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

† The material presented here is based on research supported principally by NASA under grant NCC2-550, and also in part by the National Science Foundation under grants MIP-8713274, MIP-9116578, and CCR-9117028, by the State of California under the MICRO program, and by the International Business Machines Corporation, Philips Laboratories/Signetics, Apple Computer Corporation, Intel Corporation, Digital Equipment Corporation, Mitsubishi and Sun Microsystems.
§ This paper has been issued as technical report USC-CS-92-525 of the Computer Science Department at USC and technical report UCB-CSD-92-699 of the Computer Science Division, UC Berkeley, August 15, 1992.
‡ Computer Science Department, Henry Salvatori Building, University of Southern California, Los Angeles, California 90089-0781.
# Computer Science Division, EECS Department, University of California, Berkeley, California 94720.

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[2]. 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. Richarson 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 prescence 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 [Peut77].

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:

$$T_{A,M} = \sum_{i=1}^{n} C_{i,A} P_{i,M} = \mathbf{C_A} \cdot \mathbf{P_M}. \quad (1)$$

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 ($\mathbf{C_A}$) and the performance of the abstract operations ($\mathbf{P_M}$) may change. $\mathbf{C_A}$ changes when the optimizer eliminates some part of the computation. The raw performance measurements represented by $\mathbf{P_M}$ 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

$$T_{A,M,O} = \sum_{i=1}^{n} C_{i,A,O} P_{i,M,O} = \mathbf{C_{A,O}} \cdot \mathbf{P_{M,O}} \quad (2)$$

Our problem here is to obtain $\mathbf{C_{A,O}}$ and $\mathbf{P_{M,O}}$ 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 $\mathbf{C_A}$, but without affecting $\mathbf{P_M}$. The difficulty in characterizing the performance improvement due to these optimizations is that we need to know how $\mathbf{C_A}$ 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 $\mathbf{P_M}$, while leaving $\mathbf{C_A}$ 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

```
        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 (MRSL), 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

$$Time = (P_{SRSL} + 2 \cdot P_{ARR2} + 2 \cdot P_{ARR1} + P_{MRSL} + P_{ARSL} + P_{LOOV}) N^2 + (P_{LOIN} + P_{LOOV}) N + P_{LOIN}.$$

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

abstract operation	assembler code without optimization	assembler code with optimization
arr2	loadi r14, 80444(sp) loadi r15, 40036(sp) mul_i r24, r15, 100 add_i r25, r14, r24 sub_i r8, r25, 101 mul_i r9, r8, 4 add_i r10, r9, -40424 add_i r11, sp, 80464 add_i r12, r10, r11 loadf f6, 0(r12)	loadf f4, 24708(r3) add_i r3, r3, 4
arr2	loadi r13, 32(sp) mul_i r15, r13, 100 add_i r24, r14, r15 sub_i r25, r24, 101 mul_i r8, r25, 4 add_i r9, r8, -80428 add_i r10, r9, r11 loadf f8, 0(r10)	loadf f6, 17472(r4) add_i r4, r4, 4
mrsl	mul_f f10, f6, f8	mul_f f8, f4, f6
arr1	sub_i r12, r14, 1 mul_f r13, r12, 4 add_i r15, r13, -424 add_i r24, sp, 80464 add_i r25, r15, r24 loadf f16, 0(r25)	loadf f16, -428(r2)
arsl	add_f f18, f16, f10	add_f f10, f16, f8
srsl	storf f18, 0(r25)	storf f10, -428(r2)
loov	loadi r8, 80444(sp) add_i r9, r8, 1 storf r9, 80444(sp) loadi r11, 80440(sp) br_ne r9, r11, r34	add_i r2, r2, 4 br_ne r2, r6, r35

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

$$T_{A,M,O} = \sum_{i=1}^{n} C_{i,A} P_{i,M,O} = \mathbf{C_A \cdot P_{M,O}} \tag{3}$$

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 the detect that the expression `Y(J,K) * 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 `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

```
        TMP = 0.0
        DO 1 I = 1, N
            TMP = TMP + Y(I,K) * Z(I,L)
1       CONTINUE
        DO 2 I = 1, N
            X(I) = TMP
2       CONTINUE
```

The contribution of this code to the total execution time is

$$Time = N(P_{MRSL} + P_{ARSL} + 2 \cdot P_{ARR2} + P_{ARR1} + P_{SRSL} + P_{TRSL} + 2 \cdot P_{LOOV}) + 2 \cdot P_{LOIN} + P_{TRSL}.$$

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.

reduced parameters	HP 720			MIPS M/2000			Sparcstation 1+		
	-O0	-O1	-O2	-O0	-O1	-O2	-O0	-O2	-O3
memory latency	108	104	61	173	135	52	545	511	269
integer add	95	60	29	165	89	69	247	188	41
integer multiply	442	419	170	574	463	489	1132	1265	950
logical operations	193	139	95	229	245	144	586	598	383
single prec. add	99	53	45	175	139	95	319	299	233
single prec. multiply	128	96	35	260	201	200	406	343	133
double prec. add	100	73	45	223	167	117	488	427	233
double prec. multiply	129	117	35	348	279	276	799	598	254
division	300	234	188	780	669	575	2648	2592	1933
procedure calls	99	96	72	328	238	161	215	76	77
address	136	76	42	462	262	147	426	267	166
branches & iteration	151	97	41	286	164	95	318	135	105
intrinsic functions	2561	2490	2477	3306	3246	3405	7442	7747	8568

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 Sparcstation 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.

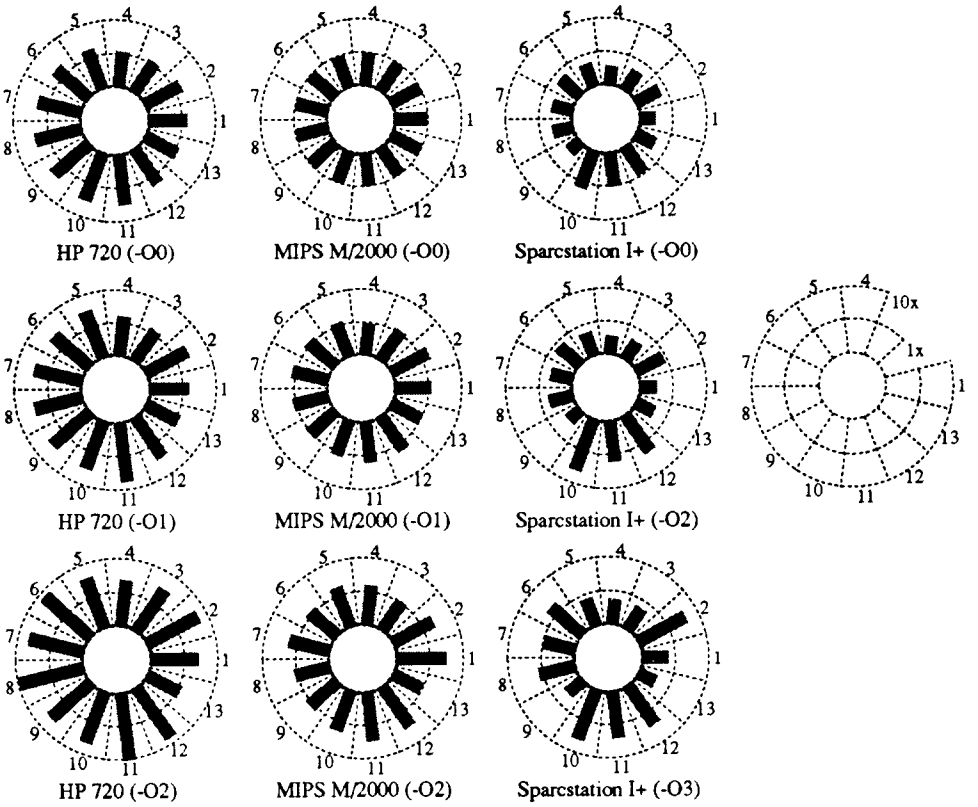

Figure 1: Performance shapes (pershapes) of different optimization levels. The thirteen dimensions correspond to the same thirteen parameters used in table 1. All dimensions are normalized with respect to the MIPS M/2000 with optimization level 0.

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.

	Minimum opt level		Maximum opt level	
Machine	Average	RMS	Average	RMS
HP-9000/720	−8.51 %	21.84 %	+3.42 %	35.60 %
MIPS M/2000	+1.95 %	16.81 %	+10.64 %	33.67 %
Sparcstation 1+	−7.52 %	22.87 %	−6.03 %	25.34 %

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.

Program	Minimum Opt. Level		Maximum Opt. Level	
	Average	RMS	Average	RMS
Doduc	+4.10 %	6.76 %	−14.06 %	16.93 %
Fpppp	+5.93 %	11.74 %	−19.39 %	29.97 %
Tomcatv	−11.92 %	13.54 %	−18.44 %	21.60 %
Matrix300	−37.87 %	39.00 %	−46.00 %	51.33 %
Nasa7	−18.01 %	19.98 %	−4.14 %	12.30 %
Spice2g6	+11.87 %	17.36 %	+17.66 %	35.87 %
ADM	−18.17 %	22.73 %	−7.40 %	7.43 %
QCD	+30.72 %	31.04 %	+43.98 %	54.11 %
MDG	+12.15 %	15.43 %	+3.23 %	13.43 %
TRACK	+16.71 %	17.16 %	−14.78 %	15.54 %
BDNA	−13.18 %	14.27 %	+0.32 %	15.19 %
OCEAN	+3.45 %	4.26 %	+45.84 %	48.85 %
DYFESM	−25.76 %	28.62 %	+10.40 %	27.87 %
ARC2D	−35.28 %	35.63 %	−26.75 %	35.38 %
TRFD	−22.04 %	26.37 %	−8.44 %	15.31 %
FLO52	−19.50 %	22.86 %	+34.12 %	47.42 %
Alamos	+2.71 %	11.15 %	+27.69 %	34.48 %
Baskett	+16.33 %	16.58 %	+24.42 %	25.28 %
Erathostenes	−14.58 %	18.54 %	−45.02 %	47.01 %
Linpack	−6.25 %	15.74 %	+23.01 %	31.54 %
Livermore	+12.41 %	17.16 %	+21.98 %	31.45 %
Mandelbrot	+6.17 %	8.01 %	+1.65 %	10.48 %
Shell	+6.21 %	21.60 %	+33.60 %	39.38 %
Smith	−18.14 %	19.86 %	+1.27 %	28.78 %
Whetstone	−11.82 %	16.87 %	−22.69 %	25.58 %
Totals	−4.83 %	20.59 %	+2.48%	31.89 %

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 5: **Error distribution for execution time predictions**					
level	< 5 %	< 10 %	< 15 %	< 20 %	< 30 %
no optimization	15 (22.06)	26 (36.76)	39 (54.41)	46 (64.71)	61 (85.29)
max optimization	11 (12.86)	19 (24.29)	28 (37.14)	36 (48.57)	47 (68.29)

level	> 30 %	> 40 %	> 50 %
no optimization	11 (14.71)	3 (4.41)	0 (0.00)
max optimization	27 (35.71)	17 (22.86)	10 (12.86)

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.

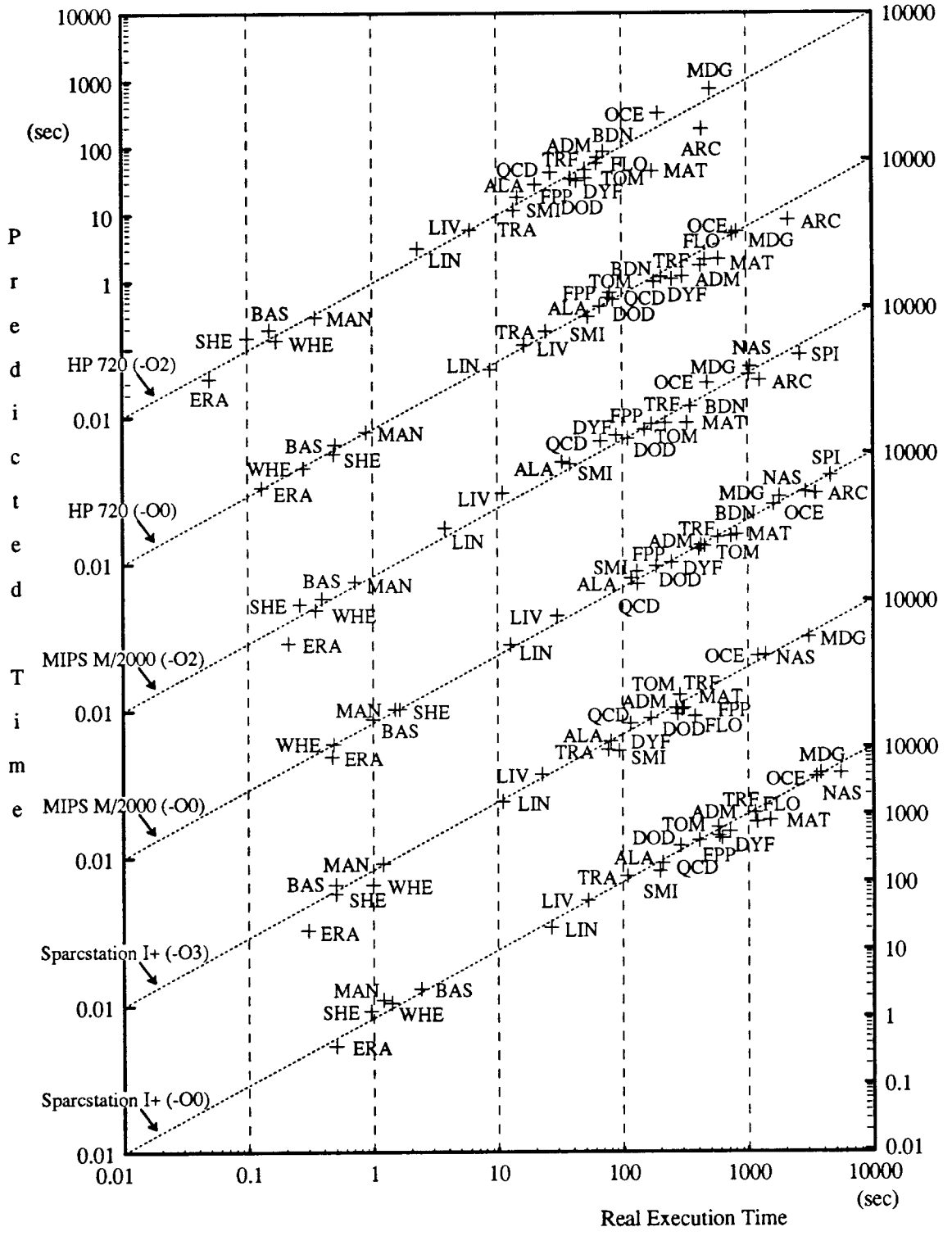

Figure 2: Each broken diagonal line, which corresponds to a particular machine and optimization level combination, shows the accuracy of the predictions compared to the real execution times. The left end point of each diagonal line maps to (0.01, 0.01) and the right end point to (10000, 10000). Points 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(2,3*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(2,3*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
2     CONTINUE
```

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.

```
      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)
                      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
                ENDIF
2       CONTINUE
```

Here the values of common subexpressions are computed once and stored in variables *I3*, *J3*, and *K3*[4]. 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

[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.

program	HP 720		MIPS M/2000		Sparcstation 1+		Geom. Mean
	-O1	-O2	-O1	-O2	-O2	-O3	Max. Opt.
Doduc	1.307	2.123 21	1.255	1.701 21	1.439	1.468 20	1.744 21
Fpppp	1.344	2.000 22	1.222	1.437 23	1.479	1.541 19	1.642 22
Tomcatv	1.504	3.497 10	1.445	2.994 10	1.866	1.927 16	2.722 14
Matrix300	1.377	3.413 11	1.263	2.475 14	3.788	4.854 2	3.448 7
Nasa7	1.477	3.318 14	1.300	2.817 11	3.759	3.953 3	3.331 9
Spice2g6	1.345	2.560 19	1.250	1.739 19	1.231	1.462 21	1.867 20
ADM	1.305	4.000 6	1.372	–	2.506	2.646 10	3.253 10
QCD	1.374	2.793 16	1.351	1.957 18	1.443	1.621 18	2.069 18
MDG	1.215	1.698 24	1.250	1.701 20	1.208	1.238 25	1.529 24
TRACK	1.316	1.786 23	1.377	1.700 22	1.318	1.403 23	1.621 23
BDNA	1.414	2.890 15	1.381	2.088 16	1.237	1.440 22	2.056 19
OCEAN	1.370	3.891 7	1.408	3.344 8	2.066	2.924 8	3.363 8
DYFESM	1.468	6.993 3	1.335	4.525 3	4.367	5.263 1	5.502 2
ARC2D	1.340	4.878 5	1.368	3.417 7	2.118	3.606 6	3.917 5
TRFD	1.664	7.143 2	1.361	4.338 4	3.690	3.891 5	4.940 3
FLO52	1.460	8.333 1	1.360	6.008 2	3.610	3.937 4	5.820 1
Alamos	1.397	3.344 12	1.311	3.571 5	1.362	2.558 11	3.126 11
Baskett	1.316	3.333 13	1.370	2.564 13	2.331	2.801 9	2.882 13
Erathostenes	1.300	2.597 18	1.305	2.237 15	1.667	1.667 17	2.132 17
Linpack	1.600	3.831 8	1.410	3.344 9	2.584	3.268 7	3.472 6
Livermore	1.473	2.703 17	1.570	2.725 12	2.045	2.326 12	2.578 15
Mandelbrot	1.348	2.545 20	1.429	2.083 17	2.000	2.000 15	2.197 16
Shell	1.634	4.902 4	1.592	6.289 1	1.357	2.093 14	4.011 4
Smith	1.350	3.597 9	1.282	3.472 6	2.000	2.105 13	2.973 12
Whetstone	1.218	1.647 25	1.200	1.372 24	1.300	1.300 24	1.432 25
Geom. Mean	1.392	3.271	1.348	2.665	1.973	2.296	2.722

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.

machines	Coefficient of Correlation	level of significance	Spearman's Rank Correlation	level of significance
HP 720 and MIPS M/2000	0.8677	.0002	0.9417	.0003
HP 720 and Sparc 1+	0.7390	.0009	0.7954	.0012
MIPS M/2000 and Sparc 1+	0.5656	.0070	0.7652	.0020

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 ($\hat{\mu}_1$ and $\hat{\mu}_2$) and the significance level of the following statistical test: $\hat{\mu}_1 \leq \hat{\mu}_2$. 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
          T0 = 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) - T0
2     CONTINUE
```

```
      DO 4 J = 1, 20
          T0 = 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) - T0
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 other 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 interesting 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 others 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][5]. 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

Machine	Compiler	Name/Location
VAX-11/785	BSD Unix F77 1.0	arpa.berkeley.edu
MIPS M/2000	MIPS F77 2.0	mammoth.berkeley.edu
Sparcstation 1+	Sun F77 1.3	heffal.berkeley.edu
VAX-11/785	Ultrix Fort 4.5	pioneer.arc.nasa.gov
Amdahl 5860	Amdahl F77 2.0	prandtl.nas.nasa.gov
CRAY Y-MP/8128	CRAY CFT77 4.0.1	reynolds.arc.nasa.gov
IBM RS/6000 530	IBM XL Fortran 1.1	coyote.berkeley.edu
Motorola M88K	Motorola F77 2.0b3	rumble.berkeley.edu

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.

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

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.

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

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.

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.

compiler	strength reduction	address calculation	inline substitution	loop unrolling
BSD Unix F77 1.0	partial	marginal	no	no
Mips F77 2.0 -O2	yes	yes	marginal	yes
Mips F77 2.0 -O1	no	yes	no	no
Sun F77 1.3 -O3	partial	marginal	no	yes
Sun F77 1.3 -O2	partial	no	no	yes
Sun F77 1.3 -O1	no	no	no	yes
Ultrix Fort 4.5	yes	yes	no	no
Amdahl F77 2.0	no	no	no	no
CRAY CFT77 4.0.1	yes	yes	yes	yes
IBM XL Fortran 1.1	yes	yes	partial	yes
Motorola F77 2.0b3	partial	no	no	no

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 abled 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.

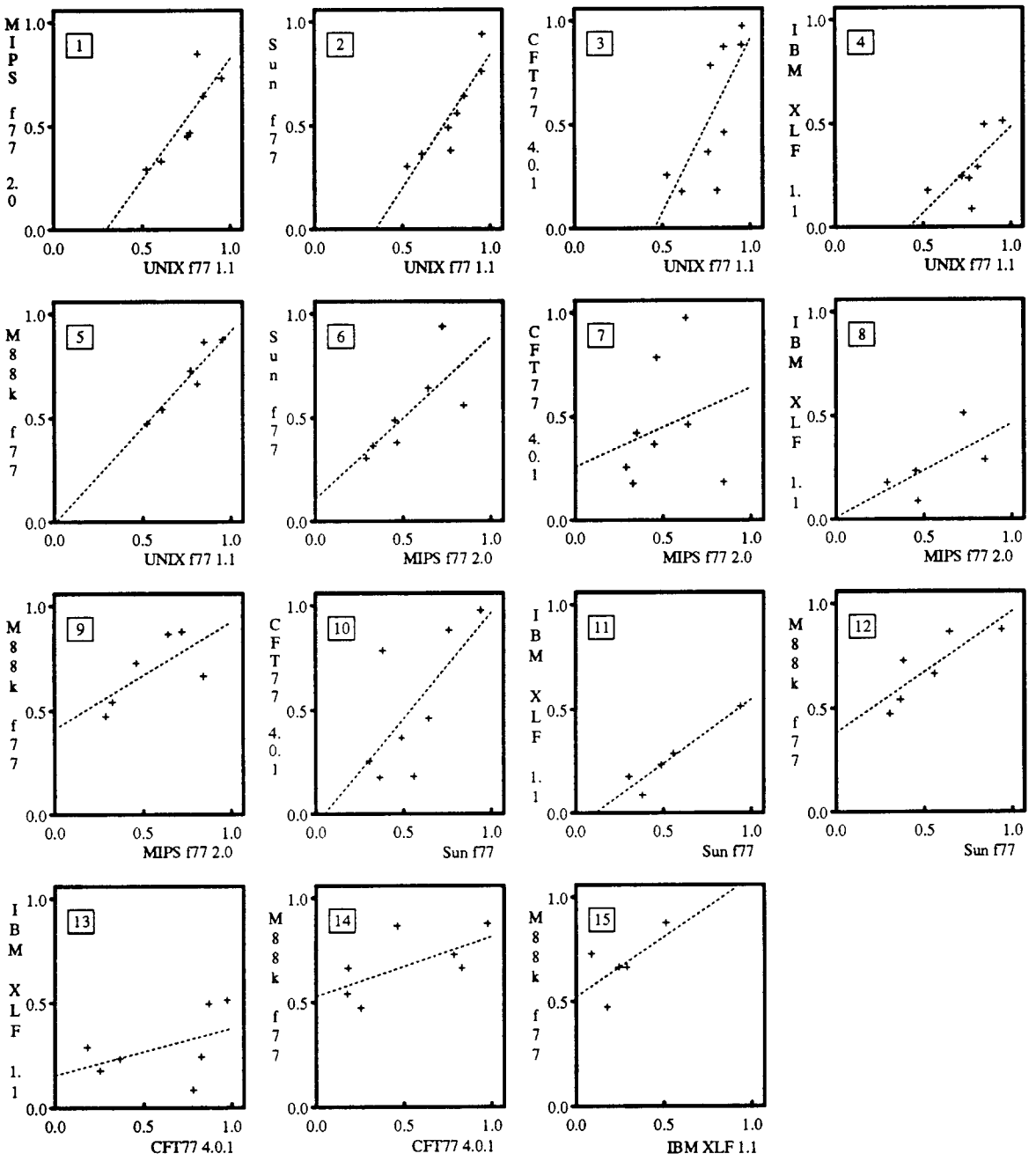

Figure 4: Correlation between execution time improvements of various optimizing compilers. Each graph includes the best linear fit.

Compiler 1	Compiler 2	Obs.	Slope	y Intercept	Correlation coefficient	Level of significance
BSD Unix F77 1.0	MIPS F77 2.0	7	1.1883	–0.3600	0.8294	**0.020**
BSD Unix F77 1.0	Sun F77 1.3	8	1.2981	–0.4555	0.8987	**0.001**
BSD Unix F77 1.0	CRAY CFT77 4.01	9	1.6864	–0.7733	0.7376	**0.015**
BSD Unix F77 1.0	IBM XLF 1.1	7	0.8282	–0.3452	0.6892	0.040
BSD Unix F77 1.0	M88K F77 2.0b3	6	0.9912	–0.0547	0.9489	**0.001**
MIPS F77 2.0	Sun F77 1.3	7	0.7816	0.1064	0.7454	**0.025**
MIPS F77 2.0	CRAY CFT77 4.01	8	0.3829	0.2566	0.2699	0.300
MIPS F77 2.0	IBM XLF 1.1	5	0.4523	0.0086	0.6361	0.150
MIPS F77 2.0	M88K F77 2.0b3	6	0.5109	0.4106	0.6899	0.070
Sun F77 1.3	CRAY CFT77 4.01	8	1.0224	–0.0558	0.6864	0.040
Sun F77 1.3	IBM XLF 1.1	5	0.6152	–0.0686	0.9510	**0.007**
Sun F77 1.3	M88K F77 2.0b3	6	0.5805	0.3829	0.8325	**0.020**
CRAY CFT77 4.01	IBM XLF 1.1	7	0.2220	0.1555	0.4608	0.170
CRAY CFT77 4.01	M88K F77 2.0b3	7	0.2855	0.5375	0.6329	0.070
IBM XLF 1.1	M88K F77 2.0b3	5	0.5764	0.5298	0.6322	0.150

Table 12: Slope, y–intercept, correlation coefficient, and level of significance for pairs of compilers.

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].

[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 X21Y21 (X,Y)
      DOUBLE PRECISION X(21), Y(21)
      REAL XX(21), YY(21)
      DATA XX /0.2, 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,
     .         4804.5,5743.49/

      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 number 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)
         YX = Y(IP,J) - Y(IM,J)
         XY = X(I,JP) - X(I,JM)
         YY = Y(I,JP) - Y(I,JM)
         A  = 0.250 * (XY * XY + YY * YY)
         B  = 0.250 * (XX * XX + YX * YX)
         C  = 0.125 * (XX * XY + YX * 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. * X(I,J) + X(IM,J)
         QXX = Y(IP,J) - 2. * Y(I,J) + Y(IM,J)
         PYY = X(I,JP) - 2. * X(I,J) + X(I,JM)
         QYY = Y(I,JP) - 2. * 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 * QXX + B * QYY - C * QXY + YX * 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(IP,J)`, `X(I,J)`, and `X(IM,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 to 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 *XLF* 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 * 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 data 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.

```
      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
            DO 110 I = 1, L
               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:

```
      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(JCOLNO+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+LOCC)
      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

[AhoA86] Aho, A.V., Sethi, R., and Ullman J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1986.

[Alle87] Allen, R. and Kennedy, K., "Automatic Translation of FORTRAN Programs to Vector Form", *ACM Transactions on Programming Languages and Systems*, Vol.9, No.4, October 1987, pp. 491-542.

[Arno83] Arnold, C.N., "Vector Optimization on the CYBER 205", *Proc. of the 1983 Int. Conf. on Parallel Processing*, Columbus, Ohio, August 23-26 1983, pp. 530-536.

[BalH86] Bal, H.E. and Tanenbaum, A.S., "Language- and Machine-Independent Global Optimization on Intermediate Code", *Computer Languages*, Vol.11, No.2, 1986, pp. 105-121.

[Bane88] Banerjee, U., *Dependency Analysis for Supercomputing*, Kluwer Academic Publishers, Boston, 1988.

[Bras88] Braswell, R.N. and Keech, M.S., "An Evaluation of Vector FORTRAN 200 Generated by CYBER 205 and ETA-10 Pre-Compilation Tools", *Proc. of the Supercomputing '88 Conf.*", Orlando, Florida, November 14-18 1988, pp. 106-113.

[Call88] Callahan, D., Dongarra, J., and Levine, D., "Vectorizing Compilers: A Test Suite and Results", *Proc. of the Supercomputing '88 Conf.*, Orlando, Florida, November 14-18 1988, pp. 98-105.

[Call90] Callahan, D., Kennedy, K., and Carr, S., "Improving Register Allocation for Subscript Variables", *Proc. of the ACM SIGPLAN '90 Conf. on Prog. Lang. Design and Implementation*, White Plains, New York, June 1990, pp. 53-65.

[Call91] Callahan, D., Kennedy, K., and Porterfield, A., "Software Prefetching", *Proc. of the 2nd Int. Conf. on Arch. Support for Prog. Lang. and Oper. Sys. (ASPLOS III)*, Santa Clara, California, April 8-11 1991, pp. 40-52.

[Chai82] Chaitin, G.J., "Register Allocation and Spilling via Graph Coloring", *Proc. of the SIGPLAN '82 Symp. on Compiler Construction*, June 1982, pp. 98-105.

[Chow83] Chow, F., *A Portable Machine-Independent Global Optimizer*, Ph.D. Dissertation and Technical Report No. 83-254, Computer Systems Laboratory, Stanford University, December 1983.

[Chow84] Chow, F. and Hennessy J.L., "Register Allocation by Priority-Based Coloring", *Proc. of the ACM SIGPLAN '84 Symp. on Compiler Construction*, Vol.19, No.6, June 1984, pp. 222-233.

[Chow86] Chow, F., Himelstein, M., Killian, E., and Weber, L., "Engineering a RISC Compiler System", *Proc. of the Compcon '86 Conf.*, San Francisco, California, March 4-6 1986, pp. 132-137.

[Cmel91] Cmelik, R.F., Kong., S.I., Ditzel, D.R., and Kelly, E.J., "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks", *Proc. of the 2nd Int. Conf. on Arch. Support for Prog. Lang. and Oper. Sys. (ASPLOS III)*, Santa Clara, California, April 8-11 1991, pp. 290-302.

[Cock80] Cocke, J. and Markstein, P., "Measurement of Program Improvement Algorithms", Technical Report No. RC-8111 (#35193), IBM Yorktown Heights, February 7 1980.

[Cude89] Cuderman, K.J. and Flynn, M.J., "The Relative Effects of Optimization on Instruction Architecture Performance", Technical Report No. CSL-TR-89-398, Computer Systems Laboratory, Stanford University, October 1989.

[Cybe90] Cybenko, G., Kipp, L., Pointer, L., and Kuck, D., *Supercomputer Performance Evaluation and the Perfect Benchmarks*, University of Illinois Center for Supercomputing R&D Technical Report 965, March 1990.

[Dong87] Dongarra, J.J., Martin, J., and Worlton, J., "Computer Benchmarking: paths and pitfalls", *Computer*, Vol.24, No.7, July 1987, pp. 38-43.

[Ferr91] Ferrante, J., Sarkar, V., and Thrash, W., "On Estimating and Enhancing Cache Effectiveness", *Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, California, August 1991.

[Hira91] Hiranandani, S., Kennedy, K., and Tseng, C.W., "Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines", *Proc. of the Supercomputing '91 Conf.*, Albuquerque, New Mexico, November 18-22 1991, pp. 86-100.

[Jaza86] Jazayeri, M. and Haden, M., "Optimizing Compilers Are Here (mostly)", *SIGPLAN Notices*, Vol.21, No.5, May 1986, pp. 61-63.

[John86] Johnson, M.S. and Miller, T.C., "Effectiveness of a Machine-Level, Global Optimizer", *Proc. of the SIGPLAN '86 Symp. on Compiler Construction*, Palo Alto, June 25-27 1986, pp. 99-108.

[Knut71] Knuth, D.E., "An Empirical Study of Fortran Programs", *Software-Practice and Experience*, Vol.1, 1971, pp. 105-133.

[Lind86] Lindsay, D.S and Bell, T.E., "Directed Benchmarks for CPU Architecture Evaluation", *Proc. of the CMG '86 Conf*, Las Vegas, Nevada, December 9-12 1986, pp. 379-385.

[Much86] Muchnick, S.S., "Here Are (Some of) the Optimizing Compilers", *SIGPLAN Notices*, Vol.21, No.2, February 1986, pp. 11-15.

[Noba89] Nobayashi, H. and Eoyang, C., "A Comparison Study of Automatically Vectorizing FORTRAN Compilers", *Proc. of the Supercomputing '89 Conf.*, Reno, Nevada, November 13-17 1989, pp. 820-825.

[O'Br90] O'Brien, K., Hay, B., Minisk, J., Schaffer, H., Schloss B., Shepherd, A., and Zaleski, M., "Advanced Compiler Technology for the RISC System/6000 Architecture", *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation, 1990, pp. 154-161.

[Patt85] Patterson, D.A., "Reduced Instructions Set Computers", *Comm of the ACM*, Vol.28, No.1, January 1985, pp. 8-21.

[Paud86] Pauda, D.A. and Wolfe, M.J., "Advanced Compiler Optimizations for Supercomputers", *Comm. of the ACM*, Vol.29, No.12, December 1986, pp. 1184-1201.

[Peut77] Peuto, B.L., and Shustek, L.J., "An Instruction Timing Model of CPU Performance", *The 4th Annual Symposium on Computer Architecture*, Vol.5, No.7, March 1977, pp. 165-178.

[Pond90] Ponder, C.G., "An Analytical Look at Linear Performance Models", Lawrence Livermore National Laboratory, Technical Report UCRL-JC-106105, September 1990.

[Port89] Porterfield, A., *Software Methods for Improvement of Cache Performance on Supercomputers Applications*, Ph.D. Dissertation and Technical Report No. COMP-TR89-93, Rice University, 1989.

[Rich89] Richardson, S. and Ganapathi, M., "Interprocedural Optimization: Experimental Results", *Software–Practice and Experience*, Vol.19, No.2, February 1989, pp. 149-170.

[Saav89] Saavedra-Barrera, R.H., Smith, A.J., and Miya, E. "Machine Characterization Based on an Abstract High-Level Language Machine", *IEEE Trans. on Comp.* Vol.38, No.12, December 1989, pp. 1659-1679.

[Saav90a] Saavedra-Barrera, R.H. "The SPEC and Perfect Club Benchmarks: Promises and Limitations", *Hot Chips Symposium 2*. Santa Clara, CA, August 1990.

[Saav90b] Saavedra-Barrera, R.H. and Smith, A.J., *Benchmarking and The Abstract Machine Characterization Model*, U.C. Berkeley Technical Report No. UCB/CSD 90/607, November 1990.

[Saav92a] Saavedra-Barrera, R.H., *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*, Ph.D. Thesis, U.C. Berkeley, Technical Report No. UCB/CSD 92/684, February 1992.

[Saav92b] Saavedra, R.H. and Smith, A.J., "Analysis of Benchmark Characteristics and Benchmark Performance Prediction", *paper in preparation*, 1992.

[Shus78] Shustek, L.J., *Analysis and Performance of Instruction Sets*, Ph.D. Dissertation, Stanford University, May 1978.

[Sing91] Singh, J.P. and Hennessy J.L., "An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelization", *Int. Symp. on Shared Memory Multiprocessing*, Tokyo, Japan, April 1991, pp. 25-36.

[SPEC89] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.1, Issue 1, Fall 1989.

[SPEC90] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.2, Issue 3, Summer 1990.

[SPEC91a] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.3, Issue 2, Spring 1991.

[SPEC91b] SPEC, "*SPEC Newsletter: Benchmark Results*", Vol.3, Issue 3, Fall 1991.

[Wolf91] Wolf, M. and Lam, M., "A Data Locality Optimizing Algorithm", *Proc. of the ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, June 1991, pp. 30-44.

[Wolf85] Wolfe, M. and Macke, T., "Where are the Optimizing Compilers", *SIGPLAN Notices*, Vol.20, No.11, November 1985, pp. 64-77.

[Worl84] Worlton, J., "Understanding Supercomputer Benchmarks", *Datamation*, September 1, 1984, pp. 121-130.

Appendix A

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

machine	optim	SRSL	ARSL	MRSL	DRSL	ERSL	XRSL	TRSL
HP 720	-O0	40	89	117	273	84	10173	180
HP 720	-O1	40	47	108	198	75	10447	135
HP 720	-O2	7	40	31	167	58	10426	20
MIPS M/2000	-O0	40	159	258	583	316	26136	213
MIPS M/2000	-O1	< 1	101	169	498	229	26068	113
MIPS M/2000	-O2	< 1	95	145	395	219	26255	< 1
Sparcstation 1+	-O0	80	305	384	1856	2419	18414	510
Sparcstation 1+	-O2	75	298	322	1849	2398	18512	499
Sparcstation 1+	-O3	< 1	234	133	1444	2575	19603	35

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

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

Group 3: Integer Arithmetic Operations (single, local)

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

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)

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

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

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

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

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

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)

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

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

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

Group 9, 10: Conditional and Logical Parameters

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

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

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

Group 13, 14: Branching and DO loop Parameters

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

Group 15: Intrinsic Functions (single precision)

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

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)

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

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

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

Groups 19: Intrinsic Functions (type conversion)

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

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

HP-9000/720

	Optimization level 0			Optimization level 1			Optimization level 2		
program	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
Doduc	85	85	−0.12	65	66	+1.08	40	34	−13.67
Fpppp	78	90	+15.30	58	69	+19.52	39	35	−10.28
Tomcatv	182	157	−14.09	121	111	−8.42	52	35	−31.40
Matrix300	598	340	−43.13	434	159	−63.43	175	45	−74.44
Nasa7	–	–	–	1120	715	−36.15	387	318	−17.80
Spice2g6	–	–	–	–	–	–	509	727	+42.73
ADM	252	172	−31.92	193	116	−39.90	63	58	−8.08
QCD	81	108	+33.21	59	86	+46.59	29	41	+81.40
MDG	827	861	+4.04	681	631	−7.34	487	474	−2.69
TRACK	25	28	+14.63	19	21	+8.39	14	12	−13.13
BDNA	208	185	−11.32	147	152	+3.47	72	87	+21.37
OCEAN	764	791	+3.52	558	548	−1.84	196	320	+63.07
DYFESM	307	187	−39.09	209	101	−51.70	44	32	−25.92
MG3D	–	–	–	–	–	–	1088	1088	−0.03
ARC2D	2137	1267	−40.70	1594	593	−62.81	439	187	−57.47
TRFD	464	329	−28.98	279	202	−27.70	65	71	+9.61
FLO52	426	274	−35.70	292	142	−51.20	51	48	−7.23
Alamos	67	67	+0.45	48	37	−22.38	20	28	+38.73
Baskett	0.50	0.58	+16.00	0.38	0.41	+7.89	0.15	0.20	+33.33
Erathostenes	0.13	0.13	+1.54	0.10	0.08	−20.00	0.05	0.04	−26.00
Linpack	8.8	7.6	−13.67	5.5	5.1	−7.82	2.3	3.3	+41.30
Livermore	16.5	17.7	+7.27	11.2	11.4	+1.79	6.1	6.1	−0.49
Mandelbrot	0.89	0.89	0.00	0.66	0.69	+4.55	0.35	0.31	−11.43
Shell	0.49	0.42	−13.67	0.30	0.25	−16.67	0.10	0.15	+50.00
Smith	54	47	−12.15	40	32	−18.23	15	18	+25.17
Whetstone	0.28	0.26	−7.14	0.23	0.21	−8.70	0.17	0.14	−17.64

Table 18: Nonoptimized and optimized benchmark results on the HP 720. All times are reported in seconds and the errors are computed as $100\times(pred-real)/real$.

MIPS M/2000

	Optimization level 0			Optimization level 1			Optimization level 2		
program	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
Doduc	187	208	+11.69	149	143	–4.03	110	107	–2.73
Fpppp	247	239	–8.14	202	174	–13.86	172	177	+2.91
Tomcatv	452	415	–3.21	313	251	–19.81	151	145	–3.97
Matrix300	816	614	–24.77	646	370	–42.72	330	182	–44.85
Nasa7	2906	2634	–9.36	2234	1703	–23.77	1033	980	–5.13
Spice2g6	4576	4539	–0.81	3660	2679	–26.80	2630	1937	–26.35
ADM	424	426	+0.47	309	259	–16.18	–	165	–
QCD	131	176	+34.48	97	124	+27.84	67	98	+46.27
MDG	1796	2254	+25.50	1436	1566	+9.05	1056	1281	+21.31
TRACK	58	71	+22.20	42	47	+10.49	34	31	–9.94
BDNA	733	582	–20.60	531	388	–26.93	351	328	–6.55
OCEAN	1618	1722	+6.47	1148	1063	–7.40	483	732	+51.55
DYFESM	407	370	–9.10	305	221	–27.54	90	119	+32.22
ARC2D	3465	2470	–28.70	2548	1155	–54.67	1014	799	–21.20
TRFD	577	566	–1.87	424	335	–20.99	133	184	–16.74
FLO52	721	853	–15.52	530	529	+0.19	120	208	+73.33
Alamos	118	139	+16.95	90	85	–5.56	33	48	+45.45
Baskett	1.00	1.13	+13.00	0.73	0.80	+9.59	0.39	0.46	+17.94
Erathostenes	0.47	0.31	–21.28	0.36	0.19	–47.22	0.21	0.10	–52.38
Linpack	12.7	14.5	+13.94	9.0	10.3	+14.44	3.80	5.13	+35.00
Livermore	30.0	38.6	+28.80	19.1	22.9	+19.90	11.0	16.8	+52.72
Mandelbrot	1.50	1.59	+6.00	1.05	1.04	–0.95	0.72	0.82	+13.88
Shell	1.64	1.59	–2.44	1.03	0.64	–37.86	0.26	0.38	+46.15
Smith	132	113	–15.05	103	68	–34.24	38	45	+17.80
Whetstone	0.48	0.48	+0.00	0.40	0.36	–10.00	0.35	0.31	–11.43

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 $100 \times (pred - real)/real$.

Sparcstation 1+

	Optimization level 0			Optimization level 2			Optimization level 3		
program	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)	real (sec)	pred (sec)	error (%)
Doduc	407	410	+0.74	283	348	+22.79	277	205	−25.79
Fpppp	590	481	+10.63	399	421	+5.62	383	188	−50.80
Tomcatv	576	637	−18.46	309	484	+56.48	299	239	−19.96
Matrix300	1502	815	−45.70	397	545	+38.26	309	251	−18.72
Nasa7	5516	4046	−26.66	1467	2826	+92.65	1393	1540	+10.52
Spice2g6	5348	6660	+24.54	4504	5669	−25.86	3659	2320	+36.60
ADM	716	551	−23.05	286	438	+53.11	271	253	−6.72
QCD	274	341	+24.47	190	265	+39.87	169	176	+4.27
MDG	3809	4072	+6.90	3153	3480	+10.37	3079	2804	−8.94
TRACK	108	122	+13.30	82	98	+19.54	77	60	−21.28
BDNA	1270	1173	−7.63	1027	914	−11.09	882	760	−13.87
OCEAN	3538	3551	+0.37	1713	2210	+29.00	1209	1485	+22.90
DYFESM	617	437.1	−29.10	141	296	+109.50	117	146	+24.91
ARC2D	6437	4091	−36.44	3039	3083	+1.45	1785	1757	−1.58
TRFD	1181	765	−35.26	320	525	+64.35	303	248	−18.18
FLO52	1132	1049	−7.28	314	785	+150.40	288	393	+36.27
Alamos	207	188	−9.26	152	121	−20.29	81	80	−1.12
Baskett	1.40	1.68	+20.00	0.60	1.13	+88.33	0.50	0.61	+22.00
Erathostenes	0.50	0.38	−24.00	0.30	0.27	−10.00	0.30	0.13	−56.67
Linpack	35.9	21.7	−19.03	13.9	15.0	+7.92	11.0	10.2	−7.27
Livermore	52.6	53.5	+1.17	25.7	42.6	+65.76	22.6	25.7	+13.72
Mandelbrot	2.40	2.70	+12.50	1.20	2.67	+122.50	1.20	1.23	+2.50
Shell	0.95	1.28	+34.74	0.70	1.07	+52.86	0.43	0.45	+4.65
Smith	198	144	−27.22	99	113	−14.23	94	57	−39.17
Whetstone	1.30	0.86	−28.33	1.00	0.77	−23.00	1.00	0.61	−39.00

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

compiler	Integer		Floating Point		Mixed	
	local	global	local	global	local	global
BSD Unix F77 1.0	no	no	no	no	no	no
Mips F77 2.0 -O2	yes	yes	partial[1]	partial[1]	partial[1]	no
Mips F77 2.0 -O1	yes	no	no	no	no	no
Sun F77 1.3 -O3	yes	no	no	no	no	no
Sun F77 1.3 -O2	yes	no	no	no	no	no
Sun F77 1.3 -O1	no	no	no	no	no	no
Ultrix Fort 4.5	yes	no	yes	no	yes	no
Amdahl F77 2.0	no	no	no	no	no	no
CRAY CFT77 4.0.1	yes	yes	yes	yes	yes	yes
IBM XL Fortran 1.1	yes	yes	yes	no	yes	no
Motorola F77 2.0b3	yes	no	yes	no	yes	no

1 variables assigned to constant expressions are not propagated

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

compiler	Integer		Floating Point		Mixed	
	local	global	local	global	local	global
BSD Unix F77 1.0	yes	no	yes	no	no	no
Mips F77 2.0 -O2	yes	yes	yes	yes	yes	yes
Mips F77 2.0 -O1	yes	no	yes	no	partial[1]	no
Sun F77 1.3 -O3	yes	yes	yes	yes	yes	yes
Sun F77 1.3 -O2	yes	yes	yes	yes	yes	yes
Sun F77 1.3 -O1	no	no	no	no	no	no
Ultrix Fort 4.5	yes	yes	yes	yes	yes	yes
Amdahl F77 2.0	no	no	no	no	no	no
CRAY CFT77 4.0.1	yes	partial[2]	yes	partial[2]	yes	partial[2]
IBM XL Fortran 1.1	yes	yes	partial[3]	partial[3]	partial[3]	partial[3]
Motorola F77 2.0b3	yes	partial[2]	yes	partial[2]	yes	partial[2]

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)."

compiler	Integer		Floating Point		Mixed	
	local	global	local	global	local	global
BSD Unix F77 1.0	partial[1]	partial[1]	no	no	marginal[2]	marginal[2]
Mips F77 2.0 -O2	yes	yes	yes	yes	yes	yes
Mips F77 2.0 -O1	no	no	no	no	no	no
Sun F77 1.3 -O3	yes	yes	yes	partial[3]	yes	partial[3]
Sun F77 1.3 -O2	yes	yes	yes	partial[3]	yes	partial[3]
Sun F77 1.3 -O1	no	no	no	no	no	no
Ultrix Fort 4.5	yes	yes	yes	yes	yes	yes
Amdahl F77 2.0	no	no	no	no	no	no
CRAY CFT77 4.0.1	yes	partial[3]	yes	partial[3]	yes	partial[3]
IBM XL Fortran 1.1	yes	yes	yes	yes	yes	yes
Motorola F77 2.0b3	yes	no	yes	no	yes	no

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).

compiler	Integer		Floating Point		Mixed	
	local	global	local	global	local	global
BSD Unix F77 1.0	partial[1]	no	partial[1]	no	partial[1]	no
Mips F77 2.0 -O2	partial[2]	marginal[1]	partial[2]	marginal[1]	partial[2]	marginal[1]
Mips F77 2.0 -O1	marginal[1]	no	marginal[1]	no	marginal[1]	no
Sun F77 1.3 -O3	no	no	no	no	no	no
Sun F77 1.3 -O2	no	no	no	no	no	no
Sun F77 1.3 -O1	no	no	no	no	no	no
Ultrix Fort 4.5	yes	partial[3]	yes	partial[3]	yes	partial[3]
Amdahl F77 2.0	no	no	no	no	no	no
CRAY CFT77 4.0.1	yes	no	yes	no	yes	no
IBM XL Fortran 1.1	partial[4]	marginal[1]	partial[4]	marginal[1]	partial[4]	marginal[1]
Motorola F77 2.0b3	no	no	no	no	no	no

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).

compiler	Integer		Floating Point		Mixed	
	local	global	local	global	local	global
BSD Unix F77 1.0	no	no	no	no	no	no
Mips F77 2.0 -O2	yes	yes	yes	yes	yes	yes
Mips F77 2.0 -O1	no	no	no	no	no	no
Sun F77 1.3 -O3	yes	yes	yes	yes	yes	yes
Sun F77 1.3 -O2	yes	yes	no	no	no	no
Sun F77 1.3 -O1	no	no	no	no	no	no
Ultrix Fort 4.5	yes	yes	yes	yes	yes	yes
Amdahl F77 2.0	no	no	no	no	no	no
CRAY CFT77 4.0.1	yes	yes	yes	yes	yes	yes
IBM XL Fortran 1.1	yes	yes	yes	yes	yes	yes
Motorola F77 2.0b3	no	no	no	no	no	no

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

compiler	strength reduction	address calculation	inline substitution		loop unrolling
			apply	affects	
BSD Unix F77 1.0	partial[1]	marginal[2]	no	no	no
Mips F77 2.0 -O2	yes	yes	partial[2]	no	yes[4]
Mips F77 2.0 -O1	no	yes	no	no	no
Sun F77 1.3 -O3	partial[1]	marginal[1]	no	no	yes[4]
Sun F77 1.3 -O2	partial[1]	marginal[1]	no	no	yes[4]
Sun F77 1.3 -O1	partial[1]	no	no	no	yes[4]
Ultrix Fort 4.5	yes	yes	no	no	no
Amdahl F77 2.0	no	no	no	no	no
CRAY CFT77 4.0.1	yes	yes	yes	yes	yes[4]
IBM XL Fortran 1.1	yes	yes	yes	no[3]	no
Motorola F77 2.0b3	partial[1]	no	no	no	no

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.