faster but equivalent sequence of operations. Type II optimizations change \( P_M \), while leaving \( C_A \) unchanged. We call these optimizations invariant with respect to the abstract decomposition of the program. The advantage to us of invariant optimizations over type I optimizations is that we can characterize the performance improvement of the former by just running our machine characterizer with optimization enabled. If the optimizer changes the code it generates when it encounters an abstract operation in a program, it does the same action when it encounters it in the machine characterizer; thus the performance effect of this change can be quantified.
Whether an optimization is of type I or II depends mainly on the level at which we define the abstract machine. If the abstract machine were defined at the level of the machine’s instruction set, then all optimizations would be of type I, since every machine instruction eliminated affects the decomposition of the program. If, on the other hand, the abstract operations consisted of different algorithms, then almost all optimizations are of type II. As long as the algorithm is not eliminated, changes to it are considered only as different implementations of the same abstract operation. Given the level of abstraction of our model, it happens that most source to source transformation are optimizations of type I, and low level transformations are optimizations of type II.

To illustrate the difference between invariant and non-invariant optimizations, consider the following code excerpt

```plaintext
DO 2 I = 1, N
   DO 1 J = 1, N
      X(I) = X(I) + Y(J,K) * Z(J,L)
   1 CONTINUE
2 CONTINUE
```

During program analysis we identify the different abstract operations, e.g., a floating point add (ARSL), floating point multiply (MRSLS), computing the addresses of a 1- and 2-dimensional array elements (ARR1 and ARR2), DO loop initialization and overhead (LOIN and LOOV), floating point store (SRSL). Combining this static decomposition with information on how many times each basic block is executed we can then obtain the contribution of this code to the total execution time

\[ Time = (P_{SRSL} + 2P_{ARR2} + 2P_{ARR1} + P_{MRSLS} + P_{ARSL} + P_{LOOV})N^2 + (P_{LOIN} + P_{LOOV})N + P_{LOIN}. \]

In table 1 we show the sequence of assembler instructions generated by the MIPS Co. f77 compiler version 1.21 for the innermost loop for each abstract operation (left column) without and with maximum optimization. (We have made inconsequential changes to the syntax of the machine instructions to make the code more readable.)

3.1. Optimization Viewed as an Optimized Implementation of the Abstract Machine

What the above example shows is that even when the two sequences of machine instructions, one unoptimized and the other optimized, are very different, we can still identify in both the original abstract operations. Thus in this case the optimizer has reduced the execution time, but the characterization of the program excerpt, in terms of our abstract machine, has not changed. We refer to these type of optimizations, which improve the execution time of a program but do not change the distribution of abstract operations, as being invariant with respect to the abstract machine model.

It is important to note that the optimizations applied to the program excerpt in table 1 are not only simple low-level optimizations. The compiler here has to apply strength reduction, backward code motion, and address collapsing in order to eliminate the 2 loads, 2 multiplies, and 4 add/sub operations in the sequence associated with ARR2. This requires identifying that some part of the address computation is invariant with respect to the loop induction variable so it can be moved out of the loop; and that the sequence of array addresses is generated by a linear recurrence, so future values can be
<table>
<thead>
<tr>
<th>abstract operation</th>
<th>assembler code without optimization</th>
<th>assembler code with optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr2</td>
<td>load r14, 80444(sp)</td>
<td>load r4, 24708(r3)</td>
</tr>
<tr>
<td></td>
<td>load r15, 40036(sp)</td>
<td>add r3, r3, 4</td>
</tr>
<tr>
<td></td>
<td>mul r24, r15, 100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r25, r14, r24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub r8, r25, 101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mul r9, r8, 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r10, r9, -40424</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r11, sp, 80464</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r12, r10, r11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>load r6, 0(r12)</td>
<td></td>
</tr>
<tr>
<td>arr2</td>
<td>load r13, 32(sp)</td>
<td>load r6, 17472(r4)</td>
</tr>
<tr>
<td></td>
<td>mul r15, r13, 100</td>
<td>add r4, r4, 4</td>
</tr>
<tr>
<td></td>
<td>add r24, r14, r15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub r25, r24, r24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mul r8, r8, 101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r9, r8, -80428</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r10, r9, r11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>load r8, 0(r10)</td>
<td></td>
</tr>
<tr>
<td>mrs1</td>
<td>mul f f10, f6, f8</td>
<td>mul f f8, f4, f6</td>
</tr>
<tr>
<td>arr1</td>
<td>sub r12, r14, 1</td>
<td>load f16, -428(r2)</td>
</tr>
<tr>
<td></td>
<td>mul f r13, r12, 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r15, r13, -424</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r24, sp, 80464</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r25, r15, r24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>load f16, 0(r25)</td>
<td></td>
</tr>
<tr>
<td>ars1</td>
<td>add f f18, f16, f10</td>
<td>add f f10, f16, f8</td>
</tr>
<tr>
<td>srs1</td>
<td>stor f18, 0(r25)</td>
<td>stor f10, -428(r2)</td>
</tr>
<tr>
<td>loov</td>
<td>load r8, 80444(sp)</td>
<td>add r2, r2, 4</td>
</tr>
<tr>
<td></td>
<td>add r9, r8, 1</td>
<td>br ne r2, r6, r35</td>
</tr>
<tr>
<td></td>
<td>stor r9, 80444(sp)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>load r11, 80440(sp)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>br ne r9, r11, r34</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Nonoptimized and optimized assembler code for the innermost loop. On the left side we show the abstract machine operations represented by the assembler code.

computed from previous ones using only adds. However, from our perspective, the optimized program still executes operation ARR2, even though the new version consumes fewer cycles. Therefore we consider the above optimizations invariant with respect to parameter ARR2, which now has a new ‘optimized’ execution time. We can do this as long as 2-dimensional array references can be optimized in a similar way by the compiler in most programs and in our program characterizer. For some optimizations this assumption is reasonable, but on others it is not. Overall, as we will observe, this assumption works well.

In the case that all optimizations are invariant, predicting the execution of the optimized version requires only taking the dot product between the unchanged abstract characterization of the program excerpt and the ‘optimized’ set of machine parameters. This ‘optimized’ machine characterization is obtained by using the optimized version of the machine characterizer to measure the parameter values.

The relevance of viewing optimization not as an attempt to improve the object code which executes on the same machine but as running the same abstract set of instructions on an ‘optimized’ machine, is that we effectively avoid having to predict how an
arbitrary optimizer would transform the program.

Although it is not always possible to know how optimization in general will affect a program, it is possible, for many programs, to obtain reasonable predictions by assuming that most of the optimization improvement comes from applying invariant optimizations. Under this assumption the execution time of an optimized program is

\[ T_{A,M,O} = \sum_{i=1}^{n} C_{i,A} P_{i,M,O} = C_A \cdot P_{M,O}. \] (3)

There are three main reasons why this approach works. First, optimizations are applied at a low level when most of the program structure is not present any more, so most of the improvement derived is from optimizing sequences of machines instructions and not from eliminating abstract operations. Second, optimizers are consistent in detecting optimizations. If an optimizer is capable of improving the code emitted by the compiler in the expansion of a particular abstract operation, then it can also do it in most of the other instances where the same sequence appears, such as in the machine characterizer. Lastly, the execution time of programs is normally determined by a small number of basic blocks, and it appears that for the programs we've studied, programmers try to eliminate obvious machine-independent optimizations on these blocks to guarantee that the programs will execute efficiently.

The second argument in the previous paragraph is worth discussing in more detail. Even when a type I optimization changes the distribution of abstract operations of programs by eliminating some operations, it can be considered an invariant optimization as long as the same operations are eliminated from all occurrences in all programs, including our machine characterizer. For example, suppose that a very good compiler is capable of eliminating at compile time all multiply operations. As long as the optimizer is always successful, we can include this optimization in our predictions, because our measurements with the machine characterizer will indicate that the execution time of the multiply operation is zero or close to zero. The corresponding execution time computed using this value will correspond to the actual execution time. Our focus in this subsection is in quantifying the performance effect of optimization and not in finding out which optimizations are applied. In §4 we characterize the particular optimizations that compilers can apply.

3.2. Limitations of Invariant Optimizations

The above approach to optimization works as long as the optimizer attempts to reduce the execution time of the programs without changing the original computations embodied in the source code. This, however, is not always the case. For example, a sophisticated vectorizing compiler can apply loop interchange, code motion, and loop unrolling [Paud86] to the previous code excerpt to dramatically reduce the number of operations and consequently the execution time\(^3\). These source to source transformations

\(^3\) Loop interchange transposes the order of the loops. This allows the compiler to detect that the expression \( \mathbf{x} (J, K) \ast \mathbf{z} (J, L) \) is invariant with respect to the induction variable \( I \) and hence can be moved out from the loop. The compiler can then identify that all elements of array \( \mathbf{x} \) get the same value, which can be computed only once and the result added to all elements.
produce the following equivalent piece of code

\[
\text{TMP} = 0.0 \\
\text{DO} 1 \text{ I = 1, N} \\
\text{TMP = TMP + Y(I,K) * Z(I,L)} \\
1 \text{ CONTINUE} \\
\text{DO} 2 \text{ I = 1, N} \\
\text{X(I) = TMP} \\
2 \text{ CONTINUE}
\]

The contribution of this code to the total execution time is

\[
\text{Time} = N \left( P_{\text{MRSL}} + P_{\text{ARSL}} + 2 \cdot P_{\text{ARR2}} + P_{\text{ARR1}} + P_{\text{SRLS}} + P_{\text{TRSL}} + 2 \cdot P_{\text{LOOV}} \right) + 2 \cdot P_{\text{LOIN}} + P_{\text{TRSL}}
\]

This equation is now linear with respect to the number of iterations instead of quadratic. This example shows that, in general, without detailed knowledge of which transformations are applied by the optimizer, it is not possible to predict the execution time after optimization.

3.3. Machine Characterizations Results with Optimization

In the previous section we argued that we can easily extend our model to include invariant optimizations, if we consider them as defining a faster machine rather than optimizing the object code. This ‘optimized machine’ has its own machine performance vector which is obtained by executing the system characterizer with optimization enabled. Furthermore we can apply to the performance vector the same metrics as in the unoptimized case. Thus, the concept of performance shape and machine similarity [Saav89] are well defined and provide useful information with respect to the effectiveness of optimization. In this section we compare different machine characterizations under various levels of compiler optimization.

<table>
<thead>
<tr>
<th>reduced parameters</th>
<th>HP 720</th>
<th>MIPS M/2000</th>
<th>Sparstation 1+</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-00</td>
<td>-01</td>
<td>-02</td>
</tr>
<tr>
<td>memory latency</td>
<td>108</td>
<td>104</td>
<td>61</td>
</tr>
<tr>
<td>integer add</td>
<td>95</td>
<td>60</td>
<td>29</td>
</tr>
<tr>
<td>integer multiply</td>
<td>442</td>
<td>419</td>
<td>170</td>
</tr>
<tr>
<td>logical operations</td>
<td>193</td>
<td>139</td>
<td>95</td>
</tr>
<tr>
<td>single prec. add</td>
<td>99</td>
<td>53</td>
<td>45</td>
</tr>
<tr>
<td>single prec. multiply</td>
<td>128</td>
<td>96</td>
<td>35</td>
</tr>
<tr>
<td>double prec. add</td>
<td>100</td>
<td>73</td>
<td>45</td>
</tr>
<tr>
<td>double prec. multiply</td>
<td>129</td>
<td>117</td>
<td>35</td>
</tr>
<tr>
<td>division</td>
<td>300</td>
<td>234</td>
<td>188</td>
</tr>
<tr>
<td>procedure calls</td>
<td>99</td>
<td>96</td>
<td>72</td>
</tr>
<tr>
<td>address</td>
<td>136</td>
<td>76</td>
<td>42</td>
</tr>
<tr>
<td>branches &amp; iteration</td>
<td>151</td>
<td>97</td>
<td>41</td>
</tr>
<tr>
<td>intrinsic functions</td>
<td>2561</td>
<td>2490</td>
<td>2477</td>
</tr>
</tbody>
</table>

Table 2: Optimization performance results in terms of the reduced parameters. Each parameter represents a particular characteristic of the machine and is computed from a subset of basic abstract machine parameters. All units are in nanoseconds. On the Sparstation 1+ the results for optimization levels 0 and 1 were almost identical, so we only report results for level 0.
We ran the system characterizer using different optimization levels on three high performance workstations. The complete results, including those without optimization, are shown in Appendix A (tables 13-17). Table 2 shows a set of thirteen parameters which were synthesized from the basic measurements.

The vector of reduced parameters can be used to characterize a machine and to compute the degree of similarity between machines. We can also use a graphical representation of performance called the performance shape (pershape [Saav89]), a type of Kiviat graph, as shown in figure 1. There we plot the (inverse of the) performance of each machine, at each level of optimization, normalized to the MIPS M/2000 with no optimization; each bar is on a logarithmic scale.

![Diagram of performance shapes](image)

**Figure 1:** Performance shapes (pershapes) of different optimization levels. The thirteen dimensions correspond to the same thirteen parameters used in table 1. All dimensions are normalized with respect to the MIPS M/2000 with optimization level 0.

The results in figure 1 clearly show that some abstract and reduced parameters benefit more from optimization than others. The parameters that benefit most are memory bandwidth, integer addition, floating point arithmetic operations, address computation, branching and iteration. Conversely, intrinsic functions show little if any improvement. This is because normally the same libraries are used at all optimization levels. In fact, the average execution time for intrinsic functions on the Sparcstation 1+
increases with the level of optimization, and on the MIPS M/2000 the average time at the maximum level of optimization is larger than for other two cases. This is because the call to an intrinsic function can inhibit optimizations that would otherwise occur to the surrounding code; our methodology attributes that loss of performance to (the presence of) the intrinsic function.

3.4. Execution Time Prediction For Optimized Code

In this section we show that we can predict, reasonably well, the execution time of optimized programs when most of the optimization improvement comes from the application of invariant transformations. The experiments were done using a large set of Fortran programs taken from the SPEC and Perfect Club suites, and also some popular benchmarks. A description of the programs and their dynamic statistics can be found in [Saav92b]. First, we compiled the programs using different levels of optimization and measured their respective execution times. At the same time we collected machine characterizations for the different levels of optimization. Using machine characterizations and the dynamic statistics of the programs, we predicted the expected execution times.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Minimum opt level</th>
<th>Maximum opt level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average RMS</td>
<td>Average RMS</td>
</tr>
<tr>
<td>HP-9000/720</td>
<td>21.84 %</td>
<td>+3.42 %</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>16.81 %</td>
<td>+10.64 %</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>22.87 %</td>
<td>-6.03 %</td>
</tr>
<tr>
<td></td>
<td></td>
<td>35.60 %</td>
</tr>
<tr>
<td></td>
<td></td>
<td>33.67 %</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25.34 %</td>
</tr>
</tbody>
</table>

Table 3: Summary of execution time errors by machine at the minimum and maximum levels of optimization. RMS represents the root mean square error. The plus (negative) sign for average errors indicate that the predictions were above (below) the real execution times.

In figure 2 we show the comparison between the real and predicted execution times for both optimized and unoptimized programs; the abbreviations for the various programs are explained in [Saav92a]. For each graph the vertical distance to the diagonal represents the error of the prediction. Although the scale is logarithmic and hence the errors appear smaller than they are, it is clear from the figure that the predictions even at the maximum optimization level are quite good. Tables 18-20 on Appendix B gives the exact execution times and relative errors. Summaries of the predictive errors, by machine and program, are presented in tables 3 and 4. The RMS error, shown in tables 3 and 4, is the square root of the average of the square of the individual errors. As expected the magnitude of the error increases with the optimization level, but this increase is relatively small with an average error of less than 11%. Note that the average actual run time increases relative to the predicted run time; that increase reflects optimizations that are not invariant.

Figure 2 (see also tables 18-20) clearly shows that some programs benefit more than others from optimization. For example, the execution time improvement of WHETSTONE on the four machines is only 20 percent; the smallest of all benchmarks. This is because of the relatively large number of intrinsic functions executed by the program, which do not run faster when the program is optimized.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>RMS</td>
</tr>
<tr>
<td>Doduc</td>
<td>+4.10 %</td>
<td>6.76 %</td>
</tr>
<tr>
<td>Fpppp</td>
<td>+5.93 %</td>
<td>11.74 %</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>-11.92 %</td>
<td>13.54 %</td>
</tr>
<tr>
<td>Matrix300</td>
<td>-37.87 %</td>
<td>39.00 %</td>
</tr>
<tr>
<td>Nasa7</td>
<td>-18.01 %</td>
<td>19.98 %</td>
</tr>
<tr>
<td>Spice2g6</td>
<td>+11.87 %</td>
<td>17.36 %</td>
</tr>
<tr>
<td>ADM</td>
<td>-18.17 %</td>
<td>22.73 %</td>
</tr>
<tr>
<td>QCD</td>
<td>+30.72 %</td>
<td>31.04 %</td>
</tr>
<tr>
<td>MDG</td>
<td>+12.15 %</td>
<td>15.43 %</td>
</tr>
<tr>
<td>TRACK</td>
<td>+16.71 %</td>
<td>17.16 %</td>
</tr>
<tr>
<td>BDNA</td>
<td>-13.18 %</td>
<td>14.27 %</td>
</tr>
<tr>
<td>OCEAN</td>
<td>+3.45 %</td>
<td>4.26 %</td>
</tr>
<tr>
<td>DYESM</td>
<td>-25.76 %</td>
<td>28.62 %</td>
</tr>
<tr>
<td>ARC2D</td>
<td>-35.28 %</td>
<td>35.63 %</td>
</tr>
<tr>
<td>TRFD</td>
<td>-22.04 %</td>
<td>26.37 %</td>
</tr>
<tr>
<td>FLO52</td>
<td>-19.50 %</td>
<td>22.86 %</td>
</tr>
<tr>
<td>Alamos</td>
<td>+2.71 %</td>
<td>11.15 %</td>
</tr>
<tr>
<td>Baskett</td>
<td>+16.33 %</td>
<td>16.58 %</td>
</tr>
<tr>
<td>Erathostenes</td>
<td>-14.58 %</td>
<td>18.54 %</td>
</tr>
<tr>
<td>Linpack</td>
<td>-6.25 %</td>
<td>15.74 %</td>
</tr>
<tr>
<td>Livermore</td>
<td>+12.41 %</td>
<td>17.16 %</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>+6.17 %</td>
<td>8.01 %</td>
</tr>
<tr>
<td>Shell</td>
<td>+6.21 %</td>
<td>21.60 %</td>
</tr>
<tr>
<td>Smith</td>
<td>-18.14 %</td>
<td>19.86 %</td>
</tr>
<tr>
<td>Wheatstone</td>
<td>-11.82 %</td>
<td>16.87 %</td>
</tr>
<tr>
<td>Totals</td>
<td>-4.83 %</td>
<td>20.59 %</td>
</tr>
</tbody>
</table>

Table 4: Summary of execution time errors by program at the minimum and maximum levels of optimization for the programs on table 3. The real and predicted execution times are given in tables 19-21 in Appendix B. RMS represents the root mean square error.

<table>
<thead>
<tr>
<th>level</th>
<th>&lt; 5 %</th>
<th>&lt; 10 %</th>
<th>&lt; 15 %</th>
<th>&lt; 20 %</th>
<th>&lt; 30 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>no optimization</td>
<td>15 (22.06)</td>
<td>26 (36.76)</td>
<td>39 (54.41)</td>
<td>46 (64.71)</td>
<td>61 (85.29)</td>
</tr>
<tr>
<td>max optimization</td>
<td>11 (12.86)</td>
<td>19 (24.29)</td>
<td>28 (37.14)</td>
<td>36 (48.57)</td>
<td>47 (68.29)</td>
</tr>
</tbody>
</table>

Table 5: Error distribution for execution time predictions

<table>
<thead>
<tr>
<th>level</th>
<th>&gt; 30 %</th>
<th>&gt; 40 %</th>
<th>&gt; 50 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>no optimization</td>
<td>11 (14.71)</td>
<td>3 (4.41)</td>
<td>0 (0.00)</td>
</tr>
<tr>
<td>max optimization</td>
<td>27 (35.71)</td>
<td>17 (22.86)</td>
<td>10 (12.86)</td>
</tr>
</tbody>
</table>

Table 5: Error distribution for the predicted execution times with and without optimization. For each error interval, we indicate the number of programs having errors that fall inside the interval (percentages inside parenthesis). The error is computed as the relative distance to the real execution time.

By modeling the execution time of a program using the abstract machine model in combination with the tools we have developed, we can get an understanding of how much optimization really affects the execution time of a program across many machines. We talk in more detail about this in §3.7.
Figure 2: Each broken diagonal line, which corresponds to a particular machine and optimization level combination, shows the accuracy of the predictions compared to the real execution times. The left end point of each diagonal line maps to (0.01, 0.01) and the right end point to (10000, 10000). Points along the diagonal are of the form \((T, T)\). All scales are in seconds.
3.5. Accuracy in Predicting the Execution Time of Optimized Programs

Our assumption that most of the performance improvement obtained from optimization is due to invariant optimizations is a simplification which is not necessarily valid on all programs. Nevertheless, the results of the previous section show that for most programs the assumption is reasonable. In table 5 we compare the distribution of errors for both non-optimized and optimized programs; we can see that for maximum optimization the average error increases. For the results shown in figure 2, table 5 shows that while 85% of the non-optimized predictions are within 30% of the real execution time, this value decreases to 68% for optimized programs. Moreover, almost 13% of the predictions have errors of more than 50%, while none of the non-optimized prediction have errors of that magnitude.

If a program exhibits a significantly larger positive prediction error at the maximum optimization level than it does with no optimization, then it is probably the case that the error is the result of ignoring non-invariant optimizations. In table 4 we see several programs for which this is true. An analysis of the source code shows that in these cases, optimizers are applying optimizations that are not invariant. For example, the code excerpt below taken from QCD contributes significantly to the total execution time. It contains many opportunities for the compiler to apply common subexpression elimination (3*I+P+1, 3*J+Q+1, and 3*K+R+1) and thus significantly reduce the execution time.

```
DO 2 I = 0, 2
DO 2 P = 0, 2
DO 2 J = 0, 2
DO 2 Q = 0, 2
DO 2 K = 0, 2
IF (EPSILO(I+1,J+1,K+1) .NE. 0) THEN
  DO 3 R = 0, 2
    IF (EPSILO(P+1,Q+1,R+1) .NE. 0) THEN
      FAC = EPSILO(I+1,J+1,K+1) * EPSILO(P+1,Q+1,R+1)
      TOT(1) = TOT(1) + FAC * U1(1,3*I+P+1) * U2(1,3*J+Q+1) * U3(1,3*K+R+1)
      TOT(1) = TOT(1) - FAC * U1(1,3*I+P+1) * U2(2,3*J+Q+1) * U3(1,3*K+R+1)
      TOT(1) = TOT(1) - FAC * U1(1,3*I+P+1) * U2(2,3*J+Q+1) * U3(2,3*K+R+1)
      TOT(1) = TOT(1) - FAC * U1(1,3*I+P+1) * U2(1,3*J+Q+1) * U3(2,3*K+R+1)
      TOT(2) = TOT(2) + FAC * U1(1,3*I+P+1) * U2(1,3*J+Q+1) * U3(2,3*K+R+1)
      TOT(2) = TOT(2) + FAC * U1(1,3*I+P+1) * U2(2,3*J+Q+1) * U3(1,3*K+R+1)
      TOT(2) = TOT(2) + FAC * U1(2,3*I+P+1) * U2(1,3*J+Q+1) * U3(1,3*K+R+1)
      TOT(2) = TOT(2) - FAC * U1(2,3*I+P+1) * U2(2,3*J+Q+1) * U3(2,3*K+R+1)
    ENDIF
  CONTINUE
ENDIF
```

Common subexpression elimination in this context is not an invariant optimization as defined in §3.1. Replacing an arithmetic expression by a reference to a previously computed equivalent value eliminates the abstract operations involved and thus distorts
our predictions. This is what happens on QCD, for which all of our predictions are greater than the real time; on two of the machines machines the errors are as high as 47% and 81% (tables 18 and 19).

3.6. Improving Predictions in the Presence of Non-Invariant Optimizations

We can improve our predictions of run times by identifying the applicable non-invariant optimizations and performing them manually on the source code. By applying common subexpression elimination to the previous example, we obtain the equivalent code shown below.

```fortran
DO 2 I = 0, 2
  DO 2 P = 0, 2
    DO 2 J = 0, 2
      DO 2 K = 0, 2
        IF (EPSILON(I+1,J+1,K+1) .NE. 0) THEN
          IF (EPSILON(P+1,Q+1,R+1) .NE. 0) THEN
            FAC = EPSILON(I+1,J+1,K+1) * EPSILON(P+1,Q+1,R+1)
            I3 = 3 * I + P + 1
            J3 = 3 * J + Q + 1
            K3 = 3 * K + R + 1
            T11 = U1(1,I3) * U2(1,J3)
            T12 = U1(1,I3) * U2(2,J3)
            T21 = U1(2,I3) * U2(1,J3)
            T22 = U1(2,I3) * U2(2,J3)
            U31 = U3(1,K3)
            U32 = U3(2,K3)
            T111 = T11 * U31
            T112 = T11 * U32
            T121 = T12 * U31
            T122 = T12 * U32
            T211 = T21 * U31
            T212 = T21 * U32
            T221 = T22 * U31
            T222 = T22 * U32
            TOT(1) = TOT(1) + FAC * (T111 - T221 - T122 - T212)
            TOT(2) = TOT(2) + FAC * (T112 + T121 + T211 - T222)
          ENDIF
        CONTINUE
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Here the values of common subexpressions are computed once and stored in variables $I3$, $J3$, and $K3$. In a similar way, we can eliminate other common subexpressions and in this way reduce the number of integer operations from 60 to 15 and the floating point operations from 37 to 23. After making the above changes, we found that on all machines the prediction errors were less than 30%.

By distinguishing the invariant and non-invariant optimizations, we can assess the performance impact of each, because the performance improvement due to non-invariant

---

Footnote 4: Although $I3$ and $J3$ are invariant with respect to the induction variables of the two innermost loops, it is not profitable to move the code outside the loops because the two IFs eliminate a large fraction of the innermost iterations.
optimizations is equal to the difference between our prediction, considering only invariant optimizations, and the real execution time.

### 3.7. Amount of Optimization in Benchmarks

By comparing the execution times before and after optimization for several different compilers, we can measure how much potential optimization exists in programs. In table 6 we show the program speedup achieved by each optimization level for the three machines previously discussed.

In §2.1 we mentioned that previous studies on the effectiveness of optimizing compilers for languages like C, Pascal, and PL/1 reported speedups of less than a factor of 2. The results in table 6, however, show that at the maximum level of optimization the speedups observed on Fortran programs are frequently larger than 2, with some programs experiencing speedups of more than a factor of 5.

<table>
<thead>
<tr>
<th>program</th>
<th>HP 720</th>
<th>MIPS M/2000</th>
<th>Sparcstation 1+</th>
<th>Geom. Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doduc</td>
<td>1.307</td>
<td>1.213 21</td>
<td>1.255 21</td>
<td>1.439 20</td>
</tr>
<tr>
<td>Fpppp</td>
<td>1.344</td>
<td>2.000 22</td>
<td>1.222 23</td>
<td>1.479 19</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>1.504</td>
<td>3.497 10</td>
<td>1.445 10</td>
<td>1.866 16</td>
</tr>
<tr>
<td>Matrix300</td>
<td>1.377</td>
<td>3.413 11</td>
<td>1.263 14</td>
<td>3.788 2</td>
</tr>
<tr>
<td>Nasa7</td>
<td>1.477</td>
<td>3.318 14</td>
<td>1.300 11</td>
<td>3.759 3</td>
</tr>
<tr>
<td>Spice2g6</td>
<td>1.345</td>
<td>2.560 19</td>
<td>1.250 19</td>
<td>1.231 21</td>
</tr>
<tr>
<td>ADM</td>
<td>1.305</td>
<td>4.000 6</td>
<td>1.372 -</td>
<td>2.506 10</td>
</tr>
<tr>
<td>QCD</td>
<td>1.374</td>
<td>2.793 16</td>
<td>1.351 18</td>
<td>1.443 18</td>
</tr>
<tr>
<td>MDG</td>
<td>1.215</td>
<td>1.698 24</td>
<td>1.250 20</td>
<td>1.208 25</td>
</tr>
<tr>
<td>TRACK</td>
<td>1.316</td>
<td>1.786 23</td>
<td>1.377 22</td>
<td>1.318 23</td>
</tr>
<tr>
<td>BDNA</td>
<td>1.414</td>
<td>2.890 15</td>
<td>1.381 16</td>
<td>1.237 22</td>
</tr>
<tr>
<td>OCEAN</td>
<td>1.370</td>
<td>3.891 7</td>
<td>1.408 8</td>
<td>2.066 8</td>
</tr>
<tr>
<td>DYFESM</td>
<td>1.468</td>
<td>6.993 3</td>
<td>1.335 3</td>
<td>4.367 3</td>
</tr>
<tr>
<td>ARC2D</td>
<td>1.430</td>
<td>4.878 5</td>
<td>1.368 7</td>
<td>2.118 6</td>
</tr>
<tr>
<td>TRFD</td>
<td>1.664</td>
<td>7.143 2</td>
<td>1.361 4</td>
<td>3.690 5</td>
</tr>
<tr>
<td>FLO52</td>
<td>1.460</td>
<td>8.333 1</td>
<td>1.360 2</td>
<td>3.610 4</td>
</tr>
<tr>
<td>Alamos</td>
<td>1.397</td>
<td>3.344 12</td>
<td>1.311 5</td>
<td>1.362 11</td>
</tr>
<tr>
<td>Baskets</td>
<td>1.316</td>
<td>3.333 13</td>
<td>1.370 13</td>
<td>2.331 9</td>
</tr>
<tr>
<td>Eratosthenes</td>
<td>1.300</td>
<td>2.597 18</td>
<td>1.305 15</td>
<td>1.667 17</td>
</tr>
<tr>
<td>Linpack</td>
<td>1.600</td>
<td>3.831 4</td>
<td>1.410 9</td>
<td>2.584 7</td>
</tr>
<tr>
<td>Livermore</td>
<td>1.473</td>
<td>2.703 17</td>
<td>1.570 12</td>
<td>2.045 12</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>1.348</td>
<td>2.545 20</td>
<td>1.429 17</td>
<td>2.000 15</td>
</tr>
<tr>
<td>Shell</td>
<td>1.634</td>
<td>4.902 4</td>
<td>1.592 9</td>
<td>1.357 4</td>
</tr>
<tr>
<td>Smith</td>
<td>1.350</td>
<td>3.597 6</td>
<td>1.282 6</td>
<td>2.000 13</td>
</tr>
<tr>
<td>Whetstone</td>
<td>1.218</td>
<td>1.647 25</td>
<td>1.200 24</td>
<td>1.300 24</td>
</tr>
<tr>
<td>Geom. Mean</td>
<td>1.392</td>
<td>3.271 1</td>
<td>1.348 2</td>
<td>1.973 2</td>
</tr>
</tbody>
</table>

Table 6: Optimization speedups under different optimization levels. Each speedup is computed by taking the ratio between the nonoptimized and optimized execution times. The last column gives the geometric mean of the machine speedups obtained at the maximum level of optimization. The small number on the right of each speedup indicates its relative magnitude, with the numeral 1 representing the largest speedup. Program ADM did not execute correctly on the MIPS M/2000 at the maximum optimization.
The results of table 6 show that speedups on FLO52, DYFESM, TRFD, ARC2D, and SHELL are the highest of all programs, while those of DODUC, FPPPP, TRACK, MDG, and WHETSTONE are the lowest. Our analysis of the source code shows that the programs in each group share similar characteristics. For example, the sizes of the most time-consuming basic blocks of the programs with the highest speedups are quite small. These consist of a few arithmetic statements where most of the operands are elements of multi-dimensional arrays. Our examination of those programs shows that most of the optimization improvement comes from collapsing the computation of the array addresses, good register allocation, and eliminating loads and stores of temporary values.

The programs with the smallest speedups are different. They tend to have substantially larger basic blocks. For example, the largest basic block on FPPPP has 590 lines of mostly scalar code. Here register files having as many as 32 or 64 registers cannot keep most of the variables in registers between their definition and use. Furthermore, on these programs, most of the operands are either scalars or one-dimensional arrays, so address collapsing, the elimination of time consuming address calculations in multi-dimensional arrays, does not produce very much improvement. They also tend to execute a larger number of intrinsic functions whose execution is mostly unaffected by optimization. This is also the case for MDG and WHETSTONE. Further discussion of the optimizations possible in these programs appears in §5.1.

<table>
<thead>
<tr>
<th>machines</th>
<th>Coefficient of Correlation</th>
<th>level of significance</th>
<th>Spearman’s Rank Correlation</th>
<th>level of significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720 and MIPS M/2000</td>
<td>0.8677</td>
<td>.0002</td>
<td>0.9417</td>
<td>.0003</td>
</tr>
<tr>
<td>HP 720 and Sparc 1+</td>
<td>0.7390</td>
<td>.0009</td>
<td>0.7954</td>
<td>.0012</td>
</tr>
<tr>
<td>MIPS M/2000 and Sparc 1+</td>
<td>0.5656</td>
<td>.0070</td>
<td>0.7652</td>
<td>.0020</td>
</tr>
</tbody>
</table>

Table 7: Coefficient of correlation and Spearman’s rank correlation of pairwise optimization speedup results. The statistical significance level gives the probability that there is not a positive correlation involved.

It is dangerous to draw conclusions about the effectiveness of the different optimizers from the speedup results of table 6. The overall speedup is as much a function of the quality of the non-optimized object code as it is of the optimizer, since it is always possible to improve the overall speedup by generating worse non-optimized code. This is particularly true for the HP 720, for which the overall speedup is significantly higher because the compiler generates native code for the 700 series only at the maximum level of optimization. For compatibility reasons, the object code at low levels of optimization is for the 800 series, which is emulated on the 720 in software.

Program SHELL is a good example of how the quality of nonoptimized code affects the amount of speedup observed on different programs. This benchmark is one of the few integer programs in our suite and implements shellsort. As Table 6 shows, SHELL is the program with the largest speedup on the MIPS M/2000 (6.289), and is one of the top four for the HP 720 (4.902). On the other hand, the speedup on the Sparcstation 1+ is significantly lower (2.093), even lower than the overall improvement for all programs (2.296). The reason for this is not because the Sun’s optimizer fails to improve the code, but is due to the fact that with no optimization, the MIPS M/2000 and HP 720 generate especially poor code. This is evident in the number of machine instructions generated by
each compiler. On the Sparcstation 1+, the number of instructions changes from 41 without optimization to 23 with optimization. The corresponding numbers for the MIPS M/2000 are 74 and 16, with speedups of 1.873 and 4.625. This discrepancy is clearly present in the actual execution times. Benchmark results normally rate the MIPS M/2000 as being at least 50% faster than the Sparcstation 1+. The results for SHELL on tables 19 and 20 (Appendix B), however, indicate that at low levels of optimization the Sparcstation 1+ is faster than the MIPS M/2000 (0.95 sec vs. 1.64 sec and 0.70 sec vs. 1.03 sec). It is only at the maximum optimization level that the MIPS M/2000 exhibits a smaller execution time (0.43 sec vs. 0.26 sec).

We can test if there is positive correlation between the amount of speedup produced by pairs of optimizers on these benchmarks, by computing either the coefficient of correlation or the Spearman's rank correlation coefficient. Table 7 gives the value of the coefficients and the level of significance for the three combinations. As is evident, there are substantial but not perfect correlations in the speedup produced by the three compilers.

4. The Characterization of Compiler Optimizations

In the last section we discussed how to measure and predict the performance improvement produced by optimizers. In this section we characterize the set of optimizations that compilers actually apply, and in which contexts. The context indicates whether a particular optimization can be performed on all data types or only on a subset of them. We are also interested in knowing if the optimization is detected when it is present inside a basic block and/or across basic blocks. In what follows, we refer to a local optimization as one that is detected inside a basic block and a global optimization when it spans more than one basic block.

Our approach to detecting optimizations is similar in some respects to the way we characterize basic machine performance [Saav89]. We have developed a Fortran program consisting of a number of tests which detect individual optimizations; each test is made separately for integers, floating point or mixed mode expressions. When appropriate, we also test for the local and global cases.

We detect whether a particular optimization is applied or not by running experiments which show a difference in their execution time only when the optimization is performed. In this way we can avoid having to analyze the assembler code. Each optimization test consists of two almost identical measurement where the only difference between them is that the second measurement contains a potential optimization. The running time of the two cases differs significantly only if the optimization is performed. Each experiment is repeated 20 times to collect a large statistic and a post-processor computes the average execution times of each experiment (μ₁ and μ₂) and the significance level of the following statistical test: μ₁ ≤ μ₂. If there is sufficient evidence to reject the null hypothesis, then we can assume that the optimization was performed. The level of significance represents the probability that random variations in our measurements would appear as supporting the conclusion that the optimization was detected when in fact it was not. Nevertheless, in all cases we have double checked that the optimizations were applied by analyzing the assembler code.

Figure 3 illustrates the basic structure of our experiments. This example is one of the tests for detecting local dead code elimination. The two corresponding innermost
DO 2 J = 1, 20
TO = SECOND (P)
DO 1 I = 1, ITER
W1 = X * W1 + (A * (B * C))
W2 = X * W2 + ((A * B) * C)
W3 = Y * W3 + ((C * A) * B)
A = XA - A
B = XB - B
C = XC - C
W1 = Y * W1 + (A * (B * C))
W2 = Y * W2 + ((A * B) * C)
W3 = X * W3 + ((C * A) * B)
A = XA - A
B = XB - B
C = XC - C
1 CONTINUE
T(J) = SECOND (P) - TO
2 CONTINUE

DO 4 J = 1, 20
TO = SECOND (P)
DO 3 I = 1, ITER
W1 = X * W1 + (A * (B * C))
W2 = X * W2 + ((A * B) * C)
W3 = Y * W3 + ((C * A) * B)
A = XA - A
B = XB - B
C = XC - C
W1 = X * A + (A * (B * C))
W2 = Y * W1 + ((A * B) * C)
W3 = Y * W2 + ((C * A) * B)
A = XA - A
B = XB - B
C = XC - C
3 CONTINUE
T(J) = SECOND (P) - TO
4 CONTINUE

Figure 3: A particular experiment to detect dead code elimination. On the left hand experiment all definitions inside the innermost loop are used at least once, while on the right hand experiment the topmost definitions of W1, W2, and W3 are not used. The three definitions can be eliminated by the optimizer.

loops are almost identical with only one difference: in the right hand side, the first set of definitions of variables W1, W2, and W3 are not used subsequently by any other statement. Furthermore, these definitions are killed by the second set of definitions to the same variables. Formally, we say that there are no forward dependencies having as source the first definitions. Hence, if the compiler can detect this, it can eliminate their computation. In contrast, this does not occur on the left side where every definition is the source of a forward dependency. Eliminating the first three statements on the second experiment reduces the execution time between 25% and 50% on most machines.

4.1. Standard Optimizations Detected

The types of optimizations that we are interested in detecting are machine-independent. This is consistent with our methodology which permits comparing different machines, and in this case their compilers, by providing a unified representation of the execution while ignoring machine-level details. Machine-dependent optimizations, like those performed by peephole optimizers, are invariant with respect to our model. Most machine-independent optimizations detected by current optimizers have been known for many years. A good reference describing these optimizations and the general problem of compiler optimization is [AhoA86]. [Chow83 and BalH86] describe how optimizations are implemented in a real compiler. The following are the optimizations that we currently detect:

- **Constant Folding**: replace symbolic constants by their actual values and evaluate the resulting expressions at compile time. If during this process other variables get a recently computed constant value, then their values are again propagated until no more constant expressions remain. The current emphasis on program modularity and portability has increased the
use of symbolic constants and correspondingly the importance of applying this optimization.

- **Common Subexpression Elimination**: identify two or more identical subexpressions in a region without an intervening definition of any of the relevant variables. Compute the subexpression at the beginning and replace subsequent computations by a reference to a temporary variable holding the result of the computation.

- **Code Motion**: identify expressions or statements which are invariant with respect to the induction variables of the loop and are computed unnecessarily on every iteration, and to move them out of the loop. The performance improvement obtained is proportional to the number of times the loop is executed. In scientific programs this is one of the most important optimizations along with address collapsing. Both of them are used in conjunction in the optimization of array references.

- **Dead Code Elimination**: in some programs there are pieces of code which can be statically proved never to be executed or whose execution does not have any semantic effect on the final computation. This code can be safely eliminated by the compiler to reduce the execution time and/or the object code size. Although this optimization does not appear very promising, as most programmers do not deliberately write needless code, occasionally some statements become dead as the result of applying other optimizations, or as the result of revisions to the program.

- **Copy Propagation**: some optimizations like common subexpression elimination, code motion, and address collapsing create large number of copy instructions, e.g., $x = y$. By replacing uses of the copy with the original variable it is possible to simplify the code and expose new optimizations. Optimizations that benefit from copy propagations are common subexpression elimination and register allocation.

- **Address Collapsing**: eliminate slow address computations for multi-dimensional array elements in innermost loops by precomputing outside the loop the addresses of the elements referenced in the first iteration and updating their values by adding a constant in subsequent iterations. This optimization is based on the observation that in the majority of nested loops the sequence of machine addresses associated with a specific array reference form an arithmetic progression, which is completely determined by the first value and the increment.

- **Strength Reduction**: this optimization is a generalization of address collapsing as it attempts to replace a time-consuming computation with an equivalent but faster one. One example is replacing an exponentiation having a small integer exponent which is known at compile time with a series of multiplications. Similarly, multiplies can often be replaced by additions. On array references, the combination of strength reduction and code motion makes it possible to collapse address computations.

- **Subroutine Inlining**: substitute for a call to a subroutine the actual subroutine code. This avoids the overhead of the call, and exposes optimizations present at the site of the call. Although most optimizers claim that they do subroutine inlining, they tend to differ substantially in the amount of integration they perform.

- **Loop Unrolling**: expand several iterations of the loop into a single basic block and hence expose new optimization opportunities. This also reduces the impact of the loop overhead.

In this paper we have concentrated on scalar optimizations. But in addition to the above optimizations, there are other program transformations which have been designed to exploit vector and parallel hardware. Some of these like loop distribution, loop interchange, loop fusion, loop peeling, and stripmining are used to help compilers in recognizing hardware vector instructions [Paud86, Alle87, Hira91]. A description of a large test suite and evaluation of vectorizing Fortran compilers can be found in [Call88].

---

5 *Loop distribution* separates independent statements inside a single loop into multiple loops which can be optimized independently [Hira91]. *Loop fusion* transforms two adjacent loops into a
<table>
<thead>
<tr>
<th>Machine</th>
<th>Compiler</th>
<th>Name/Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAX-11/785</td>
<td>BSD Unix F77 1.0</td>
<td>arpa.berkeley.edu</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>MIPS F77 2.0</td>
<td>mammoth.berkeley.edu</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>Sun F77 1.3</td>
<td>heffal.berkeley.edu</td>
</tr>
<tr>
<td>VAX-11/785</td>
<td>Ultrix Fort 4.5</td>
<td>pioneer.arc.nasa.gov</td>
</tr>
<tr>
<td>Amdahl 5860</td>
<td>Amdahl F77 2.0</td>
<td>prandl.nas.nasa.gov</td>
</tr>
<tr>
<td>CRAY Y-MP/8128</td>
<td>CRAY CFT77 4.0.1</td>
<td>reynolds.arc.nasa.gov</td>
</tr>
<tr>
<td>IBM RS/6000 530</td>
<td>IBM XL Fortran 1.1</td>
<td>coyote.berkeley.edu</td>
</tr>
<tr>
<td>Motorola M88K</td>
<td>Motorola F77 2.0b3</td>
<td>rumble.berkeley.edu</td>
</tr>
</tbody>
</table>

Table 8: List of machines with their respective Fortran compilers.

4.2. Optimization Results

We have run our experiments on several optimizing compilers and for different levels of optimization. In table 8 we give the list of machines along with their corresponding compilers. The complete results are presented in tables 21-26 in Appendix C, while tables 9-11 summarize the same information. The Appendix’s tables indicate for each optimization and different context (integer, float and mixed), whether the optimization was detected or not. In our experiments we make a distinction between local and global optimizations. A local optimization (tables 9-10) is one in which the optimization and all the information needed for its detection are found within a single basic block. A global optimization (table 11) requires the propagation of control and data flow information across basic block boundaries. In these tables, a ‘yes’ or ‘no’ entry indicates that the optimizer was able to detect all or none of the optimizations in the tests. The other two alternatives, two out of three and one out of three, correspond to entries ‘partial’ and ‘marginal’, for the three cases of real, integer and mixed mode computations. The results show that some compilers are only able to apply optimizations under certain conditions and not on all cases.

The optimization results for constant folding illustrate the difficulties in evaluating the effectiveness of an optimizer. While almost all the compilers are able to propagate integer constants inside a basic block, with the exception of the f77 BSD Unix and Amdahl compilers, the situation is less clear for floating point constant and global constant propagation. The Sun Fortran compiler does not apply constant propagation for floating point or across basic blocks, while the fort Ultrix compiler from DEC implements constant propagation on all data types but only inside a basic block. For the MIPS compiler, constant propagation is applied in the local and global context only for integers. For floating point, the value of a variable known at compile time is propagated only if the variable is assigned a constant value, but not if it gets the constant as a result of evaluating an expression.

---

single loop so as to reduce the loop overhead. Loop collapsing transforms two nested loops into a single one so as to increase the effective vector length. Stripmining transforms a single loop into two nested loops when the number of iterations of the original loops is much larger than the number of elements in the vector registers [Paud86]. Loop Fusion and loop collapsing are, respectively, the inverse transformations of loop distribution and stripmining.
Table 9: Summary of local optimizations. Each entry summarizes how well the optimizer detects the optimization using integer, floating point, and mixed data types in arithmetic expressions. These optimizations do not extend beyond a single basic block.

<table>
<thead>
<tr>
<th>compiler</th>
<th>constant folding</th>
<th>common subexpr elim</th>
<th>code motion</th>
<th>copy propagation</th>
<th>dead code elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSD Unix F77 1.0</td>
<td>no</td>
<td>partial</td>
<td>marginal</td>
<td>partial</td>
<td>no</td>
</tr>
<tr>
<td>Mips F77 2.0 -O2</td>
<td>partial</td>
<td>yes</td>
<td>yes</td>
<td>partial</td>
<td>yes</td>
</tr>
<tr>
<td>Mips F77 2.0 -O1</td>
<td>partial</td>
<td>yes</td>
<td>no</td>
<td>marginal</td>
<td>no</td>
</tr>
<tr>
<td>Sun F77 1.3 -O3</td>
<td>marginal</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Sun F77 1.3 -O2</td>
<td>marginal</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>partial</td>
</tr>
<tr>
<td>Sun F77 1.3 -O1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Ultrix Fort 4.5</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Amdahl F77 2.0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>CRAY CFT77 4.0.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IBM XL Fortran 1.1</td>
<td>yes</td>
<td>partial</td>
<td>yes</td>
<td>partial</td>
<td>yes</td>
</tr>
<tr>
<td>Motorola F77 2.0b3</td>
<td>marginal</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 10: Summary of global optimizations. Each entry summarizes how well the optimizer detects the optimization using integer, floating point, and mixed expressions. These optimizations cover more than one basic block.

<table>
<thead>
<tr>
<th>compiler</th>
<th>constant folding</th>
<th>common subexpr elim</th>
<th>code motion</th>
<th>copy propagation</th>
<th>dead code elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSD Unix F77 1.0</td>
<td>no</td>
<td>no</td>
<td>marginal</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Mips F77 2.0 -O2</td>
<td>partial</td>
<td>yes</td>
<td>yes</td>
<td>marginal</td>
<td>yes</td>
</tr>
<tr>
<td>Mips F77 2.0 -O1</td>
<td>partial</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Sun F77 1.3 -O3</td>
<td>no</td>
<td>yes</td>
<td>partial</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Sun F77 1.3 -O2</td>
<td>no</td>
<td>yes</td>
<td>partial</td>
<td>no</td>
<td>partial</td>
</tr>
<tr>
<td>Sun F77 1.3 -O1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Ultrix Fort 4.5</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>partial</td>
<td>yes</td>
</tr>
<tr>
<td>Amdahl F77 2.0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>CRAY CFT77 4.0.1</td>
<td>yes</td>
<td>partial</td>
<td>partial</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>IBM XL Fortran 1.1</td>
<td>partial</td>
<td>partial</td>
<td>yes</td>
<td>marginal</td>
<td>yes</td>
</tr>
<tr>
<td>Motorola F77 2.0b3</td>
<td>no</td>
<td>partial</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Common subexpression elimination is successfully detected by most compilers in all contexts. Although the IBM XLF compiler identified almost all common subexpressions, it missed a couple which involved floating point adds and multiplies. The reason is that the RS/6000 series provides, in addition to the normal add and multiply operations, a combined multiply-add instruction. In our experiments the compiler generated for two occurrences of the same subexpression, a multiply followed by an add in one case, but a single multiply-add for the other case. As a result of this, it did not recognize that the two expressions were identical. Missing an optimization as a result of applying another, however, is in many cases acceptable if the first optimization provides a better improvement.
Table 11: Additional optimizations. These optimizations are tested using a single data type, as their application is not affected by this kind of context. Here partial and marginal have a different meaning than on tables 8 and 9. Instead of summarizing the results of several experiments, they represent the effectiveness of the optimizers on a single test.

Table 11 shows that our tests detected that three compilers have some ability to inline procedures, but only the CRAY CFT77 compiler takes full advantage of it. In the case of MIPS f77 2.0, the compiler does not perform an actual inline substitution. The only transformation done is that the compiler does not use a new stack frame for the leaf procedure, but instead execution is carried out on the caller’s frame [Chow86]. In contrast, a real inline substitution is done by the IBM XLF 1.1 compiler [O’Br90], but here the insertion of unnecessary extra code obscures optimizations that inlining should have exposed. Only the CRAY’s CFT77 compiler was able to detect all optimizations present after proper inlining.

4.3. Correlation Between Different Optimizing Compilers

An interesting question to consider is how well different compilers correlate in their ability to improve the execution time of individual programs. If indeed there is a strong correlation between the amount of optimization obtained by different compilers, then knowing how much one optimizer reduces the execution time of a program would allow us to estimate the reduction on the other optimizer. Thus, this would give us alternative way of predicting execution times that would work, not only for invariant optimizations, but also for non-invariant ones.

A suite of programs was run both with and without optimization and the ratio of run times was computed. We then computed the coefficient of correlation between pairs of optimizers and the level of significance involved. In figure 4 we show the scattergrams and include on each one the best fit to the data. Table 12 gives the numerical values for the slope, y-intercept, correlation coefficient, and level of significance.

As expected there is a positive correlation between all optimizers; on the average more improvement by one compiler means more improvement in the other. Unfortunately, the correlation across compilers does not appear to be strong enough to make this approach better than estimating the execution time using the concept of invariant optimizations.
Figure 4: Correlation between execution time improvements of various optimizing compilers. Each graph includes the best linear fit.
Table 12: Slope, y–intercept, correlation coefficient, and level of significance for pairs of compilers.

<table>
<thead>
<tr>
<th>Compiler 1</th>
<th>Compiler 2</th>
<th>Obs.</th>
<th>Slope</th>
<th>y Intercept</th>
<th>Correlation coefficient</th>
<th>Level of significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSD Unix F77 1.0</td>
<td>MIPS F77 2.0</td>
<td>7</td>
<td>1.1883</td>
<td>-0.3600</td>
<td>0.8294</td>
<td>0.020</td>
</tr>
<tr>
<td>BSD Unix F77 1.0</td>
<td>Sun F77 1.3</td>
<td>8</td>
<td>1.2981</td>
<td>-0.4555</td>
<td>0.8987</td>
<td>0.001</td>
</tr>
<tr>
<td>BSD Unix F77 1.0</td>
<td>CRAY CFT77 4.01</td>
<td>9</td>
<td>1.6864</td>
<td>-0.7733</td>
<td>0.7376</td>
<td>0.015</td>
</tr>
<tr>
<td>BSD Unix F77 1.0</td>
<td>IBM XLF 1.1</td>
<td>7</td>
<td>0.8282</td>
<td>-0.3452</td>
<td>0.6892</td>
<td>0.040</td>
</tr>
<tr>
<td>BSD Unix F77 1.0</td>
<td>M88K F77 2.0b3</td>
<td>6</td>
<td>0.9912</td>
<td>-0.5479</td>
<td>0.9489</td>
<td>0.001</td>
</tr>
<tr>
<td>MIPS F77 2.0</td>
<td>Sun F77 1.3</td>
<td>7</td>
<td>0.7816</td>
<td>0.1064</td>
<td>0.7454</td>
<td>0.025</td>
</tr>
<tr>
<td>MIPS F77 2.0</td>
<td>CRAY CFT77 4.01</td>
<td>8</td>
<td>0.3829</td>
<td>0.2566</td>
<td>0.2699</td>
<td>0.300</td>
</tr>
<tr>
<td>MIPS F77 2.0</td>
<td>IBM XLF 1.1</td>
<td>5</td>
<td>0.4523</td>
<td>0.0086</td>
<td>0.6361</td>
<td>0.150</td>
</tr>
<tr>
<td>MIPS F77 2.0</td>
<td>M88K F77 2.0b3</td>
<td>6</td>
<td>0.5109</td>
<td>0.4106</td>
<td>0.6899</td>
<td>0.070</td>
</tr>
<tr>
<td>Sun F77 1.3</td>
<td>CRAY CFT77 4.01</td>
<td>8</td>
<td>1.0224</td>
<td>-0.0558</td>
<td>0.6864</td>
<td>0.040</td>
</tr>
<tr>
<td>Sun F77 1.3</td>
<td>IBM XLF 1.1</td>
<td>5</td>
<td>0.6152</td>
<td>-0.0686</td>
<td>0.9510</td>
<td>0.007</td>
</tr>
<tr>
<td>Sun F77 1.3</td>
<td>M88K F77 2.0b3</td>
<td>6</td>
<td>0.5805</td>
<td>0.3293</td>
<td>0.8325</td>
<td>0.020</td>
</tr>
<tr>
<td>CRAY CFT77 4.01</td>
<td>IBM XLF 1.1</td>
<td>7</td>
<td>0.2220</td>
<td>0.1555</td>
<td>0.4608</td>
<td>0.170</td>
</tr>
<tr>
<td>CRAY CFT77 4.01</td>
<td>M88K F77 2.0b3</td>
<td>7</td>
<td>0.2855</td>
<td>0.5375</td>
<td>0.6329</td>
<td>0.070</td>
</tr>
<tr>
<td>IBM XLF 1.1</td>
<td>M88K F77 2.0b3</td>
<td>5</td>
<td>0.5764</td>
<td>0.5298</td>
<td>0.6322</td>
<td>0.150</td>
</tr>
</tbody>
</table>

5. Compiler Optimization and Benchmarks

In this section we discuss the main optimizations present in the SPEC benchmarks. We focus on those basic blocks in the source code that account for the bulk of the execution time, since they provide most of the potential for useful optimization. We consider both the optimizations that current compilers can detect and those that might be found with better optimizers.

5.1. Amount of Optimization in the SPEC Fortran Benchmarks

5.1.1. DODUC

This program, along with FPPPP, are the two SPEC programs showing the least optimization improvement in table 6. In these programs the most important basic blocks consist of a large sequence of scalar arithmetic expressions with very little reuse. In DODUC most of the computation does not reside in loops, so optimizations like strength reduction, address collapsing, code motion, and register allocation, which are effective in other scientific programs, are not profitable here. Furthermore, the execution time of the program is not determined by a few basic blocks, so optimizers are forced to do a good job on the whole program to significantly reduce the execution time. Therefore DODUC can be considered a good challenge to any optimizer, and it is not surprising that its SPECRatio on almost all machines is consistently lower than the overall SPECmark even on those machines with very good floating point performance [SPEC90, 91a, 91b].

DODUC does have some opportunities for optimization that are not addressed by current compilers. Consider the procedure below, which is one of the most time-consuming basic blocks and accounts on the average for almost 5 percent of the execution time.

---

6 The contribution of a basic block to the total time varies from machine to machine, thus the 5 percent represents only an approximation.
SUBROUTINE X2Y2 (X, Y)
DOUBLE PRECISION X(21), Y(21)
REAL XX(21), YY(21)
DATA XX /0., 200., 400., 600., 800., 1000., 1500., 2000.,
2500., 3000., 3500., 4000., 6000., 8000., 10000.,
15000., 20000., 25000., 30000., 40000., 50000. /
DATA YY /0., 69.31, 120.68, 166.92, 210.12, 251.19, 347.43,
437.34, 522.82, 604.92, 684.31, 761.46, 1053.22,
1325.78, 1584.89, 2192.16, 2759.46, 3298.77, 3816.78,
4804.35, 5743.49 /
DO 1 I = 1, 21
X(I) = XX(I)
Y(I) = YY(I)
1 CONTINUE
RETURN
END

The code clearly shows that XX and YY are constant vectors and that the purpose of this subroutine is to copy their values into arrays X and Y. This procedure is executed almost 150,000 times. What is surprising is that vectors X and Y remain constant for the whole execution, that is, their values are never redefined outside this procedure; obviously there is no need to call the procedure more than once. This can be easily detected by inlining the subroutine at its only call site, where data flow analysis will detect that X and Y can be safely replaced by XX and YY. Doing this will reduce the execution time by around 5 percent. To detect this possible optimization requires procedure inlining and extending copy and constant propagation optimization to include vectors as well as simple variables.

5.1.2. FPPPP

As we mentioned above, optimizing FPPPP is quite difficult, because of the structure and size of its basic blocks. Its most time-consuming block contains 3000 floating point operations without a single branch, and accounts for approximately 40 percent of the execution time. The block uses 653 different variables, so achieving a good allocation of variables to registers is the most important optimization problem here. Furthermore, on the average a variable is referenced only 6 times, with around 161 intervening references between two consecutive uses of the same variable. For this block, a register set with an unlimited number of registers would eliminate close to 82% of all loads and stores\(^7\). For comparison, the MIPS compiler, at the maximum optimization level, is capable of eliminating only 22% of the loads and stores, while Sun's optimizer eliminates only 11%. There other basic blocks in the program with similar characteristics.

Given that this program's performance is strongly determined by the scalar floating point performance of the machine, the only way to significantly improve its performance is by reducing the execution time of floating point operations, by increasing the size of the register file, or by improving the register allocation algorithm [Chai82, Chow84]\(^8\).

---

\(^7\) It is evident that a register file having at least as many registers as variables in a basic block makes it possible to generate the minimum number of loads and stores.

\(^8\) It should also be possible to also speed up the program by finding some way to do loads faster than issuing successive load instructions. For example, load multiple.
5.1.3. TOMCATV

This is a good benchmark for testing an optimizer, as its most time consuming loop contains ample opportunities for optimization. Some of these optimizations cannot be detected by many of today's best optimizing compilers. TOMCATV contains a couple of statements which serve no function at all, and which can be eliminated only if the optimizer implements the correct optimizations. The loop shown below is responsible for approximately 60% of the total execution time.

```
DO 250 I = I1P, I2M
   IP = I + 1
   IM = I - 1
   XX = X(IP,J) - X(IM,J)
   YY = Y(IP,J) - Y(IM,J)
   XY = X(I,JP) - X(I,JM)
   YX = Y(I,JP) - Y(I,JM)
   A  = 0.250 * (XX * XY + YY * YX)
   B  = 0.250 * (XX * XX + YX * YX)
   C  = 0.125 * (XX * XY + YY * YY)
   QI = 0.0
   QJ = 0.0
C   QI = A * 0.5
C   QJ = B * 0.5
   AA(I,M) = -B
   DD(I,M) = B + B + A * REL
   PXX = X(IP,J) - 2.0 * X(I,J) + X(IM,J)
   QQX = Y(IP,J) - 2.0 * Y(I,J) + Y(IM,J)
   PYY = X(I,JP) - 2.0 * X(I,J) + X(I,JM)
   QQY = Y(I,JP) - 2.0 * Y(I,J) + Y(I,JM)
   PXY = X(IP,JP) - X(IP,JM) - X(IM,JP) + X(IM,JM)
   QXY = Y(IP,JP) - Y(IP,JM) - Y(IM,JP) + Y(IM,JM)
   RX(I,M) = A * PXX + B * PYY - C * PXY + XX * QI + XY * QJ
   RY(I,M) = A * QQX + B * QQY - C * QXY + YY * QI + YY * QJ
250  CONTINUE
```

First, consider the two statements above the comments; not only can they be moved out of the loop, but if their constant values are propagated, the two rightmost subexpressions of the last two statements (XX * QI + XY * QJ and YX * QI + YY * QJ) can be eliminated, as they reduced to zero. This can be done, however, only if the optimizer implements floating point constant propagation. The results of §4.2 show that the Sun's Fortran compiler cannot detect this optimization. Our measurements indicate that if the Sun compilers were capable of eliminating the useless computations, then the execution time of TOMCATV on the Sparcstation 1+ would improve by 9 percent.

The most obvious way of optimizing this loop, which compilers can do, is to eliminate the address calculation of array elements. Once this is done, most of the improvement comes by eliminating as many loads and stores as possible. First, most elements of arrays X and Y are used twice in the loop, so they need to be loaded only once. Second, a good compiler may notice that all scalar variables are temporaries whose values do not need to be stored for the duration of the loop. After this we are still left with 18 loads and 4 stores per iteration. However, few optimizers can achieve this because of the limited number of registers available to them. For example, the MIPS Fortran compiler, which is one of the best compilers, cannot keep all temporaries in registers and is forced to make 26 loads and 10 stores per iteration. This is because the R2010 coprocessor has only 16 floating point registers.
If the machine has more than 16 floating point registers it can further eliminate loads and stores by using a novel optimization technique called predictive commoning [O’Br90]. The idea here is to identify a sequence of values used in an iteration which contains a subsequence which is reused in the next iteration as a successor of the same sequence. An example of this is sequence \( X(I, J), X(I, J'), \) and \( X(M, J') \), whose first two elements are reused in the next iteration. The optimization consists of eliminating all the loads of the reused values by moving them at the end of each iteration to the registers into which their successors would be loaded. In the code there are six such sequences, so we can eliminate 12 of the 18 loads. Although this introduces 12 register move instructions, these can be eliminated by unrolling the loop.

The SPECratio of the IBM RS/6000 on this benchmark is much higher than that of the other benchmarks; a factor of three with respect to the overall SPECmark. This is due to the exceptional ability of the compiler to detect most of the optimizations we described and to the 32 floating point registers in the machine. The IBM XL compiler is capable of eliminating most of the loads by reusing registers and applying predictive commoning.

5.1.4. MATRIX300

It has been documented that this benchmark is completely dominated by a single basic block, which accounts for 99% of the execution time [Saav90a, 90b, 92a]. This basic block implements the SAXPY vector to vector operation \( Y[1,1:N] = Y[1,1:N] + A \times X[1,1:N] \). This subroutine is used in the program to compute eight different variations of matrix multiplication, each representing a particular operation between the three matrices and their transposes. Because the distance between two elements is different depending on whether the matrix is traversed by column or by row, this makes SAXPY difficult to optimize as each time it is called using different strides. Furthermore, the combined size of the matrices is greater than 2MB, so they do not fit in any of the caches of current machines.

Although MATRIX300 is difficult to optimize, it can be done. The best way to achieve this is to incorporate the same compiler technology developed for supercomputers to generate vectorized code. The idea is to apply dependence analysis to identify loops that can either be decomposed into vector operations or parallel execution. The same technology, however, can also be used in scalar machines even if they do not have vector hardware. The idea here is for a preprocessor to generate, instead of vector instructions, subroutine calls to hand-coded routines. These routines can be highly optimized by taking into consideration the best scheduling and blocking (tiling) factor to produce substantial reductions in the execution time. Several high performance workstations are starting to use this approach: the new HP 700 series includes a preprocessor developed by Kuck and Associates to make a source to source transformation of the program which includes calls to library routines.

The latest SPEC results [SPEC91a, 91b] clearly indicate that manufacturers are now using preprocessors to dramatically improve performance for MATRIX300. Note that this approach is not entirely "fair" for benchmark purposes, and makes programs susceptible to this type of manipulation of questionable value for benchmarking purposes. In particular, this type of preprocessing changes MATRIX300 from a memory bound benchmark, which generates huge numbers of cache misses, into one which is CPU
bound; thus the preprocessing even changes what the benchmark attempts to measure.

5.1.5. NASA7

This benchmark consists of several computation intensive kernels which are frequently found in scientific applications. As most of the work is localized in highly-nested loops, most of the optimization improvements come from eliminating computation from the innermost loops by using array addressing, code motion and strength reduction optimizations.

As in other benchmarks based on kernels, this benchmark has some characteristics which can be exploited by a clever compiler. For example, the code excerpt below implements matrix multiply by doing a 4-way unrolling of the outer loop and accounts for approximately 16 percent of the execution time.

```fortran
SUBROUTINE MXM (A, B, C, L, M, N)
IMPLICIT DOUBLE PRECISION(A-H, O-Z)
DIMENSION A(L,M), B(M,N), C(L,N)

DO 100 K = 1, N
   DO 100 I = 1, L
      C(I,K) = 0.
   100 CONTINUE

DO 110 J = 1, M, 4
   DO 110 K = 1, N
      C(I,K) = C(I,K) + A(I,J) * B(J,K)
      + A(I,J+1) * B(J+1,K) + A(I,J+2) * B(J+2,K)
      + A(I,J+3) * B(J+3,K)
   110 CONTINUE

RETURN
END
```

This subroutine is called 100 times by the following loop:

```fortran
DO 120 II = 1, IT
   CALL MXM (A, B, C, L, M, N)
120 CONTINUE
```

It is possible for a good compiler to detect, after inlining the subroutine, that the code inside loop 120 is invariant with respect to the induction variable and hence it only has to be executed once.

5.1.6. Spice2g6

This scalar code is similar to DODUC and FPPPP. The few opportunities for optimization involve finding small common subexpressions and allocating frequently used variables to registers. Some of the most executed blocks are very small and do not contain much that can be optimized. The following spaghetti-like code, for example, accounts for almost 43% of the total execution and contains few opportunities for optimization.
135 IF (J .LT. I) GO TO 145
LOCIJ = LOCC
140 LOCIJ = NODPLC(IRPT+LOCIJ)
IF (NODPLC(IROWNO+LOCIJ) .EQ. I) GO TO 155
GO TO 140
145 LOCIJ = LOCR
150 LOCIJ = NODPLC(JCPT+LOCIJ)
IF (NODPLC(JCMLNO+LOCIJ) .EQ. J) GO TO 155
GO TO 150
155 VALUE(LVN+LOCIJ) = VALUE(LVN+LOCIJ) - VALUE(LVN+LOCC) * 
VALUE(LVN+LOCR)
.160 LOCC=NODPLC(JCPT+LOC)
GO TO 130

The small size of the basic blocks and the irregular way in which the array elements are accessed makes it difficult for the compiler to improve the code.

6. Conclusions

Evaluating and explaining the performance of a machine requires relating observed performance to the individual components of the system. Machine designers are able to do this by constructing detailed models and simulators of their machines [Peut77, Shus78, Cmel91]. These machine models, however, are machine-dependent and generally they can only be used for one machine. Our research has concentrated on developing a sound methodology for evaluating machines and compilers in a machine independent manner. We have created a machine independent model for program execution, measured its parameters, and demonstrated its ability to make accurate predictions.

In this paper we have discussed how optimization can be incorporated in our methodology and have shown that it is possible to evaluate different optimizing compilers, not only by detecting the set of optimizations which they can perform, but also by predicting and explaining how much improvement they provide on large applications. In earlier work [Saav89], we said that we did not expect our methodology to extend naturally to include optimization, because we believed that it would be necessary for us to know how an arbitrary optimizer could transform any possible program. Since that time, we have discovered that our abstract machine paradigm largely extends to optimized code. By assuming that most of the optimizations are invariant with respect to the abstract decomposition of the program, we change the nature of the problem from one of detecting how a program could be changed by the compiler to characterizing the performance of the 'optimized' machine defined by the optimizer. Using this approach we showed that it is possible to measure the contribution of optimization and predict the execution time of optimized programs, although not as well as in the nonoptimized case.

We have written programs to detect local and global machine-independent optimizations and measured several optimizing compilers. We showed that optimizing compilers differ in the effectiveness to which they can apply the same optimizations. We also evaluated the optimization improvement provided by several optimizers on the Fortran SPEC, Perfect Club, and other popular benchmarks. Finally, we discussed the main optimizations found in specific benchmarks and discussed some of the characteristics that could be exploited by clever compilers.
Acknowledgements

We would like to thank K. Stevens, Jr. for providing access to facilities at NASA Ames, as well as Jean Gascon and Ruby Lee from HP, and David E. Culler and Oscar Loureiro from U.C. Berkeley who let us run our programs on their machines.

Bibliography


Appendix A

### Group 1: Floating Point Arithmetic Operations (single, local)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>SRL</th>
<th>ARSL</th>
<th>MRLS</th>
<th>DRLS</th>
<th>ERLS</th>
<th>XRLS</th>
<th>TRSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>40</td>
<td>89</td>
<td>117</td>
<td>273</td>
<td>84</td>
<td>10173</td>
<td>180</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>40</td>
<td>47</td>
<td>108</td>
<td>198</td>
<td>75</td>
<td>10447</td>
<td>135</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>7</td>
<td>40</td>
<td>31</td>
<td>167</td>
<td>58</td>
<td>10426</td>
<td>20</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>40</td>
<td>159</td>
<td>258</td>
<td>583</td>
<td>316</td>
<td>26136</td>
<td>213</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>&lt;1</td>
<td>101</td>
<td>169</td>
<td>498</td>
<td>229</td>
<td>26068</td>
<td>113</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>&lt;1</td>
<td>95</td>
<td>145</td>
<td>395</td>
<td>219</td>
<td>26255</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>80</td>
<td>305</td>
<td>384</td>
<td>1856</td>
<td>2419</td>
<td>18414</td>
<td>510</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>75</td>
<td>298</td>
<td>322</td>
<td>1849</td>
<td>2398</td>
<td>18512</td>
<td>499</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>&lt;1</td>
<td>234</td>
<td>133</td>
<td>1444</td>
<td>2575</td>
<td>19603</td>
<td>35</td>
</tr>
</tbody>
</table>

### Group 2: Floating Point Arithmetic Operations (complex, local)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>SCSL</th>
<th>ACSL</th>
<th>MCSL</th>
<th>DCSL</th>
<th>ECSL</th>
<th>XCSL</th>
<th>TCSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>73</td>
<td>163</td>
<td>496</td>
<td>3603</td>
<td>1726</td>
<td>40659</td>
<td>276</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>67</td>
<td>116</td>
<td>251</td>
<td>3574</td>
<td>1727</td>
<td>40470</td>
<td>288</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>54</td>
<td>59</td>
<td>130</td>
<td>3528</td>
<td>1700</td>
<td>40525</td>
<td>224</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>22</td>
<td>568</td>
<td>1162</td>
<td>5059</td>
<td>4696</td>
<td>37344</td>
<td>339</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>16</td>
<td>305</td>
<td>795</td>
<td>5019</td>
<td>4672</td>
<td>36864</td>
<td>263</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>12</td>
<td>227</td>
<td>713</td>
<td>4926</td>
<td>4352</td>
<td>37249</td>
<td>161</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>247</td>
<td>2772</td>
<td>5106</td>
<td>7471</td>
<td>5065</td>
<td>71209</td>
<td>1094</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>142</td>
<td>588</td>
<td>3287</td>
<td>8073</td>
<td>2423</td>
<td>75121</td>
<td>998</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>20</td>
<td>523</td>
<td>2698</td>
<td>8314</td>
<td>2841</td>
<td>72563</td>
<td>649</td>
</tr>
</tbody>
</table>

### Group 3: Integer Arithmetic Operations (single, local)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>SISL</th>
<th>AISL</th>
<th>MISL</th>
<th>DISL</th>
<th>EISL</th>
<th>XISL</th>
<th>TISL</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>&lt;1</td>
<td>81</td>
<td>429</td>
<td>455</td>
<td>657</td>
<td>1603</td>
<td>162</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>&lt;1</td>
<td>48</td>
<td>419</td>
<td>439</td>
<td>640</td>
<td>1589</td>
<td>108</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>&lt;1</td>
<td>20</td>
<td>200</td>
<td>438</td>
<td>463</td>
<td>954</td>
<td>20</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>&lt;1</td>
<td>134</td>
<td>564</td>
<td>1551</td>
<td>662</td>
<td>1270</td>
<td>198</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>&lt;1</td>
<td>72</td>
<td>460</td>
<td>1506</td>
<td>715</td>
<td>1272</td>
<td>135</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>&lt;1</td>
<td>37</td>
<td>489</td>
<td>1685</td>
<td>497</td>
<td>1038</td>
<td>47</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>&lt;1</td>
<td>216</td>
<td>1104</td>
<td>2625</td>
<td>4645</td>
<td>5967</td>
<td>519</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>&lt;1</td>
<td>188</td>
<td>1265</td>
<td>2755</td>
<td>4501</td>
<td>5832</td>
<td>556</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>&lt;1</td>
<td>41</td>
<td>949</td>
<td>2625</td>
<td>4559</td>
<td>5876</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 13: Characterization results for Group 1-3 under different optimization levels. A value '<1' indicates that the parameter was not detected by the experiment.
### Group 4: Floating Point Arithmetic Operations (double, local)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>SRDL</th>
<th>ARDL</th>
<th>MRDL</th>
<th>DRDL</th>
<th>ERDL</th>
<th>XRDL</th>
<th>TRDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>40</td>
<td>89</td>
<td>117</td>
<td>313</td>
<td>106</td>
<td>9051</td>
<td>180</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>32</td>
<td>70</td>
<td>112</td>
<td>289</td>
<td>84</td>
<td>9118</td>
<td>140</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>7</td>
<td>40</td>
<td>31</td>
<td>202</td>
<td>59</td>
<td>9325</td>
<td>20</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>67</td>
<td>206</td>
<td>347</td>
<td>935</td>
<td>421</td>
<td>26040</td>
<td>305</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>26</td>
<td>112</td>
<td>220</td>
<td>808</td>
<td>295</td>
<td>26208</td>
<td>274</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>&lt; 1</td>
<td>101</td>
<td>172</td>
<td>643</td>
<td>302</td>
<td>26292</td>
<td>27</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>232</td>
<td>440</td>
<td>757</td>
<td>3336</td>
<td>3797</td>
<td>34902</td>
<td>1045</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>226</td>
<td>413</td>
<td>469</td>
<td>3339</td>
<td>3839</td>
<td>35032</td>
<td>1047</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>&lt; 1</td>
<td>234</td>
<td>253</td>
<td>2430</td>
<td>4243</td>
<td>37339</td>
<td>263</td>
</tr>
</tbody>
</table>

### Group 5: Floating Point Arithmetic Operations (single, global)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>SRSG</th>
<th>ARSG</th>
<th>MRSG</th>
<th>DRSG</th>
<th>ERSG</th>
<th>XRSG</th>
<th>TRSG</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>41</td>
<td>109</td>
<td>140</td>
<td>286</td>
<td>80</td>
<td>10202</td>
<td>167</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>20</td>
<td>59</td>
<td>85</td>
<td>205</td>
<td>94</td>
<td>10469</td>
<td>170</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>14</td>
<td>49</td>
<td>38</td>
<td>175</td>
<td>81</td>
<td>10384</td>
<td>32</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>40</td>
<td>192</td>
<td>261</td>
<td>625</td>
<td>307</td>
<td>26134</td>
<td>203</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>24</td>
<td>176</td>
<td>232</td>
<td>523</td>
<td>240</td>
<td>26036</td>
<td>108</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>&lt; 1</td>
<td>95</td>
<td>256</td>
<td>481</td>
<td>302</td>
<td>26650</td>
<td>19</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>95</td>
<td>333</td>
<td>427</td>
<td>1928</td>
<td>2391</td>
<td>18249</td>
<td>522</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>84</td>
<td>299</td>
<td>363</td>
<td>1850</td>
<td>2388</td>
<td>18511</td>
<td>419</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>&lt; 1</td>
<td>233</td>
<td>133</td>
<td>1432</td>
<td>2563</td>
<td>19517</td>
<td>37</td>
</tr>
</tbody>
</table>

### Group 6: Floating Point Arithmetic Operations (complex, global)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>SCSG</th>
<th>ACSG</th>
<th>MCSG</th>
<th>DCSG</th>
<th>ECSG</th>
<th>XCSG</th>
<th>TCSG</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>81</td>
<td>209</td>
<td>609</td>
<td>3624</td>
<td>3305</td>
<td>47213</td>
<td>272</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>67</td>
<td>120</td>
<td>293</td>
<td>3573</td>
<td>1719</td>
<td>41728</td>
<td>241</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>53</td>
<td>73</td>
<td>152</td>
<td>3548</td>
<td>1710</td>
<td>41541</td>
<td>210</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>185</td>
<td>652</td>
<td>1278</td>
<td>5172</td>
<td>4705</td>
<td>37529</td>
<td>430</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>152</td>
<td>374</td>
<td>858</td>
<td>5190</td>
<td>4528</td>
<td>36802</td>
<td>270</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>135</td>
<td>225</td>
<td>736</td>
<td>5129</td>
<td>4245</td>
<td>37490</td>
<td>166</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>241</td>
<td>2843</td>
<td>5217</td>
<td>7517</td>
<td>5083</td>
<td>71267</td>
<td>1143</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>190</td>
<td>612</td>
<td>3498</td>
<td>8095</td>
<td>2516</td>
<td>75239</td>
<td>989</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>19</td>
<td>586</td>
<td>2676</td>
<td>8397</td>
<td>2736</td>
<td>72942</td>
<td>977</td>
</tr>
</tbody>
</table>

**Table 14:** Characterization results for Group 4-6 under different optimization levels. A value ‘< 1’ indicates that the parameter was not detected by the experiment.
### Group 7: Integer Arithmetic Operations (single, global)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>SISG</th>
<th>AISG</th>
<th>MISG</th>
<th>DISG</th>
<th>EISG</th>
<th>XISG</th>
<th>TISG</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>&lt; 1</td>
<td>109</td>
<td>454</td>
<td>482</td>
<td>669</td>
<td>1619</td>
<td>161</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>&lt; 1</td>
<td>72</td>
<td>419</td>
<td>450</td>
<td>670</td>
<td>1636</td>
<td>90</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>&lt; 1</td>
<td>37</td>
<td>139</td>
<td>449</td>
<td>408</td>
<td>864</td>
<td>50</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>&lt; 1</td>
<td>197</td>
<td>584</td>
<td>1608</td>
<td>676</td>
<td>1281</td>
<td>240</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>&lt; 1</td>
<td>106</td>
<td>466</td>
<td>1671</td>
<td>694</td>
<td>1230</td>
<td>152</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>&lt; 1</td>
<td>101</td>
<td>489</td>
<td>1747</td>
<td>761</td>
<td>1352</td>
<td>97</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>20</td>
<td>278</td>
<td>1160</td>
<td>2688</td>
<td>4664</td>
<td>5998</td>
<td>516</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>19</td>
<td>188</td>
<td>1265</td>
<td>2755</td>
<td>5609</td>
<td>5830</td>
<td>492</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>&lt; 1</td>
<td>41</td>
<td>951</td>
<td>2611</td>
<td>4535</td>
<td>5864</td>
<td>40</td>
</tr>
</tbody>
</table>

### Group 8: Floating Point Arithmetic Operations (double, global)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>SRDG</th>
<th>ARDG</th>
<th>MRDG</th>
<th>DRDG</th>
<th>ERDG</th>
<th>XRDG</th>
<th>TRDG</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>40</td>
<td>110</td>
<td>141</td>
<td>326</td>
<td>80</td>
<td>9044</td>
<td>167</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>27</td>
<td>76</td>
<td>121</td>
<td>246</td>
<td>92</td>
<td>9338</td>
<td>166</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>13</td>
<td>49</td>
<td>39</td>
<td>210</td>
<td>83</td>
<td>9266</td>
<td>32</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>67</td>
<td>240</td>
<td>350</td>
<td>978</td>
<td>410</td>
<td>25942</td>
<td>312</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>58</td>
<td>221</td>
<td>339</td>
<td>846</td>
<td>341</td>
<td>26064</td>
<td>269</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>&lt; 1</td>
<td>133</td>
<td>381</td>
<td>782</td>
<td>376</td>
<td>27206</td>
<td>61</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>232</td>
<td>535</td>
<td>840</td>
<td>3475</td>
<td>3806</td>
<td>34717</td>
<td>1080</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>186</td>
<td>440</td>
<td>727</td>
<td>3332</td>
<td>6484</td>
<td>34990</td>
<td>1052</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>&lt; 1</td>
<td>233</td>
<td>255</td>
<td>2427</td>
<td>4280</td>
<td>37251</td>
<td>267</td>
</tr>
</tbody>
</table>

### Group 9, 10: Conditional and Logical Parameters

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>ANDL</th>
<th>CRSL</th>
<th>CCSL</th>
<th>CISL</th>
<th>CRDL</th>
<th>ANDG</th>
<th>CRSG</th>
<th>CCSSG</th>
<th>CISG</th>
<th>CRDG</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>87</td>
<td>229</td>
<td>491</td>
<td>94</td>
<td>229</td>
<td>97</td>
<td>270</td>
<td>579</td>
<td>136</td>
<td>269</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>72</td>
<td>168</td>
<td>336</td>
<td>61</td>
<td>175</td>
<td>81</td>
<td>175</td>
<td>337</td>
<td>68</td>
<td>175</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>46</td>
<td>114</td>
<td>282</td>
<td>20</td>
<td>115</td>
<td>46</td>
<td>148</td>
<td>298</td>
<td>47</td>
<td>148</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>109</td>
<td>283</td>
<td>281</td>
<td>202</td>
<td>365</td>
<td>135</td>
<td>365</td>
<td>366</td>
<td>283</td>
<td>447</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>105</td>
<td>302</td>
<td>319</td>
<td>208</td>
<td>413</td>
<td>132</td>
<td>418</td>
<td>345</td>
<td>308</td>
<td>518</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>70</td>
<td>206</td>
<td>207</td>
<td>95</td>
<td>204</td>
<td>97</td>
<td>231</td>
<td>340</td>
<td>95</td>
<td>221</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>224</td>
<td>582</td>
<td>1492</td>
<td>420</td>
<td>742</td>
<td>241</td>
<td>664</td>
<td>1572</td>
<td>503</td>
<td>914</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>182</td>
<td>595</td>
<td>1474</td>
<td>475</td>
<td>809</td>
<td>182</td>
<td>595</td>
<td>1470</td>
<td>476</td>
<td>807</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>79</td>
<td>365</td>
<td>1515</td>
<td>149</td>
<td>365</td>
<td>80</td>
<td>366</td>
<td>1518</td>
<td>149</td>
<td>366</td>
</tr>
</tbody>
</table>

Table 15: Characterization results for Group 7-10 under different optimization levels. A value ‘< 1’ indicates that the parameter was not detected by the experiment.
### Group 11, 12: Function Call, Arguments and References to Array Elements

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>PROC</th>
<th>ARGU</th>
<th>ARR1</th>
<th>ARR2</th>
<th>ARR3</th>
<th>IADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>81</td>
<td>153</td>
<td>64</td>
<td>217</td>
<td>320</td>
<td>6</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>85</td>
<td>122</td>
<td>30</td>
<td>130</td>
<td>187</td>
<td>9</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>79</td>
<td>50</td>
<td>47</td>
<td>28</td>
<td>50</td>
<td>&lt;1</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>408</td>
<td>80</td>
<td>309</td>
<td>612</td>
<td>936</td>
<td>&lt;1</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>283</td>
<td>145</td>
<td>150</td>
<td>380</td>
<td>583</td>
<td>&lt;1</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>152</td>
<td>141</td>
<td>122</td>
<td>184</td>
<td>191</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>196</td>
<td>328</td>
<td>224</td>
<td>637</td>
<td>1007</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>81</td>
<td>58</td>
<td>124</td>
<td>409</td>
<td>705</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>77</td>
<td>41</td>
<td>133</td>
<td>194</td>
<td>281</td>
<td>&lt;1</td>
</tr>
</tbody>
</table>

### Group 13, 14: Branching and DO loop Parameters

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>GOTO</th>
<th>GCOM</th>
<th>LOIN</th>
<th>LOOV</th>
<th>LOIX</th>
<th>LLOOK</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>28</td>
<td>243</td>
<td>363</td>
<td>202</td>
<td>552</td>
<td>303</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>20</td>
<td>263</td>
<td>242</td>
<td>121</td>
<td>447</td>
<td>161</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>12</td>
<td>172</td>
<td>68</td>
<td>42</td>
<td>182</td>
<td>61</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>81</td>
<td>609</td>
<td>580</td>
<td>322</td>
<td>838</td>
<td>604</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>&lt;1</td>
<td>648</td>
<td>325</td>
<td>201</td>
<td>445</td>
<td>354</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>&lt;1</td>
<td>486</td>
<td>1174</td>
<td>&lt;1</td>
<td>284</td>
<td>202</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>&lt;1</td>
<td>894</td>
<td>934</td>
<td>404</td>
<td>1458</td>
<td>662</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>&lt;1</td>
<td>853</td>
<td>220</td>
<td>114</td>
<td>467</td>
<td>280</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>&lt;1</td>
<td>650</td>
<td>161</td>
<td>81</td>
<td>406</td>
<td>244</td>
</tr>
</tbody>
</table>

### Group 15: Intrinsic Functions (single precision)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>EXPS</th>
<th>LOGS</th>
<th>SINS</th>
<th>TANS</th>
<th>SQRS</th>
<th>ABSS</th>
<th>MODS</th>
<th>MAXS</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>2780</td>
<td>3457</td>
<td>1878</td>
<td>3037</td>
<td>292</td>
<td>61</td>
<td>3921</td>
<td>424</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>2725</td>
<td>3396</td>
<td>1787</td>
<td>2927</td>
<td>283</td>
<td>61</td>
<td>3821</td>
<td>284</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>2701</td>
<td>3376</td>
<td>1787</td>
<td>2913</td>
<td>304</td>
<td>61</td>
<td>3782</td>
<td>323</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>4036</td>
<td>3323</td>
<td>3019</td>
<td>3657</td>
<td>3570</td>
<td>35</td>
<td>2232</td>
<td>434</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>3937</td>
<td>3280</td>
<td>2966</td>
<td>3609</td>
<td>3567</td>
<td>81</td>
<td>2118</td>
<td>449</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>4139</td>
<td>3482</td>
<td>3168</td>
<td>3793</td>
<td>3726</td>
<td>40</td>
<td>2120</td>
<td>407</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>5563</td>
<td>5996</td>
<td>9085</td>
<td>12441</td>
<td>7891</td>
<td>453</td>
<td>3673</td>
<td>3333</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>5728</td>
<td>6162</td>
<td>9266</td>
<td>12626</td>
<td>9083</td>
<td>539</td>
<td>3619</td>
<td>1204</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>6805</td>
<td>7183</td>
<td>10299</td>
<td>13777</td>
<td>9140</td>
<td>611</td>
<td>4206</td>
<td>1182</td>
</tr>
</tbody>
</table>

**Table 16:** Characterization results for Group 11-15 under different optimization levels. A value '<1' indicates that the parameter was not detected by the experiment.
### Group 16: Intrinsic Functions (double precision)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>EXPD</th>
<th>LOGD</th>
<th>SIND</th>
<th>TAND</th>
<th>SQRD</th>
<th>ABSD</th>
<th>MODD</th>
<th>MAXD</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>3142</td>
<td>3129</td>
<td>2418</td>
<td>3851</td>
<td>363</td>
<td>64</td>
<td>3813</td>
<td>429</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>3085</td>
<td>3071</td>
<td>2359</td>
<td>3795</td>
<td>365</td>
<td>61</td>
<td>3697</td>
<td>284</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>3066</td>
<td>3051</td>
<td>2340</td>
<td>3780</td>
<td>384</td>
<td>61</td>
<td>3652</td>
<td>324</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>3586</td>
<td>4156</td>
<td>4327</td>
<td>4702</td>
<td>5519</td>
<td>41</td>
<td>2244</td>
<td>495</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>3567</td>
<td>4205</td>
<td>4296</td>
<td>4718</td>
<td>5622</td>
<td>16</td>
<td>2263</td>
<td>531</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>3607</td>
<td>4180</td>
<td>4336</td>
<td>4700</td>
<td>5537</td>
<td>35</td>
<td>2252</td>
<td>408</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>9048</td>
<td>10881</td>
<td>13025</td>
<td>16035</td>
<td>18787</td>
<td>624</td>
<td>4696</td>
<td>5373</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>9067</td>
<td>10944</td>
<td>13131</td>
<td>16005</td>
<td>18734</td>
<td>847</td>
<td>4610</td>
<td>1169</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>10408</td>
<td>12175</td>
<td>14106</td>
<td>17065</td>
<td>19885</td>
<td>892</td>
<td>5967</td>
<td>1115</td>
</tr>
</tbody>
</table>

### Groups 17, 18: Intrinsic Functions (integer and complex)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>ABSI</th>
<th>MODI</th>
<th>MAXI</th>
<th>EXPC</th>
<th>LOGC</th>
<th>SINC</th>
<th>SQRC</th>
<th>ABSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>40</td>
<td>1668</td>
<td>161</td>
<td>17520</td>
<td>12060</td>
<td>30537</td>
<td>9434</td>
<td>5386</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>&lt;1</td>
<td>1630</td>
<td>80</td>
<td>17488</td>
<td>12024</td>
<td>30514</td>
<td>9406</td>
<td>5400</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>40</td>
<td>1649</td>
<td>81</td>
<td>17555</td>
<td>12059</td>
<td>30694</td>
<td>9434</td>
<td>5375</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>91</td>
<td>1522</td>
<td>349</td>
<td>13835</td>
<td>10607</td>
<td>13559</td>
<td>16574</td>
<td>4253</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>83</td>
<td>1400</td>
<td>203</td>
<td>13671</td>
<td>10178</td>
<td>13276</td>
<td>9580</td>
<td>4000</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>94</td>
<td>1579</td>
<td>243</td>
<td>13774</td>
<td>10289</td>
<td>13401</td>
<td>9562</td>
<td>3765</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>982</td>
<td>2229</td>
<td>2956</td>
<td>41051</td>
<td>22567</td>
<td>59548</td>
<td>46154</td>
<td>23767</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>314</td>
<td>2220</td>
<td>657</td>
<td>40561</td>
<td>21951</td>
<td>59170</td>
<td>45925</td>
<td>22615</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>189</td>
<td>2189</td>
<td>405</td>
<td>40533</td>
<td>23827</td>
<td>59139</td>
<td>45487</td>
<td>22478</td>
</tr>
</tbody>
</table>

### Groups 19: Intrinsic Functions (type conversion)

<table>
<thead>
<tr>
<th>machine</th>
<th>optim</th>
<th>CPLX</th>
<th>REAL</th>
<th>IMAG</th>
<th>CONJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 720</td>
<td>-O0</td>
<td>&lt;1</td>
<td>20</td>
<td>20</td>
<td>61</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O1</td>
<td>5</td>
<td>&lt;1</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>HP 720</td>
<td>-O2</td>
<td>40</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>62</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O0</td>
<td>179</td>
<td>&lt;1</td>
<td>824</td>
<td>445</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O1</td>
<td>164</td>
<td>&lt;1</td>
<td>918</td>
<td>711</td>
</tr>
<tr>
<td>MIPS M/2000</td>
<td>-O2</td>
<td>40</td>
<td>&lt;1</td>
<td>649</td>
<td>611</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O0</td>
<td>782</td>
<td>1522</td>
<td>2262</td>
<td>1482</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O2</td>
<td>161</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>111</td>
</tr>
<tr>
<td>Sparcstation 1+</td>
<td>-O3</td>
<td>567</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>144</td>
</tr>
</tbody>
</table>

Table 17: Characterization results for Group 11-19 under different optimization levels. A value '<1' indicates that the parameter was not detected by the experiment.
Appendix B

<table>
<thead>
<tr>
<th>Program</th>
<th>Optimization level 0</th>
<th>Optimization level 1</th>
<th>Optimization level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real (sec)</td>
<td>pred (sec)</td>
<td>error (%)</td>
</tr>
<tr>
<td>Doduc</td>
<td>85</td>
<td>85</td>
<td>-0.12</td>
</tr>
<tr>
<td>Fpppp</td>
<td>78</td>
<td>90</td>
<td>+15.30</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>182</td>
<td>157</td>
<td>-14.09</td>
</tr>
<tr>
<td>Matrix300</td>
<td>598</td>
<td>340</td>
<td>-43.13</td>
</tr>
<tr>
<td>Nasa7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Spice2g6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>QCD</td>
<td>81</td>
<td>108</td>
<td>+33.21</td>
</tr>
<tr>
<td>MDG</td>
<td>827</td>
<td>861</td>
<td>+4.04</td>
</tr>
<tr>
<td>BDNA</td>
<td>208</td>
<td>185</td>
<td>-11.32</td>
</tr>
<tr>
<td>OCEAN</td>
<td>764</td>
<td>791</td>
<td>+3.52</td>
</tr>
<tr>
<td>MG3D</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ARC2D</td>
<td>2137</td>
<td>1267</td>
<td>-40.70</td>
</tr>
<tr>
<td>TRFD</td>
<td>464</td>
<td>329</td>
<td>-28.98</td>
</tr>
<tr>
<td>FLO52</td>
<td>426</td>
<td>274</td>
<td>-35.70</td>
</tr>
<tr>
<td>Alamos</td>
<td>67</td>
<td>67</td>
<td>+0.45</td>
</tr>
<tr>
<td>Baskett</td>
<td>0.50</td>
<td>0.58</td>
<td>+16.00</td>
</tr>
<tr>
<td>Eratosthenes</td>
<td>0.13</td>
<td>0.13</td>
<td>+1.54</td>
</tr>
<tr>
<td>Linpack</td>
<td>8.8</td>
<td>7.6</td>
<td>-13.67</td>
</tr>
<tr>
<td>Livermore</td>
<td>16.5</td>
<td>17.7</td>
<td>+7.27</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>0.89</td>
<td>0.89</td>
<td>0.00</td>
</tr>
<tr>
<td>Shell</td>
<td>0.49</td>
<td>0.42</td>
<td>-13.67</td>
</tr>
<tr>
<td>Smith</td>
<td>54</td>
<td>47</td>
<td>-12.15</td>
</tr>
<tr>
<td>Whetstone</td>
<td>0.28</td>
<td>0.26</td>
<td>-7.14</td>
</tr>
</tbody>
</table>

Table 18: Nonoptimized and optimized benchmark results on the HP 720. All times are reported in seconds and the errors are computed as $100 \times (pred - real)/real$. 
<table>
<thead>
<tr>
<th>program</th>
<th>Optimization level 0</th>
<th>Optimization level 1</th>
<th>Optimization level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real (sec)</td>
<td>pred (sec)</td>
<td>error (%)</td>
</tr>
<tr>
<td>Doduc</td>
<td>187</td>
<td>208</td>
<td>+11.69</td>
</tr>
<tr>
<td>Fpppp</td>
<td>247</td>
<td>239</td>
<td>-8.14</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>452</td>
<td>415</td>
<td>-3.21</td>
</tr>
<tr>
<td>Matrix300</td>
<td>816</td>
<td>614</td>
<td>-24.77</td>
</tr>
<tr>
<td>Nasa7</td>
<td>2906</td>
<td>2634</td>
<td>-9.36</td>
</tr>
<tr>
<td>Spice2g6</td>
<td>4576</td>
<td>4539</td>
<td>-0.81</td>
</tr>
<tr>
<td>ADM</td>
<td>424</td>
<td>426</td>
<td>+0.47</td>
</tr>
<tr>
<td>QCD</td>
<td>131</td>
<td>176</td>
<td>+34.48</td>
</tr>
<tr>
<td>MDG</td>
<td>1796</td>
<td>2254</td>
<td>+25.50</td>
</tr>
<tr>
<td>TRACK</td>
<td>58</td>
<td>71</td>
<td>+22.20</td>
</tr>
<tr>
<td>BDNA</td>
<td>733</td>
<td>582</td>
<td>-20.60</td>
</tr>
<tr>
<td>OCEAN</td>
<td>1618</td>
<td>1722</td>
<td>+6.47</td>
</tr>
<tr>
<td>DYFESM</td>
<td>407</td>
<td>370</td>
<td>-9.10</td>
</tr>
<tr>
<td>ARC2D</td>
<td>3465</td>
<td>2470</td>
<td>-28.70</td>
</tr>
<tr>
<td>TRFD</td>
<td>577</td>
<td>566</td>
<td>-1.87</td>
</tr>
<tr>
<td>FLO52</td>
<td>721</td>
<td>853</td>
<td>-15.52</td>
</tr>
<tr>
<td>Alamos</td>
<td>118</td>
<td>139</td>
<td>+16.95</td>
</tr>
<tr>
<td>Basket</td>
<td>1.00</td>
<td>1.13</td>
<td>+13.00</td>
</tr>
<tr>
<td>Erathostenes</td>
<td>0.47</td>
<td>0.31</td>
<td>-21.28</td>
</tr>
<tr>
<td>Linpack</td>
<td>12.7</td>
<td>14.5</td>
<td>+13.94</td>
</tr>
<tr>
<td>Livermore</td>
<td>30.0</td>
<td>38.6</td>
<td>+28.80</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>1.50</td>
<td>1.59</td>
<td>+6.00</td>
</tr>
<tr>
<td>Shell</td>
<td>1.64</td>
<td>1.59</td>
<td>-2.44</td>
</tr>
<tr>
<td>Smith</td>
<td>132</td>
<td>113</td>
<td>-15.05</td>
</tr>
<tr>
<td>Whetstone</td>
<td>0.48</td>
<td>0.48</td>
<td>+0.00</td>
</tr>
</tbody>
</table>

Table 19: Nonoptimized and optimized benchmark results on the MIPS M/2000 using the f77 compiler version 1.21. All times are reported in seconds and the errors are computed as 100×(pred−real)/real.
<table>
<thead>
<tr>
<th>program</th>
<th>Optimization level 0</th>
<th>Optimization level 2</th>
<th>Optimization level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real (sec)</td>
<td>pred (sec)</td>
<td>error (%)</td>
</tr>
<tr>
<td>Doduc</td>
<td>407</td>
<td>410</td>
<td>+0.74</td>
</tr>
<tr>
<td>Fpppp</td>
<td>590</td>
<td>481</td>
<td>+10.63</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>576</td>
<td>637</td>
<td>-18.46</td>
</tr>
<tr>
<td>Matrix300</td>
<td>1502</td>
<td>815</td>
<td>-45.70</td>
</tr>
<tr>
<td>Nasa7</td>
<td>5516</td>
<td>4046</td>
<td>-26.66</td>
</tr>
<tr>
<td>Spice2g6</td>
<td>5348</td>
<td>6660</td>
<td>+24.54</td>
</tr>
<tr>
<td>ADM</td>
<td>716</td>
<td>551</td>
<td>-23.05</td>
</tr>
<tr>
<td>QCD</td>
<td>274</td>
<td>341</td>
<td>+24.47</td>
</tr>
<tr>
<td>MDG</td>
<td>3809</td>
<td>4072</td>
<td>+6.90</td>
</tr>
<tr>
<td>TRACK</td>
<td>108</td>
<td>122</td>
<td>+13.30</td>
</tr>
<tr>
<td>BDNA</td>
<td>1270</td>
<td>1173</td>
<td>-7.63</td>
</tr>
<tr>
<td>OCEAN</td>
<td>3538</td>
<td>3551</td>
<td>+0.37</td>
</tr>
<tr>
<td>DYFESM</td>
<td>617</td>
<td>437.1</td>
<td>-29.10</td>
</tr>
<tr>
<td>ARC2D</td>
<td>6437</td>
<td>4091</td>
<td>-36.44</td>
</tr>
<tr>
<td>TRFD</td>
<td>1181</td>
<td>765</td>
<td>-35.26</td>
</tr>
<tr>
<td>FLO52</td>
<td>1132</td>
<td>1049</td>
<td>-7.28</td>
</tr>
<tr>
<td>Alamos</td>
<td>207</td>
<td>188</td>
<td>-9.26</td>
</tr>
<tr>
<td>Basket</td>
<td>1.40</td>
<td>1.68</td>
<td>+20.00</td>
</tr>
<tr>
<td>Erathostenes</td>
<td>0.50</td>
<td>0.38</td>
<td>-24.00</td>
</tr>
<tr>
<td>Linpack</td>
<td>35.9</td>
<td>21.7</td>
<td>-19.03</td>
</tr>
<tr>
<td>Livermore</td>
<td>52.6</td>
<td>53.5</td>
<td>+1.17</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>2.40</td>
<td>2.70</td>
<td>+12.50</td>
</tr>
<tr>
<td>Shell</td>
<td>0.95</td>
<td>1.28</td>
<td>+34.74</td>
</tr>
<tr>
<td>Smith</td>
<td>198</td>
<td>144</td>
<td>-27.22</td>
</tr>
<tr>
<td>Whetstone</td>
<td>1.30</td>
<td>0.86</td>
<td>-28.33</td>
</tr>
</tbody>
</table>

Table 20: Nonoptimized and optimized benchmark results on the Sparcstation 1+ using the f77 compiler version 1.3. All times are reported in seconds and the errors are computed as \(100\times(pred - real)/real\).
Appendix C

<table>
<thead>
<tr>
<th>compiler</th>
<th>Integer</th>
<th></th>
<th>Floating Point</th>
<th></th>
<th>Mixed</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>local</td>
<td>global</td>
<td>local</td>
<td>global</td>
<td>local</td>
<td>global</td>
</tr>
<tr>
<td>BSD Unix F77 1.0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Mips F77 2.0 -02</td>
<td>yes</td>
<td>yes</td>
<td>partial¹</td>
<td>partial¹</td>
<td>partial¹</td>
<td>no</td>
</tr>
<tr>
<td>Mips F77 2.0 -01</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Sun F77 1.3 -03</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Sun F77 1.3 -02</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Sun F77 1.3 -01</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Ultrix Fort 4.5</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Amdahl F77 2.0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>CRAY CFT77 4.0.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IBM XL Fortran 1.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Motorola F77 2.0b3</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

1 variables assigned to constant expressions are not propagated

Table 21: Optimization results for constant folding (local and global).

<table>
<thead>
<tr>
<th>compiler</th>
<th>Integer</th>
<th></th>
<th>Floating Point</th>
<th></th>
<th>Mixed</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>local</td>
<td>global</td>
<td>local</td>
<td>global</td>
<td>local</td>
<td>global</td>
</tr>
<tr>
<td>BSD Unix F77 1.0</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Mips F77 2.0 -02</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Mips F77 2.0 -01</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sun F77 1.3 -03</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sun F77 1.3 -02</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sun F77 1.3 -01</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Ultrix Fort 4.5</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Amdahl F77 2.0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>CRAY CFT77 4.0.1</td>
<td>yes</td>
<td>partial²</td>
<td>yes</td>
<td>partial²</td>
<td>yes</td>
<td>partial²</td>
</tr>
<tr>
<td>IBM XL Fortran 1.1</td>
<td>yes</td>
<td>yes</td>
<td>partial³</td>
<td>partial³</td>
<td>partial³</td>
<td>partial³</td>
</tr>
<tr>
<td>Motorola F77 2.0b3</td>
<td>yes</td>
<td>partial²</td>
<td>yes</td>
<td>partial²</td>
<td>yes</td>
<td>partial²</td>
</tr>
</tbody>
</table>

1 not all common subexpressions are recognized
2 incomplete global analysis is not enough to detect all optimizations
3 transformations to the intermediate code destroy some common subexpressions

Table 22: Optimization results for common subexpression elimination (local and global).
<table>
<thead>
<tr>
<th>compiler</th>
<th>Integer</th>
<th>Floating Point</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>local</td>
<td>global</td>
<td>local</td>
</tr>
<tr>
<td>BSD Unix F77 1.0</td>
<td>partial&lt;sup&gt;1&lt;/sup&gt;</td>
<td>partial&lt;sup&gt;1&lt;/sup&gt;</td>
<td>no</td>
</tr>
<tr>
<td>Mips F77 2.0 -O2</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Mips F77 2.0 -O1</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Sun F77 1.3 -O3</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sun F77 1.3 -O2</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sun F77 1.3 -O1</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Ultrix Fort 4.5</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Amdahl F77 2.0</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>CRAY CFT77 4.0.1</td>
<td>yes</td>
<td>partial&lt;sup&gt;3&lt;/sup&gt;</td>
<td>yes</td>
</tr>
<tr>
<td>IBM XL Fortran 1.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Motorola F77 2.0b3</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

1 Only simple expressions are moved (<var> <op> <var>)
2 Only simple integers expressions
3 Blocks inside the loop are not considered

Table 23: Optimization results for code motion (local and global).

<table>
<thead>
<tr>
<th>compiler</th>
<th>Integer</th>
<th>Floating Point</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>local</td>
<td>global</td>
<td>local</td>
</tr>
<tr>
<td>BSD Unix F77 1.0</td>
<td>partial&lt;sup&gt;1&lt;/sup&gt;</td>
<td>no</td>
<td>partial&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>Mips F77 2.0 -O2</td>
<td>partial&lt;sup&gt;2&lt;/sup&gt;</td>
<td>marginal&lt;sup&gt;1&lt;/sup&gt;</td>
<td>partial&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Mips F77 2.0 -O1</td>
<td>marginal&lt;sup&gt;1&lt;/sup&gt;</td>
<td>no</td>
<td>marginal&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>Sun F77 1.3 -O3</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Sun F77 1.3 -O2</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Sun F77 1.3 -O1</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Ultrix Fort 4.5</td>
<td>yes</td>
<td>partial&lt;sup&gt;3&lt;/sup&gt;</td>
<td>yes</td>
</tr>
<tr>
<td>Amdahl F77 2.0</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>CRAY CFT77 4.0.1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>IBM XL Fortran 1.1</td>
<td>partial&lt;sup&gt;4&lt;/sup&gt;</td>
<td>marginal&lt;sup&gt;1&lt;/sup&gt;</td>
<td>partial&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
<tr>
<td>Motorola F77 2.0b3</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

1 Propagates only simple assignments (<var> = <var>)
2 Compiler has a limited lookahead
3 Incomplete global analysis is not enough to detect all optimizations
4 Transformations to the intermediate code destroy some common subexpressions

Table 24: Optimization results for copy propagation (local and global).
<table>
<thead>
<tr>
<th>compiler</th>
<th>Integer</th>
<th>Floating Point</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>local</td>
<td>global</td>
<td>local</td>
</tr>
<tr>
<td>BSD Unix F77 1.0</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Mips F77 2.0 -O2</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Mips F77 2.0 -O1</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Sun F77 1.3 -O3</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sun F77 1.3 -O2</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Sun F77 1.3 -O1</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Ultrix Fort 4.5</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Amdahl F77 2.0</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>CRAY CFT77 4.0.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IBM XL Fortran 1.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Motorola F77 2.0b3</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 25: Optimization results for dead code elimination (local and global).

<table>
<thead>
<tr>
<th>compiler</th>
<th>strength reduction</th>
<th>address calculation</th>
<th>inline substitution apply affects</th>
<th>loop unrolling</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSD Unix F77 1.0</td>
<td>partial¹</td>
<td>marginal²</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Mips F77 2.0 -O2</td>
<td>yes</td>
<td>yes</td>
<td>partial²</td>
<td>no</td>
</tr>
<tr>
<td>Mips F77 2.0 -O1</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes⁴</td>
</tr>
<tr>
<td>Sun F77 1.3 -O3</td>
<td>partial¹</td>
<td>marginal¹</td>
<td>no</td>
<td>yes⁴</td>
</tr>
<tr>
<td>Sun F77 1.3 -O2</td>
<td>partial¹</td>
<td>marginal¹</td>
<td>no</td>
<td>yes⁴</td>
</tr>
<tr>
<td>Sun F77 1.3 -O1</td>
<td>partial¹</td>
<td>no</td>
<td>no</td>
<td>yes⁴</td>
</tr>
<tr>
<td>Ultrix Fort 4.5</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Amdahl F77 2.0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>CRAY CFT77 4.0.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes⁴</td>
</tr>
<tr>
<td>IBM XL Fortran 1.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no³</td>
</tr>
<tr>
<td>Motorola F77 2.0b3</td>
<td>partial¹</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

1 Optimization is partially applied  
2 There is no code substitution; caller and callee use same stack frame  
3 There is code substitution; extra code obscures optimization  
4 Arbitrary unrolling of loops

Table 26: Optimization results for strength reduction, address calculation, inline substitution, and loop unrolling.