Performance Characterization of Optimizing Compilers

Rafael H. Saavedra, *Member, IEEE*, and Alan Jay Smith, *Fellow, IEEE*

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 model considers all machines to be different implementations of the same high level language abstract machine; in previous research, the model has been used as a basis to analyze machine and benchmark performance. In this paper, we show that our model can be extended to characterize the performance improvement provided by optimizers and to predict the run time of optimized programs, and measure the effectiveness of several compilers in implementing different optimization techniques.

***Index Terms*—Performance evaluation, optimizing compilers, benchmarking, execution time prediction, CPU performance characterization.**

I. 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 [9], [26]. Computer manufacturers are using these suites to evaluate the overall performance and improve the designs of future machines and compilers. By concentrating on whole systems, however, it is not possible to explain why machines perform well on some benchmarks but badly on others, or to predict the behavior on programs not included in the suites. Observed CPU performance is the result of the interactions between many hardware and software components, i.e., integer, floating point, and branch units, memory system, applications, libraries and optimizing compilers, and a comprehensive performance evaluation should characterize their respective contributions [13]. Our research has focused on 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 [20], [21], [22].

The basis for our research has been to model all computers as machines that execute Fortran. We refer to this as our abstract machine model. By measuring the execution time for primitive Fortran operations, and by counting the frequency of occurrence of the various operations in programs of interest, we have been able to accurately predict the execution time.

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.

For brevity 3) is not presented here but appears in [23].

In Section II 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 III 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.

Finally, in Section IV 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

Manuscript received September 1992; revised July 1994.
R.H. Saavedra is with the Computer Science Department, Henry Salvatori Computer Science Center, University of Southern California, Los Angeles, CA 90089-0781. e-mail: saavedra@cs.usc.edu.
A.J. Smith is with the Computer Science Division, EECS Department, University of California, Berkeley, CA 94720-1776.
IEEECS Log Number S95019.

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.

II. 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. 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.

A. Previous Performance Studies in Compiler Optimization

Knuth, in 1971, was the first to quantify the potential improvement due to optimization [12]. He statically and dynamically analyzed a number of Fortran programs and measured the speedup that could be obtained by hand-optimizing them.

Papers reporting on the effectiveness of real optimizers were not published until the beginning of the eighties [2], [6], [8], [10], [11], [14], [27]. 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 basic types or only on a small subset. As we will see in Section IV.B, very few optimizers in commercially available compilers are able to detect optimizations on all basic types; this can result in a significant loss of potential improvement when the precision and/or type is changed.

The performance of IBM's PL/1L experimental compiler is evaluated in [8]. The compiler has three 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.[1] Chow [5], who wrote the Uopt portable global optimizer at Stanford, gives statistics about the number of times that each optimization was detected and for some optimizations he reports the amount of improvement produced. On 13 small Pascal programs the average speedup observed 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.[2] Bal and Tanenbaum [2] found using the Amsterdam Compiler Kit optimizer that the speedup 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 [11] found that on the same programs used by Chow the average speedup was 1.381.

There have been other studies dealing with other aspects of optimization. Richardson and Ganapathi [19] have shown that certain types of interprocedural data flow analysis provide 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 [4]. Most commercial vectorizing compilers are based either on the VAST or KAP precompilers developed at Pacific Sierra Research and Kuck and Associates, which are compared in [3]. Singh and Hennessy [25] are studying the potential and limitations of automatic parallelization.

B. 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:

1) Most standard benchmarks and large scientific programs are written in Fortran;
2) Fortran is relatively simple to work with;
3) 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 also applies to other similar high level languages such as C, Ada, or Modula-3.

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 [20], [21]. Measurements are made by using timing loops with and without the operation to be measured. Such measurements are complicated by the fact that some operations are not separable from other operations (e.g., store), and that it is very difficult to get precise values in the presence of noise (e.g., cache misses, task switching) and low resolution clocks [20], [21]. We have also called this machine analysis phase *narrow spectrum benchmarking* or *micro 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 have to be done in an analysis at the machine instruction level [17].

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 instrument and execute the program to count the number of times each line is executed. From these two sets of measurements, we can compute the number of times each primitive operation is executed.

The third part of our methodology is to combine the operation execution times and frequencies to predict the running

1. We quantify the improvement produced by an optimizer in terms of the speedup, i.e., the ratio between the unoptimized execution time to the optimized time. The overall speedup on all benchmarks is computed by taking the geometric mean of the individual speedups. For consistency, we also follow these rules when describing work done by others.

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.

time of a given program on a given machine without having to 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 [21], [22]. We can also easily estimate the performance of hypothetical machines (or modifications of existing machines) on a variety of real or proposed workloads by replacing measured parameters in our models with proposed or hypothetical ones.

It is very important to note 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 was found to cause unacceptable errors. The curve fitting approach has been used and has been observed to be of very limited value [18]. 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 and exhibit almost no predictive power.

In [20] 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. These abstract operations were also combined into a set of reduced parameters, each of which was associated with the performance of a specific CPU functional unit. The use of such reduced parameters permitted straightforward machine to machine comparisons.

In [21], [22] we studied the characteristics of the SPEC, Perfect Club and other common 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. Our results were successful in accurately predicting `inconsistent' machine performance, i.e., that machine A is faster than B on program x, but slower on program y. 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.

C. 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

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

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

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

Our problem here is to obtain $\mathbf{C_{A,O}}$ and $\mathbf{P_{M,O}}$ by only making an analysis of the (source) program and running experiments with optimization enabled. The above scheme is general and accounts for different classes of optimizations, including those performed at the source, intermediate, or object code level representations of the program. In the next section, we presented a detailed example of how optimizations affect the distribution and performance of abstract operations.

III. Extending the Abstract Model to Include Optimization

From the discussion of the preceding subsection we can proceed to classify the effect of optimizations according to how they affect (1). Basically, we identify two types of optimizations: those that modify $\mathbf{C_p}$ (Type I), and those that only affect $\mathbf{P_M}$ (Type II). Optimizations of Type I change the program's distribution of abstract operations, either by removing or replacing some amount of code. In addition, these optimizations may or may not affect the performance of individual abstract operations ($P_{i,M}$). What is relevant in Type I optimizations is that they affect the number of abstract operations present in the original program. Hence, 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 detailed information about how an arbitrary optimizer works internally.

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 simple low-level optimizations that improve machine code sequences, but also other high-level optimizations, like 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 is 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 consist 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)       (3)
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 1D and 2D 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 I 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 (-O0) and with maximum optimization (-O2).[3] (We have made inconsequential changes to the syntax of the machine instructions to make the code more readable.)

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

The above example shows that even when the two sequences of machine instructions, one unoptimized and the other optimized, are very different, we can still identify in both cases, the sequence of machine instructions corresponding to the 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.*

TABLE I
NONOPTIMIZED AND OPTIMIZED ASSEMBLER CODE FOR THE INNERMOST LOOP. (ON THE LEFT SIDE WE SHOW THE ABSTRACT MACHINE OPERATIONS REPRESENTED BY THE ASSEMBLER CODE.)

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`

It is important to note that the optimizations applied to the program excerpt in Table I 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 two loads, two multiplies, and four add/sub operations in the sequence associated with **ARR2**. This requires determining 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 (affine function), so future values can be 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 2D 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 charac-

3. The meaning of minimum and maximum optimization level is a function of the options offered by each compiler.

terizer 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 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

$$T_{A,M,O} = \sum_{i=1}^{n} C_{i,A}\, P_{i,M,O} = \mathbf{C_A} \cdot \mathbf{P_{M,O}}. \qquad (4)$$

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 is derived from optimizing sequences of machine 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. Third, 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 Section IV we characterize the particular optimizations that compilers can apply.

B. 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 [16] to the code excerpt given in (3) to dramatically reduce the number of operations and consequently the execution time.[4] These source to source transformations produce the following version

```
      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

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

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 always predict the execution time after optimization.

C. 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. In this section we compare different machine characterizations under various levels of compiler optimization.

We ran the system characterizer using different optimization levels on three high performance workstations. The complete results, including those without optimization, can be found in [23]. Table II shows a set of 13 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 [20]), a type of Kiviat graph, as shown in Fig. 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.

The results in Table II 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

4. Loop interchange transposes the order of the loops. This allows the compiler to 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.

TABLE II
Normalized Optimization Performance Results in Terms of the Reduced Parameters

reduced parameters	HP 720			MIPS M/2000			Sparcstation 1+		
	-O0	-O1	-O2	-O0	-O1	-O2	-O0	-O2	-O3
memory latency	1.000 (108)	0.963	0.565	1.000 (173)	0.780	0.301	1.000 (545)	0.938	0.526
integer add	1.000 (95)	0.632	0.305	1.000 (165)	0.539	0.418	1.000 (247)	0.761	0.218
integer multiply	1.000 (442)	0.948	0.385	1.000 (574)	0.807	0.852	1.000 (1132)	1.117	0.751
logical operations	1.000 (193)	0.720	0.492	1.000 (229)	1.070	0.629	1.000 (586)	1.020	0.640
single prec. add	1.000 (99)	0.535	0.455	1.000 (175)	0.794	0.543	1.000 (319)	0.937	0.779
single prec. multiply	1.000 (128)	0.750	0.273	1.000 (260)	0.773	0.769	1.000 (406)	0.845	0.388
double prec. add	1.000 (100)	0.730	0.450	1.000 (223)	0.749	0.525	1.000 (488)	0.875	0.546
double prec. multiply	1.000 (129)	0.907	0.271	1.000 (348)	0.802	0.793	1.000 (799)	0.748	0.425
division	1.000 (300)	0.780	0.627	1.000 (780)	0.858	0.737	1.000 (2648)	0.979	0.746
procedure calls	1.000 (99)	0.970	0.727	1.000 (328)	0.726	0.491	1.000 (215)	0.353	1.013
address	1.000 (136)	0.559	0.309	1.000 (462)	0.567	0.318	1.000 (426)	0.627	0.622
branches & iteration	1.000 (151)	0.642	0.272	1.000 (286)	0.573	0.332	1.000 (318)	0.425	0.778
intrinsic functions	1.000 (2561)	0.972	0.967	1.000 (3306)	0.982	1.030	1.000 (7442)	1.041	1.106

(Each parameter represents a particular characteristic of the machine and is computed from a subset of basic abstract machine parameters. The results of an individual machine have been normalized with respect to the unoptimized case. Times inside parenthesis 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.)

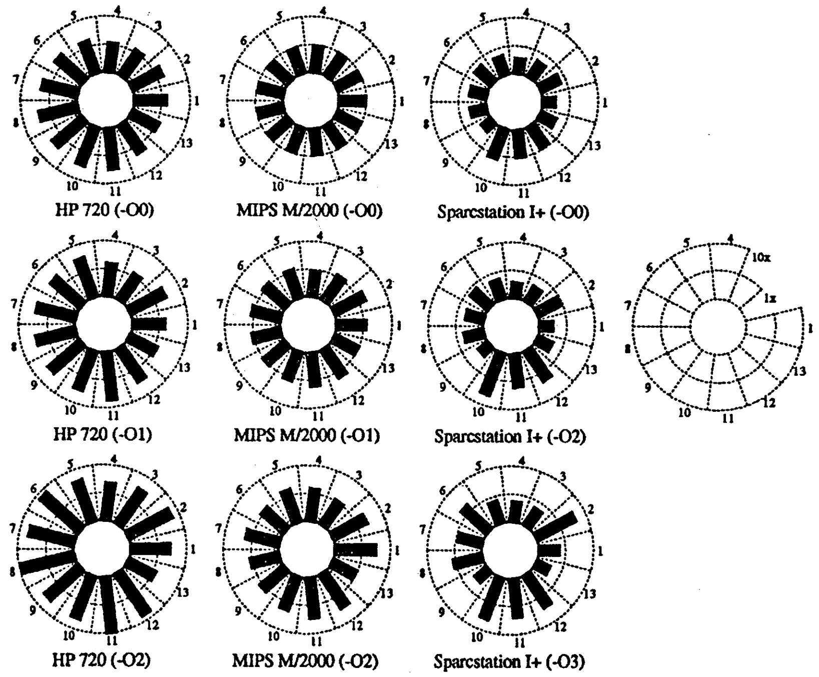

Fig. 1. Performance shapes (pershapes) of different optimization levels. The 13 dimensions correspond to the same parameters used in Table I. Here all dimensions are normalized with respect to the MIPS M/2000 with optimization level 0 (no optimization).

on the MIPS M/2000 the average time at the maximum level of optimization is larger than the 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.

D. 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 [22]. 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.

In Fig. 2 we show the comparison between the real and

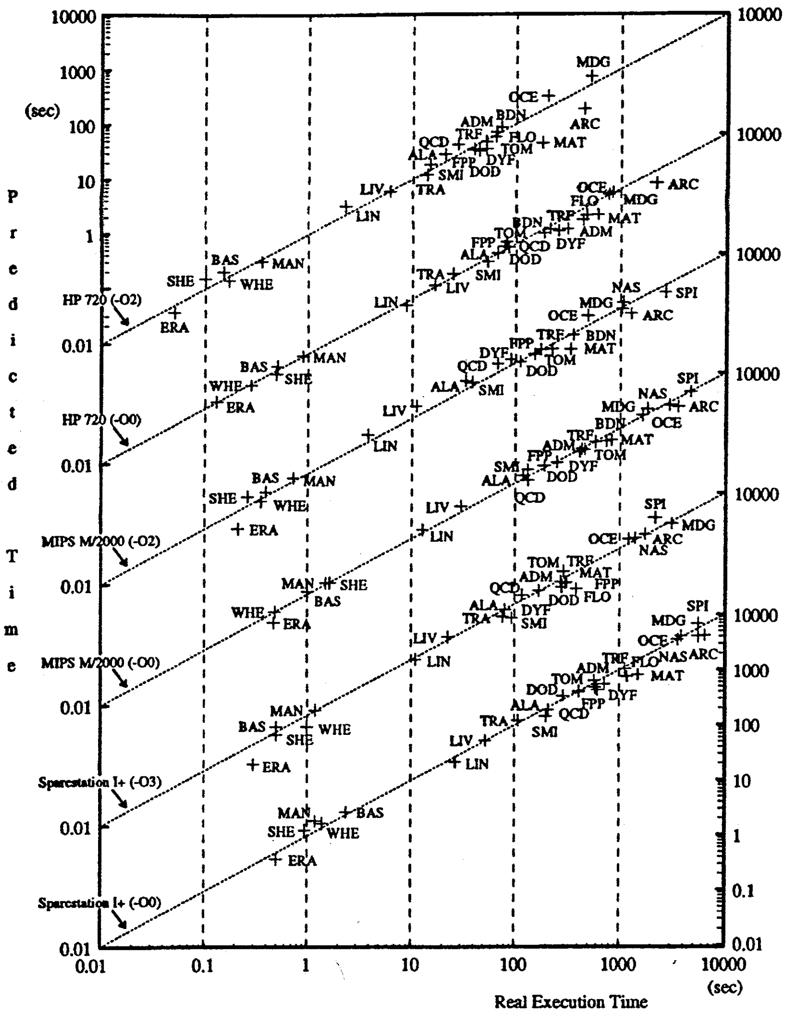

Fig. 2. Each broken diagonal line, which corresponds to a particular machine and optimization level combination, shows the accuracy of the predictions compared to the real execution times. The left end point of each diagonal line maps to (0.01, 0.01) and the right end point to (10000, 10000). Points along the diagonal are of the form (*T*, *T*). All scales are in seconds.

predicted execution times for both optimized and unoptimized programs; the abbreviations for the various programs are explained in [21]. 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. Summaries of the predictive errors, by machine and program, are presented in Tables III and IV. (The complete execution times and relative errors can be found in [23]). The RMS error, shown in Tables III and IV, 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.

Fig. 2 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%; the smallest of all benchmarks. This is because of the relatively large number of intrinsic functions executed, which do not run faster when the program is optimized.

TABLE III
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.)

	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 %

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 Section III.G.

TABLE IV
SUMMARY OF EXECUTION TIME ERRORS BY PROGRAM AT THE MINIMUM AND MAXIMUM LEVELS OF OPTIMIZATION FOR THE PROGRAMS ON FIG. 2. (RMS REPRESENTS THE ROOT MEAN SQUARE ERROR. THE INDIVIDUAL REAL AND PREDICTED EXECUTION TIMES ARE GIVEN IN [22].)

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 %

E. 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 V we compare the distribution of errors for both nonoptimized and optimized programs; we can see that for maximum optimization the average error increases. For the results shown in Fig. 2, Table V shows that while 85% of the nonoptimized 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 nonoptimized predictions has 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 noninvariant optimizations. In Table IV 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.

TABLE V
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 PARENTHESES). THE ERROR IS COMPUTED AS THE RELATIVE DISTANCE TO THE REAL EXECUTION TIME.)

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)

Common subexpression elimination in this context is not an invariant optimization as defined in Section III.A. 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 the errors are as high as 47% and 81% [23].

```
      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
```

F. Improving Predictions in the Presence of Noninvariant Optimizations

We can improve our predictions of run times by identifying the applicable noninvariant 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.

Here the values of common subexpressions are computed once and stored in variables *I3*, *J3*, and *K3*.[5] 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%.

5. 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.

```
        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
```

By distinguishing the invariant and noninvariant optimizations, we can assess the performance impact of each, because the performance improvement due to noninvariant optimizations is equal to the difference between our predicted improvement, considering only invariant optimizations, and the real execution time.

G. 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 VI we show the program speedup achieved by each optimization level for the three machines previously discussed.

In Section II.A 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 VI, however, show that at the maximum level of optimization the speedups observed on Fortran programs are frequently larger than two, with some programs experiencing speedups of more than a factor of five.

The results of Table VI 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 multidimensional 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 1D arrays, so address collapsing, the elimination of time consuming address calculations in multidimensional 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 the case for *MDG* and *WHETSTONE*. Further discussion of the optimizations possible in these programs appears in Section V.A.

TABLE VI
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.)

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

It is dangerous to draw conclusions about the effectiveness of the different optimizers from the speedup results of Table VI. The overall speedup is as much a function of the quality of the nonoptimized object code as it is of the optimizer, since it is always possible to improve the overall speedup by generating worse nonoptimized 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 VI 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*, 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) [23].

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 VII 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.

TABLE VII

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.)

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

IV. 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 [20]. We have developed a Fortran program consisting of a number of micro benchmarks 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 measurements 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 $(\hat{\mu}_1 \text{ and } \hat{\mu}_2)$ 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.

Fig. 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 loops are almost identical with only one difference: in the right-hand side, the first set of definitions of variables *W*1, *W*2, and *W*3 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.

```
      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
```

Fig. 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 *W*1, *W*2, and *W*3 are not used. The three definitions can be eliminated by the optimizer.

A. 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 [1]. [5] and [2] 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 variable 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 multidimensional array elements in innermost loops by precomputing outside the loop the addresses of the elements referenced in the first iteration and updating their values by adding a constant in subsequent iterations. This optimization is based on the observation that in the majority of nested loops the sequence of machine addresses associated with a specific array reference form an arithmetic progression, which is completely determined by the first value and the increment.
- **Strength Reduction**: This optimization is a generalization of address collapsing as it attempts to replace a time-consuming computation with an equivalent but faster one. One example is replacing an exponentiation having a small integer exponent which is known at compile time with a series of multiplications. Similarly, multiplies can often be replaced by additions. On array references, the combination of strength reduction and code motion makes it possible to collapse address computations.
- **Subroutine Inlining**: Substitute for a call to a subroutine the actual subroutine code. This avoids the overhead of the call, and exposes optimizations present at the site of the call. Although most optimizers claim that they do subroutine inlining, they tend to differ substantially in the amount of integration they perform.
- **Loop Unrolling**: Expand several iterations of the loop into a single basic block and hence expose new optimization opportunities. This also reduces the impact of the loop overhead.

In this paper we have concentrated on scalar optimizations. But in addition to the above optimizations, there are other program transformations which have been designed to exploit vector and parallel hardware. A description of a large test suite and evaluation of vectorizing Fortran compilers can be found in [4].

B. Optimization Results

We have run our experiments on several optimizing compilers and for different levels of optimization. In Table VIII we give the list of machines along with their corresponding compilers. The complete results are presented [23], while Tables IX–XI summarize the same information. In our experiments we make a distinction between local and global optimizations. A local optimization (Tables IX–X) 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 XI) 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.

TABLE VIII
LIST OF MACHINES WITH THEIR RESPECTIVE FORTRAN COMPILERS.

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 IX
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	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

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.

TABLE X
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.)

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

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.

Common subexpression elimination is successfully detected by most compilers in all contexts. Although the IBM *XLF* compiler identified almost all common subexpressions, it missed a couple which involved floating point adds and multiplies. The reason is that the RS/6000 series provides, in addition to the normal add and multiply operations, a combined multiply-add instruction. In our experiments the compiler generated for two occurrences of the same subexpression, a multiply followed by an add in one case, but a single multiply-add for the other case. As a result of this, it did not recognize that the two expressions were identical. Missing an optimization as a result of applying another, however, is in many cases acceptable if the first optimization provides a better improvement.

Table XI 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 [6]. In contrast, a real inline substitution is done by the IBM *XLF* 1.1 compiler [15], 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.

TABLE XI
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 IN TABLES VIII AND IX. INSTEAD OF SUMMARIZING THE RESULTS OF EXPERIMENTS ON DIFFERENT DATA TYPES, THEY REPRESENT THE AMOUNT OF IMPROVEMENT THAT EACH OPTIMIZER PRODUCED ON A SINGLE TEST.)

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

V. 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 [17], [24], and [7]. 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 [20], 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 extends reasonably well 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. Finally, we also evaluated the optimization improvement provided by several optimizers on the Fortran SPEC, Perfect Club, and other popular benchmarks.

ACKNOWLEDGMENTS

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.

We would like to thank Ken Stevens for providing access to facilities at NASA Ames and funding for this work, and 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.

REFERENCES

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

[2] H.E. Bal and A.S. Tanenbaum, "Language- and machine-independent global optimization on intermediate code," *Computer Languages*, vol. 11, no. 2, 1986, pp. 105–121.

[3] R.N. Braswell and M.S. Keech, "An evaluation of vector FORTRAN 200 generated by CYBER 205 and ETA-10 pre-compilation tools," *Proc. Supercomputing '88 Conf.*, Orlando, Fla., Nov. 14–18, 1988, pp. 106–113.

[4] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: A test suite and results," *Proc. Supercomputing '88 Conf.*, Orlando, Fla., Nov. 14–18 1988, pp. 98–105.

[5] F. Chow, *A Portable Machine-Independent Global Optimizer*, PhD dissertation and Technical Report No. 83-254, Computer Systems Laboratory, Stanford Univ., Dec. 1983.

[6] F. Chow, M. Himelstein, E. Killian, and L. Weber, "Engineering a RISC Compiler System," *Proc. Compcon '86 Conf.*, San Francisco, Calif., Mar. 4–6, 1986, pp. 132–137.

[7] R.F. Cmelik, S.I. Kong, D.R. Ditzel, and E.J. Kelly, "An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks," *Proc. 2nd Int'l Conf. Arch. Support for Prog. Lang. and Oper. Sys. (ASPLOS III)*, Santa Clara, Calif., Apr. 8–11, 1991, pp. 290–302.

[8] J. Cocke and P. Markstein, "Measurement of program improvement algorithms," Technical Report No. RC-8111 (#35193), IBM, Feb. 7, 1980.

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

[10] M. Jazayeri and M. Haden, "Optimizing compilers are here (mostly)," *SIGPLAN Notices*, vol. 21, no. 5, May 1986, pp. 61–63.

[11] M.S. Johnson and T.C. Miller, "Effectiveness of a machine-level, global optimizer," *Proc. SIGPLAN '86 Symp. on Compiler Construction*, Palo Alto, Calif., June 25–27, 1986, pp. 99–108.

[12] D.E. Knuth, "An empirical study of Fortran programs," *Software—Practice and Experience*, vol. 1, 1971, pp. 105–133.

[13] D.S. Lindsay and T.E. Bell, "Directed benchmarks for CPU architecture evaluation," *Proc. CMG '86 Conf.*, Las Vegas, Nev., Dec. 9–12, 1986, pp. 379–385.

[14] S.S. Muchnick, "Here are (some of) the optimizing compilers," *SIGPLAN Notices*, vol. 21, no. 2, Feb. 1986, pp. 11–15.

[15] K. O'Brien, B. Hay, J. Minisk, H. Schaffer, B. Schloss, A. Shepherd, and M. Zaleski, "Advanced compiler technology for the RISC system/6000 architecture," *IBM RISC System/6000 Technology*, SA23-2619, IBM Corp., 1990, pp. 154–161.

[16] D.A. Pauda and M.J. Wolfe, "Advanced compiler optimizations for supercomputers," *Comm. ACM*, vol. 29, no. 12, Dec. 1986, pp. 1,184–1,201.

[17] B.L. Peuto and L.J. Shustek, "An instruction timing model of CPU performance," *4th Ann. Symp. on Computer Architecture*, vol. 5, no. 7, Mar. 1977, pp. 165–178.

[18] C.G. Ponder, "An analytical look at linear performance models," Lawrence Livermore National Laboratory, Technical Report UCRL-JC-106105, Sept. 1990.

[19] S. Richardson and M. Ganapathi, "Interprocedural optimization: Experimental results," *Software—Practice and Experience*, vol. 19, no. 2, Feb. 1989, pp. 149–170.

[20] R.H. Saavedra-Barrera, A.J. Smith, and E. Miya, "Machine characterization based on an abstract high-level language machine," *IEEE Transactions on Computers*, vol. 38, no. 12, Dec. 1989, pp. 1,659–1,679.

[21] R.H. Saavedra-Barrera, *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*, PhD thesis, Univ. of California, Berkeley, Technical Report No. UCB/CSD 92/684, Feb. 1992.

[22] R.H. Saavedra and A.J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," paper in preparation, 1992.

[23] R.H. Saavedra and A.J Smith, *Performance Characterization of Optimizing Compilers*, Univ. of Southern California Technical Report No. USC-CS-92-525, Aug. 1992.

[24] L.J. Shustek, *Analysis and Performance of Instruction Sets*, PhD dissertation, Stanford Univ., May 1978.

[25] J.P. Singh and J.L. Hennessy, "An empirical investigation of the effectiveness and limitations of automatic parallelization," *Proc. Int'l Symp. Shared Memory Multiprocessing*, Tokyo, Japan, Apr. 1991, pp. 25–36.

[26] "SPEC," *SPEC Newsletter: Benchmark Results*, vol. 1, no. 1, Fall 1989.

[27] M. Wolfe and T. Macke, "Where are the optimizing compilers," *SIGPLAN Notices*, vol. 20, no. 11, Nov. 1985, pp. 64–77.

Rafael H. Saavedra (S'87-M'92) received his PhD from the University of California at Berkeley in 1992. He is assistant professor in the Computer Science Department at the University of Southern California.

Alan Jay Smith received the BS degree in electrical engineering from the Massachusetts Institute of Technology in 1971 and the MS and PhD degrees in computer science from Stanford University, the last degree in 1974. He was an NSF Graduate Fellow.

He is currently a professor in the Computer Science Division of the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley, where he has been on the faculty since 1974. His research interests include the analysis and modeling of computer systems and devices, computer architecture, and operating systems. He has published a large number of research papers, including one that won the IEEE Best Paper Award for the best paper in the *IEEE Transactions on Computers* in 1979.

Dr. Smith is a fellow of the IEEE and is a member of the Association for Computing Machinery, the IFIP Working Group 7.3, the Computer Measurement Group, Eta Kappa Nu, Tau Beta Pi,, and Sigma Xi. He is on the Board of Directors (1993–1995) and was chairman (1991–1993) of the ACM Special Interest Group on Computer Architecture (SIGARCH), was chairman (1983–1987) of the ACM Special Interest Group on Operating Systems (SIGOPS), was on the Board of Directors (1985–1989) of the ACM Special Interest Group on Measurement and Evaluation (SIGMETRICS), was an ACM national lecturer (1985–1986) and an IEEE distinguished visitor (1986–1987), was an associate editor of the *ACM Transactions on Computer Systems* (1982–1993), is a subject area editor of the *Journal of Parallel and Distributed Computing*, and is on the editorial board of the *Journal of Microprocessors and Microsystems*. He was program chairman for the SIGMETRICS '89/Performance '89 Conference, program cochair for the Second (1990) and Sixth (1994) Hot Chips Conferences, and has served on numerous program committees.