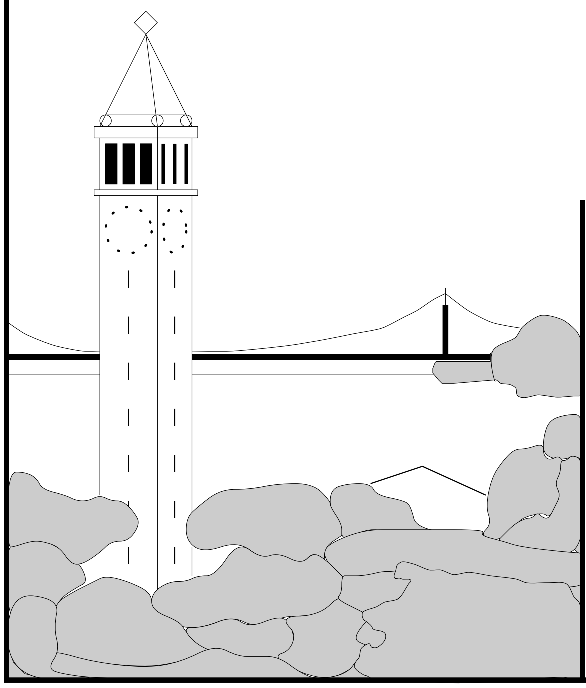
Microbenchmarking and Performance Prediction for Parallel Computers

Stephen J. Von Worley
Alan Jay Smith



Report No. UCB/CSD-95-873

May 1995

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Microbenchmarking and Performance Prediction for Parallel Computers*

Stephen J. Von Worley
Alan Jay Smith

Computer Science Division
EECS Department
University of California
Berkeley, California 94720-1776

Technical Report UCB/CSD-95-873
May 16, 1995

Abstract

Previous research on this project (in work by Saavedra and Smith) has presented performance evaluation of sequential computers. That work presented (a) measurements of machines at the source language primitive operation level; (b) analysis of standard benchmarks; (c) prediction of run times based on separate measurements of the machines and the programs; (d) analysis of the effectiveness of compiler optimizations; and (e) measurements of the performance and design of cache memories.

In this paper, we extend the earlier work to parallel computers. We describe a portable benchmarking suite and performance prediction methodology, which accurately predicts the run times of Fortran 90 programs running upon supercomputers. The benchmarking suite measures the optimization capabilities of a given Fortran 90 compiler, execution rates of abstract Fortran 90 operations, and the processing characteristics of the underlying architecture as exposed by compiler-generated code. To predict the run time of an arbitrary program, we combine our benchmark results with dynamic execution measurements, and augment the resulting prediction with simple factors which account for overhead due to architecture-specific effects, such as remote reference latencies. We measure two supercomputers: a dedicated 128-node TMC CM-5, a distributed memory multiprocessor, and a 4-node partition of a Cray YMP-C90, a tightly-integrated shared memory multiprocessor. Our measurements show that the performance of the YMP-C90 far outstrips that of the CM-5, due to the quality of the compilers available and the architectural characteristics of each machine. To validate our prediction methodology, we predict the run time of five interesting kernels on these machines; nearly all of the predicted run times are within 50-percent of actual run times, much closer than might be expected.

*The authors' research has been supported principally for this work by NASA under Grant NCC 2-550, and also in part by the National Science Foundation under grants MIP-9116578 and CCR-9117028, by the State of California under the MICRO program, and by Intel Corporation, Apple Computer Corporation, Sun Microsystems, Digital Equipment Corporation, Philips Laboratories/Signetics, International Business Machines Corporation and Mitsubishi Electric Research Laboratories.

1 Introduction

Traditional benchmarking involves running a set of benchmarks believed to representative of some target workload on one or more machines, and using the resulting run times as estimates of the performance of those machines for that workload. This approach suffers from a number of defects, including the possibility that the benchmark programs may not be representative, that the interaction between the benchmarks, the compilers, and the machine architecture was unknown and could produce misleading results, and that it was impossible to accurately predict the run time for one benchmark from that of another.

In earlier work by Saavedra and Smith ([8], [26], [14]), the approach which we call *microbenchmarking* was developed to deal with the problems we list above. In [8], Saavedra presents a system which accurately predicts the run time of Fortran 77 programs across a wide range of uniprocessors. The system models the target computer as an *abstract Fortran machine*, which executes a set of Fortran abstract operations. To predict the run time of a program A on a machine M, we measure $P_M =< P_1, P_2, ..., P_n >$, where P_i is the average execution time of abstract operation i, and $C_A =< C_1, C_2, ..., C_n >$, where C_i is the number of times abstract operation i was executed in A.[1] Saavedra's system includes a set of benchmarks which measure P_M for a given architecture, and utilities which instrument a given program to determine C_A. The predicted run time $T_{M,A}$ is simply the dot product of P_M and C_A

$$T_{M,A} = \sum P_i C_i = P_M \cdot C_A$$

Once P_M has been measured, we can predict the run time of any number of arbitrary deterministic programs upon the machine M without the need for any additional measurements of M, simply by instrumenting each program A in question and tabulating C_A upon some other machine. Saavedra's work was extended in [26] to detect and evaluate compiler optimizations (and to predict the performance of optimized code), and in [14] to measure cache and TLB performance and their effect on run times.

In this paper, we extend Saavedra's work to parallel computers. The performance prediction of automatically-parallelized programs running upon supercomputers is quite difficult. We can use the results of conventional benchmark suites to construct very rough estimates of the run times of similar programs, but we cannot adequately generalize such data for use in the prediction of arbitrary programs. Accurate performance prediction requires a different approach.

Recently, the Fortran 90 standard was introduced; it extends Fortran 77 primarily by adding constructs which allow the convenient and concise expression of operations upon arrays. Several supercomputer manufacturers support and have produced Fortran 90 compilers, and most subsequent supercomputer Fortran extensions, such as High Performance Fortran (HPF) [25], are based upon Fortran 90.

This paper describes a Fortran 90 benchmarking suite which measures various parameters of a Fortran 90 compiler and associated architecture, including the array operation execution rates, the optimization capabilities of the compiler, and the processing characteristics of the underlying architecture as exposed by compiler-generated code. We use these measurements to predict the run time of several Fortran 90 kernels.

This paper is organized as follows: Section 2 outlines the performance prediction system into which our measurement suite fits, Section 3 discusses related work, Section 4 outlines the Fortran 90 standard, Section 5 describes our model of data-parallel computations, Section 6 describes the design of our measurement suite, Section 7 validates our performance prediction strategy, and Section 8 presents our conclusions.

2 System Overview

As mentioned in the Introduction, our work extends Saavedra's microbenchmarking suite into the parallel domain. Figure 1 illustrates the basic composition of the performance prediction system which we envision.

[1] This fine grain measurement strategy is termed *microbenchmarking*.

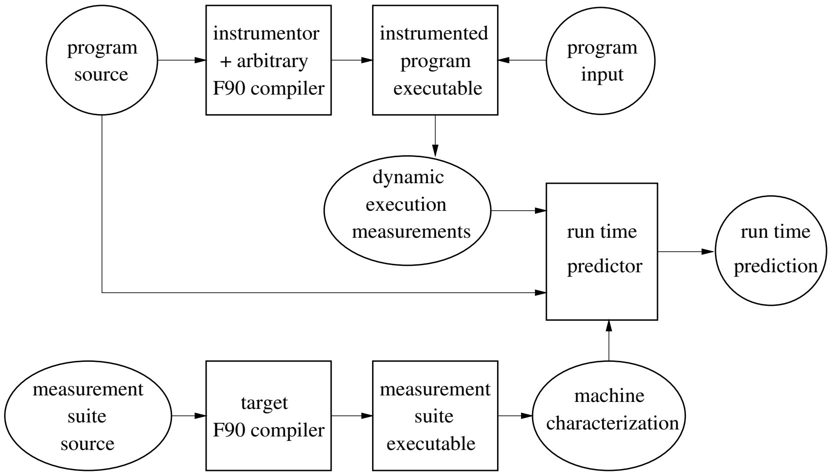


Figure 1: Overview of the Performance Prediction System

The system is composed of three major parts:[2]

- **Measurement Suite**: Measures the various parameters of our model of the target system, resulting in a *machine characterization*.
- **Instrumentor**: Instruments the target program so that it outputs dynamic execution measurements.
- **Run Time Predictor**: Predicts the run time of the target program on the target machine by combining the machine characterization, the dynamic execution measurements generated by the instrumented target program, and the original target program source.

This work details our efforts to: (a) create an appropriate model of a Fortran 90 program running upon an arbitrary supercomputer, (b) implement a Measurement Suite which estimates the parameters of this model, and (c) compose an accurate prediction methodology which will be used to create an automatic Program Instrumentor and Run Time Predictor at some later date.

We model a computer as an abstract Fortran 90 machine, in which the run time of a given operation is derived from its type and the size of its operands, in the case that its operands are arrays. We then augment this simple model to account for compiler optimizations and architecture-specific effects such as remote reference overhead. Accordingly, the following four modules compose our Measurement suite:

- **Array Operation Characterizer**: Measures the execution rate of Fortran 90 array operations, for various operand sizes.
- **Optimization Characterizer**: Determines the compiler's ability to apply parallelizing optimizations.
- **Architecture Characterizer**: Measures the parameters of the various architectural features, such as read/write latencies and scatter overhead.
- **Serial Operation Characterizer**: Measures the execution rate of Fortran 90 scalar operations.

We have designed and implemented all of the above components, except for the Serial Operation Characterizer, since its functionality is included in Saavedra's Fortran 77 measurement suite.

Each component of the Measurement Suite estimates the run time of a given operation via the linear combination of the run time of one or more carefully selected Fortran 90 code segments. For example, to crudely estimate cost of an array load, we might measure the run time of code segments A and B; code segment A copies an array into another array and code segment B stores a constant into an array:

A: `a=b` B: `a=1`

[2]Some of this terminology in this section was borrowed from [8].

If we model the system as a load-store architecture with an overhead of L per element loaded and S per element stored, t_A and t_B, the measured run times of code segments A and B, are approximately

$$t_A = nS + nL$$
$$t_B = nS$$

Thus, we can approximate the cost of an array load as $t_A - t_B = nL$.[3] Appendix A details the iterative timing method used to measure the run time of each code segment, and the complications to the measurement process which we encountered.

3 Previous and Related Work

Previous parallel performance prediction systems have taken one of three basic approaches. *Simulation-based* approaches derive the predicted run time of a program by simulating its execution on the target architecture. Simulation can be performed with varying degrees of abstraction, ranging from the instruction level to the task level. Notable examples of simulation-based approaches include:

- The Rice Parallel Processing Testbed [5], a simulator which utilizes code profiling and libraries simulating interprocessor communication to efficiently predict run time.
- Menasce and Barroso's method [6], which uses several iterations of task-level simulation to characterize the overhead due to contention in a shared memory system.
- NPAC's Fortran 90D/HPF prediction suite [28], which predicts run time via the combination of user and benchmark-specified measurements and a task graph representation of the target program, created with a tool which emulates the specific behavior of NPAC's Fortran 90D compiler.

Analytical approaches produce approximate predictions by solving for the various characteristics of a probabilistic representation of the program, usually expressed as some variant of the Markov or Petri net model. The most popular approach involves stochastic Petri nets; for an introduction, see [20] and [21]. *Kernel-based* approaches predict the run time of sections of code by interpolating from the run time of one or more similar kernels.

Unfortunately, each of the above approaches has flaws, with respect to predicting the run time of arbitrary Fortran 90 programs utilizing measurements from a portable Fortran 90 test suite. Instruction-level simulations fail because they are machine-specific. Task-level simulations assume knowledge of the number, organization, and composition of tasks composing a given parallel computation; however, we cannot accurately predict this information for an arbitrary compiler/architecture pair. For example, the Y-MP C90 `cf77` compiler conditionally parallelizes array operations based on predicted work, unpredictably affecting the characteristics and number of tasks composing the operation. Analytical approaches fail because they produce probabilistic results quantifying expected behavior, rather than determining program behavior for a specific set of inputs. Additionally, solutions to analytical representations grow very complex and resource intensive for large programs. Solely kernel-based approaches are unsuitable simply because we must measure a very large number of kernels to ensure the accurate performance prediction of arbitrary constructs. Additionally, the algorithms used to match code to kernels and to interpolate run times are potentially very complex.

An additional consideration is that our benchmarks are written in Fortran 90, which does not include constructs which can specify the mapping of computation to specific hardware. Thus, we cannot easily determine the performance of a single parallel node, since the compiler may produce executable code for a given segment of benchmarking code which runs on the host processor, some subset of the parallel nodes,

[3] Of course, since we examine only one case in this example, pipelining and other effects heavily color our run time estimate. Actual parameter calculations are more complex and involve more code segments, but involve the same basic approach.

or both. Without single node performance measurements, we cannot easily predict the run time of a task on that node, as required by many performance prediction systems.

Compiler optimizations complicate the performance prediction process. Saavedra investigated the effect of compiler optimizations in the serial domain [26]; however, optimizations in the parallel domain have a much larger effect. Compilers may apply source transformations which change the number operations executed by certain code segments by several orders of magnitude. To account for such transformations, we must collect measurements which allow us to predict how the compiler will modify and parallelize arbitrary code. Several studies have examined the parallelization/vectorization of arbitrary code segments selected in an ad hoc manner [10] [9]. Furthermore, studies have examined the overall contribution of various parallelizing optimizations to speedup [22]. These results are instructive, but not particularly useful to us; we must determine whether the compiler applies a specific operation in various specific cases.

4 Summary of the Fortran 90 Standard

This section attempts to provide a reasonable summary of the Fortran 90 standard and the terminology we use to describe it. For a more detailed treatment, see [12], and for the standard itself, see [13].

The Fortran 90 standard includes the Fortran 77 standard and extends it by adding various constructs and intrinsic functions. We use the term *operation* to denote the computation described by the application of an operator, construct, or intrinsic function. Fortran 90 operations can be classified into two main groups: *scalar* operations, which operate solely upon scalar operands, and *array* operations, which require some combination of array and scalar operands. Array operations can be further separated into three groups:

- *data parallel* operations, which produce a result array in which the ith element is some function of the ith element(s) in the array operand(s); for example, the dot product operation.
- *reduction* operations (reductions), which produce a scalar result which is some function of the array operand(s); for example, a vector sum.
- *transformation* operations, which produce a result array which may be of different dimensions (shape) than the source array(s), in which the ith element is the function of potentially many elements in the array operand(s); for example, matrix multiplication.

Fortran 90 allows Fortran 77 operations and intrinsic functions to accept array arguments, and such expressions are evaluated in a data-parallel fashion. For example, where `a`, `b`, and `c` are n-element arrays, the following statements denote the data parallel addition of `b` and `c` and the assignment of the result to `a`:

```
a=b+c
```

Note that in all such expressions (by the definition of data parallel), all array operands must be *conformable*; that is, all operands must have the same rank and dimensions, with the exception of scalars, which are conformable with all arrays.

An array operand can be an entire array, or an array *section*, which is composed of the elements in a given array between indices *x* and *y*, separated by stride *s*. We specify an array section from source array *src* as *src* ([*x*][:[*y*][:[*s*]]]), where *x* and *y* default to the beginning and end of the array, and *s* defaults to 1.

Fortran 90 defines the `where` construct, which allows the application of a data parallel operation based on the values in a conformable logical array, called a *mask*. For example, the expression

```
where (c.gt.1.0) c=0.0
```

zeroes all elements of `c` which are greater than 1.0. Fortran 90 also defines the `merge` intrinsic, which produces an array composed of the elements of two source arrays, as selected by a mask.

Fortran 90 defines a *gather* operation, which collects scattered elements from a source array into a contiguous space. Given a source array *a*, and an array *b* containing the indices of *a* to be gathered, the syntax is `a(b)`. The result array is the same size as *b*, and element i of the result is element *b(i)* of *a*. A *scatter* operation, the reverse of a gather, is specified similarly.

Fortran 90 lacks constructs which can map computations and data to specific hardware. The elements of the result of an array operation may be calculated in any order, and such computations may utilize some arbitrary subset of available hardware. Array layouts vary between machines and are compiler-specific.

Fortran 90 defines several intrinsic functions which perform array reduction operations, including:

- `sum`, `product`: Compute the sum (product) of the elements of an array.
- `maxval`, `minval`: Compute the maximum (minimum) value in an array.
- `any`, `all`: Test if any (all) elements in a logical array are true.

Fortran 90 defines several intrinsic functions which perform array transformation operations, including:

- `matmul`: Multiply two matrices.
- `transpose`: Transpose a 2-d array.
- `pack`, `unpack`: Pack (unpack) selected values of a given array into another array.

Finally, Fortran 90 defines various constructs and intrinsic functions which allow bit level manipulations of integers, modular programming, array duplication, and dynamic storage allocation.

5 Parallel Computation - Models

Fortran 90 allows the expression of data-parallel computations; this section presents various models of the execution of such computations, for the purpose of computing the run time of the operations, and subsequently, the entire program.

We consider a straightforward data-parallel computation $x = y \oplus z$, where $\oplus$ is an arbitrary binary operation. Assuming that the contribution of architectural effects is additive,[4] the execution time T is

$$T = S + \max\{D_i + C_i + X_i + Y_i + Z_i\} \;\; (0 \leq i < P) \tag{1}$$

where

P is the number of processors,
S is the time to start and terminate the parallel computation,
D_i is the time spent distributing or acquiring work on processor i,
C_i is the time spent computing on processor i,
X_i is the time spent reading the necessary elements of x on processor i,
Y_i is the time spent reading the necessary elements of y on processor i,
Z_i is the time spent writing the calculated elements of z on processor i.

We assume that $\oplus$ has a uniform execution time of c per element, when reading operands to and from the register file or lowest-level cache. Given that n is the number of elements in each array operand, Equation 1 reduces to the following forms, based upon the architecture and work-allocation strategy:

- Distributed Memory Multiprocessor or
  Shared Memory Multiprocessor, Non-Caching, Nonuniform Memory Access;
  Arrays Distributed in Blocked, Cyclic, or Blocked-Cyclic Layout;
  Owner-Computes Rule:

[4] Saavedra's research illustrated that the execution time of a program could be modelled by adding terms corresponding to various architectural effects. In any case, so long as we can model the temporal overlap of such effects by terms dependent upon the number of elements in each array operand, our conclusions still hold.

$$T \approx S + \tfrac{n}{P}(c + W_L) + \max\{X_i + Y_i\} \;\; (0 \le i < P) \tag{1.a}$$

W_L is the average time required to write an element to local storage.[5] The $\max\{X_i + Y_i\}$ term represents the maximum time taken to read required array operand elements, since under our "additive effects" assumption the processor does not overlap operand fetches. If we assume a flat communications network (i.e. one with uniform latency between each possible source and destination) a bi-level distribution of access latencies, and if we know the layout of arrays x and y and ignore congestion effects, we can easily compute X_i and Y_i. Furthermore, if x, y, and z are distributed identically, $X_i = Y_i = \frac{n}{P}R_L$, where R_L is the average time required to read an element from local storage, ignoring local memory hierarchy effects, and so

$$T \approx S + \tfrac{n}{P}(c + W_L + 2R_L) \;\; \text{(where operands are distributed identically)}$$

- Shared Memory Multiprocessor, Uniform Memory Access;
  Equally Sized Blocks Statically Assigned:

$$T \approx S + \tfrac{n}{P}(c + W_S + 2R_S) \tag{1.b}$$

  R_S and W_S are the average times required to read and write an element from or to shared memory, respectively.

- Shared Memory Multiprocessor; Uniform Memory Access;
  m-Element Blocks Dynamically Scheduled:

$$\begin{aligned} T &\approx S + \lceil \tfrac{n}{mP} \rceil D + m \lceil \tfrac{n}{mP} \rceil (c + W_S + 2R_S) \\ &\approx S + \tfrac{n}{mP} D + \tfrac{n}{P}(c + W_S + 2R_S) \text{ (for large } n) \end{aligned} \tag{1.c}$$

  D is the time to dynamically acquire a block of work, R_S and W_S are defined as above.

The above cases cover most common architecture/work-allocation combinations, with the exception of shared memory multiprocessors which use caches. The structure of each of the above equations is similar, suggesting that a single model with small architecture-specific variations might predict performance accurately.

Unfortunately, our search for suitable performance model is not yet finished. We have not yet considered vector hardware, another common feature designed to utilize parallelism. A vector system is composed of a bank of registers, each of which can hold an array of individual data elements, a set of heavily pipelined functional units, used to process register contents, and mechanisms which allow register contents to be rapidly read from and written to main memory. The number of elements which an individual register may contain, commonly called the *maximum vector length*, varies from 32 up to 1024 in most architectures.

Often, n, the length of each operand in a data-parallel computation, will exceed V, the maximum vector length. In such cases, the compiler generates code which processes the operands in runs of V elements; this technique is called *strip-mining* and is implemented via an outer loop which coordinates the enclosed vector operations. A reasonable approximation of the run time of a strip-mined vector computation, T_V, is

$$T_V = T_O + \lceil n/V \rceil T_L + nT_E \tag{2}$$

where

T_O is the one-time computation startup time,
T_L is the overhead of the strip-mining code and vector startup code, and
T_E is the vector execution rate, per element.

For large values of n, the contribution of the T_O term in Equation 2 becomes negligible, and we can approximate the ceiling term as linear, resulting in the following approximation of T_V for large n:

[5] Assuming an owner-computes rule, by definition, all writes will be to local storage.

$$T_V \approx n(T_L/V + T_E) \text{ (for large } n) \tag{3}$$

This means that, barring other contributions from other sources, we may be able to linearly interpolate the performance of large strip-mined vector operations from other measurements of large strip-mined operations. First, however, we must make one important modification to our model; effects which cause drops in vector performance, such as those due to the design of the memory hierarchy, must be accommodated. For example, the mean vector performance of the IBM 3090, as a function of n, is piecewise-linear [15], and depends upon the what fraction of the operands and destination register fits within cache memory. We can accommodate such effects by modifying Equations 2 and 3 such that T_E is a function of n.

Now, we combine our models to create a single model of the vector-parallel execution of the data-parallel computation $x = y \oplus z$. We define c, the execution rate of operation $\oplus$, per element, when operating to and from the register file or lowest-level cache, as

$$c = \frac{T_V - M}{n} \tag{4}$$

where M is overhead due to memory reads and writes. We assume that we can model M and T_E as piecewise linear functions of n which are continuous for large n, and thus, all architecture/compiler-specific derivations of Equation 1 are piecewise linear and continuous for large n, with the exception of Equation 1.a, which is piecewise linear excluding the factors which account for remote references.

Thus, we have a set of equations which model simple data-parallel operations as piecewise linear functions of the operand size. As shown in Section 7, approximations of these functions, interpolated from measurements sampled sparsely across the domain of array sizes, and combined with generic measurements of architectural characteristics, accurately predict the run times of several diverse test kernels.

6 Design of the Measurement Suite

This section details the design of the components of the Measurement Suite: the Array Operation Characterizer, Optimization Characterizer, and Architecture Characterizer. We used Saavedra's measurement suite as the Serial Operation Characterizer. The following constraints dictated the our design:

- **portability**: The system should be adaptable to all architectures/compilers with a limited number of changes, all of which should be well-documented and consistent. Additionally, the system should be able to measure a Fortran compiler which only supports some important subset of the Fortran 90 standard, such as array-extended Fortran 77 operations.
- **generality**: The measurement suite should produce measurements which can be utilized by a general performance prediction model.
- **tractability**: The system should run in a reasonable amount of time, precluding experiments which cover a large portion of the combinations of factors or measure the system at too high a resolution.

6.1 Array Operation Characterizer

The Array Operation Characterizer measures the average run time of each Fortran 90 array operation over a wide range of array sizes. The run time of a specific Fortran 90 array operation is a function of many factors, such as the:

- type of operation,
- size of the operands,
- distribution of the operands across the memory hierarchy,

Architecture	CM-5	Y-MP C90
Processors Tested	128	4
Processors Total	128	16
Processor Type	SPARC	custom
Memory Type	distributed	banked shared
Vector Hardware	yes	yes
Processor Clock Rate	32 MHz	400 MHz
Interconnection Network	fat tree	crossbar
Separate Host Processor	yes	no
Word Size	32 bits	64 bits
FP Single-Precision	32 bits	64 bits
Manufacturer	Thinking Machines Corp.	Cray Research

Table 1: Measured Architectures

- use of a mask or indexing array,
- other concurrent processing activity.

Ideally, we could vary each of the above factors independently over the set of commonly-encountered values, producing a large matrix of measurements covering the space of commonly-executed array operations. Unfortunately, such exhaustive testing would consume much time and violate our *tractability* constraint.

Instead, we measure the run time of each operation over a wide range of array sizes and fix all other factors to values we would encounter in the best case. That is, all operands are distributed ideally across the memory hierarchy,[6] no mask or indexing array is specified, memory is referenced in a regular and efficient manner, and there is minimal irregular computational activity on the system generated by other users. Optimistically, we assume that by using the results of separate tests, we can predict the effects of masks and indexing arrays, concurrently scheduled program segments, remote distributed operands in the case of distributed memory machines, and problems such as bank conflicts in the case of shared memory machines. See Section 7 for confirmation that these assumptions are reasonable.

Our suite measures the average and minimum run times of Fortran 90 array operations over a wide range of array sizes, from less than 100 to over 1 million elements. By interpolating these measurements and augmenting the results with architectural measurements, we can accurately estimate the execution time of an operation for *any* array size.

6.1.1 Measurements

We tested our suite on the following architecture-compiler pairs:

- CM-5/`cmf`: An 128-node Thinking Machines Corporation CM-5, using the `cmf` compiler.
- CM-5/`cmf-cmax`: An 128-node Thinking Machines Corporation CM-5, using the `cmf` compiler augmented with `cmax`, an optimizing source code translator.
- Y-MP C90/`cf77`: 4-nodes of a 16-node Cray Research Y-MP C90, using the `cf77` compiler.
- Y-MP C90/`f90`: 4-nodes of a 16-node Cray Research Y-MP C90, using the `f90` compiler.

Tables 1 and 2 describe each of the above architectures and compilers in greater detail.

The Array Operation Characterizer measures the run time of each relevant Fortran 90 array operation, over a wide range of array sizes; Appendix D describes each of the measured parameters, and Appendix E lists the specific run times of selected parameters for each compiler/architecture pair measured.

[6] *Ideally*, which in this case, means that each operand is distributed such that the aggregate access time is minimal.

Compiler/Translator	`cmf`	`cmax`	`cf77`	`f90`
Version	VecUnit 2.1.1-2	2.0	6.0	1.0
Full F90 Support?	yes	yes	no	yes
Optimization Flags	`-O`	-	`-Zp -Wd"-eabi6 -gl"`	`-O 3`
Auto Parallelization?	no	yes	yes	yes

Table 2: Measured Compilers

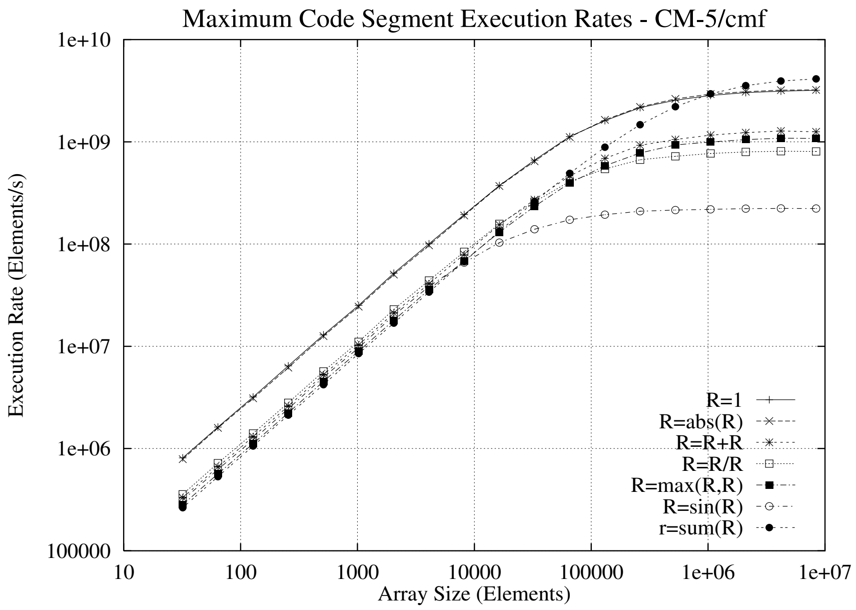


Figure 2: Code Segment Profiles - CM-5/`cmf` and CM-5/`cmf-cmax`

Analysis of the execution time of a given operation or code segment, as a function of the array size, provides valuable insights as to how a given architecture/compiler pair executes the operation. For example, Figure 2 shows the maximum observed run time of several code segments on the CM-5/`cmf` and CM-5/`cmf-cmax` pairs,[7] over a range of array sizes from 32 to over 8 million. We use the notation `{R}` to indicate a single-precision real array, and `{r}` to indicate a single-precision real variable. Note that all operands and destinations do not alias. Several characteristics become evident:

- The overhead of executing parallel operations is high, at least 40 μs per array operation, and the compiler parallelizes all array operations over the array sizes measured, irregardless of the efficiency of the resulting code.
- `{R}=abs({R})` executes almost as fast as `{R}=1`, indicating good hardware support for the absolute value operation.
- The profile of the execution rate of `{r}=sum({R})` differs from the profiles of the other code segments; for small array sizes, the execution rate is low, but for very large array sizes, it outperforms all other code segments. This performance reflects the nature of the `sum` computation - its overhead is large, since even for small array sizes, we must reduce each processor's local sum to calculate a single global sum; however, its peak performance rate is very high, since no writes to main memory are generated during the local sum computation.

[7] Operation measurements were not affected by `cmax`.

- For small array sizes, `{R}={R}+{R}` performs slightly worse than `{R}={R}/{R}`. Inspection of the assembler code generated for each code segment shows that the setup and termination code for array division is slightly more efficient than that for array addition, causing the discrepancy.

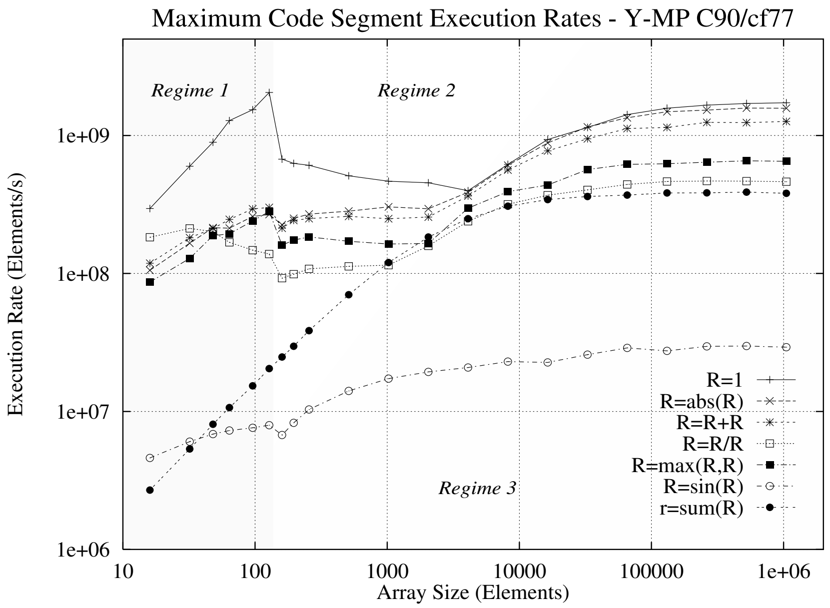


Figure 3: Code Segment Profiles - Y-MP C90/`cf77`

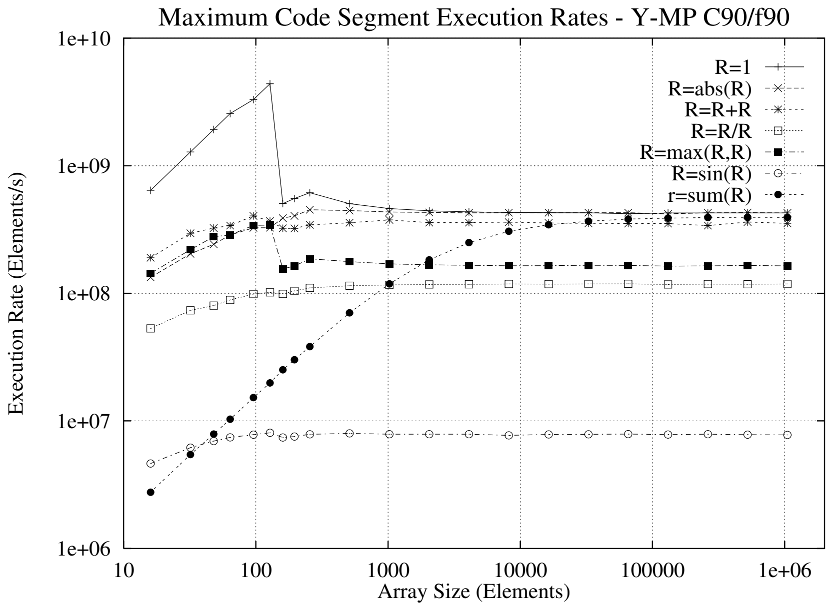


Figure 4: Code Segment Profiles - Y-MP C90/`f90`

Figure 3 illustrates the maximum observed execution rate of the same set of code segments upon the Y-MP C90/`cf77` pair. The following characteristics are evident.

- The performance of each code segment, with the exception of `{r}=sum({R})`, is composed of three major regimes. Regime 1 encompasses array sizes less than or equal to 128, Regime 2 includes array sizes greater than 128 and less than 128+x, where x is approximately linearly-related to the execution rate of the operation in question (except for slow operations where x is 0), and Regime 3 includes all larger array sizes. In Regime 1, the compiler generates single processor code, and each array operand fits completely within a vector register. In Regime 2, the compiler generates uniprocessing code, but vector operations are strip-mined, generating overhead which causes a drop in performance from the peak in Regime 1. Finally, in Regime 3, the compiler generates multiprocessing code which utilizes all four available processors and runs efficiently for large array sizes.

- A peculiar result is that array division has very high performance in Regime 1, which slowly decreases as the array size approaches 128. This result is repeatable, and we could not determine its cause, but theorize that it is related to unusual memory hierarchy or chaining behavior.
- In Regime 2, the performance of `{R}=1` decreases with increasing array size, most likely due to bank conflicts induced by the rapid rate at which elements are stored.

Finally, Figure 4 illustrates the maximum execution rate of the code segments upon the Y-MP C90/`f90` pair. The `f90` compiler does not generate multiprocessing code for most array operations, so the machine yields high performance where each array operand fits entirely within a vector register, and virtually equivalent performance elsewhere. This behavior is particularly noticeable in the performance profile for `R=1`; the time to store element is small relative to the stripmining overhead per element, so performance drops drastically when the array store is stripmined, for all array sizes over 128.

While the profiles shown above are instructive, we cannot adequately compare the overall performance of each architecture/compiler pair using similar graphs, due to the clutter induced by the large number of individual parameters involved. For purposes of simplification, we compute a set of 22 *reduced* parameters, each of which corresponds to the weighted average of a set of parameters which characterizes some architectural function. Table 3 lists each reduced parameter and the architectural function(s) represented. Appendix D contains the exact weighted average for each reduced parameter.

For presentation purposes, given a parameter T_n corresponding to the run time operation O (or the average of a group of operation run times) for array size n, we define the *simple-statement* value of T_n, $T+_n$, as the average execution time of a code segment composed of the operation O upon non-aliasing operands and storage in another non-aliasing destination. For example, given the parameter RS-ADD$_{10000}$, which represents the average run time of a data-parallel add of 10000-element single-precision floating-point array operands, RS-ADD+$_{10000}$ represents the run time of the code segment `a=b+c`, where `a`, `b`, and `c` refer to separate, non-aliased vectors.

For each reduced parameter Ri_n, we define Ni_n as the *normalized* reduced parameter, calculated as

$$\mathrm{N}i_n = \mathrm{R}i_n / C_i$$
$$\text{where } C_{15} = n^{\frac{3}{2}}, \text{ all other } C_i = n$$

Ni is simply the value of the reduced parameter Ri per element computed, except in the case of R15, which represents matrix multiplication; N15 represents the time required per multiply-and-accumulate operation used in the naive matrix multiplication algorithm. Figure 5 shows the mean values of the simple-statement normalized reduced parameters for an array sizes of 256 (2^8), 4096 (2^{12}), 65536 (2^{16}), and 1048576 (2^{20}). All matrix operations were measured on square arrays of size $\sqrt{n}$ by $\sqrt{n}$.

The Kiviat graphs in Figure 5 provide insights into strengths and weaknesses of each architecture/compiler pair. The CM-5/`cmf` pair performs very well for large array sizes, but falters when array sizes become small, due to its very large operation startup overhead. In contrast, the Y-MP C90/`cf77` pair performs in a much more balanced manner; each reduced parameter varies by less than a factor of ten over the measured values, as opposed to a factor of 1000 on the CM-5/`cmf` pair. The Y-MP C90/`f90` pair appears to have very balanced, efficient performance; however, this is due to the poor parallelization capabilities of the `f90` compiler, which, in most cases, utilizes one of the four available processors irregardless of array

Parameter	Operation/Function
R1	real SP addition
R2	real SP multiplication
R3	real SP arithmetic
R4	real SP F77 intrinsics
R5	real SP reductions
R6	real DP arithmetic
R7	real DP F77 intrinsics
R8	real DP reductions

Parameter	Operation/Function
R9	complex arithmetic
R10	complex F77 intrinsics
R11	complex reductions
R12	integer arithmetic
R13	integer F77 intrinsics
R14	integer reductions
R15	matrix multiplication
R16	other matrix operations

Parameter	Operation/Function
R17	type conversions
R18	comparisons
R19	logical operations
R20	logical reductions
R21	SP memory transfers
R22	DP memory transfers

Table 3: Reduced Parameters (*SP* and *DP* denote single and double precision, respectively.)

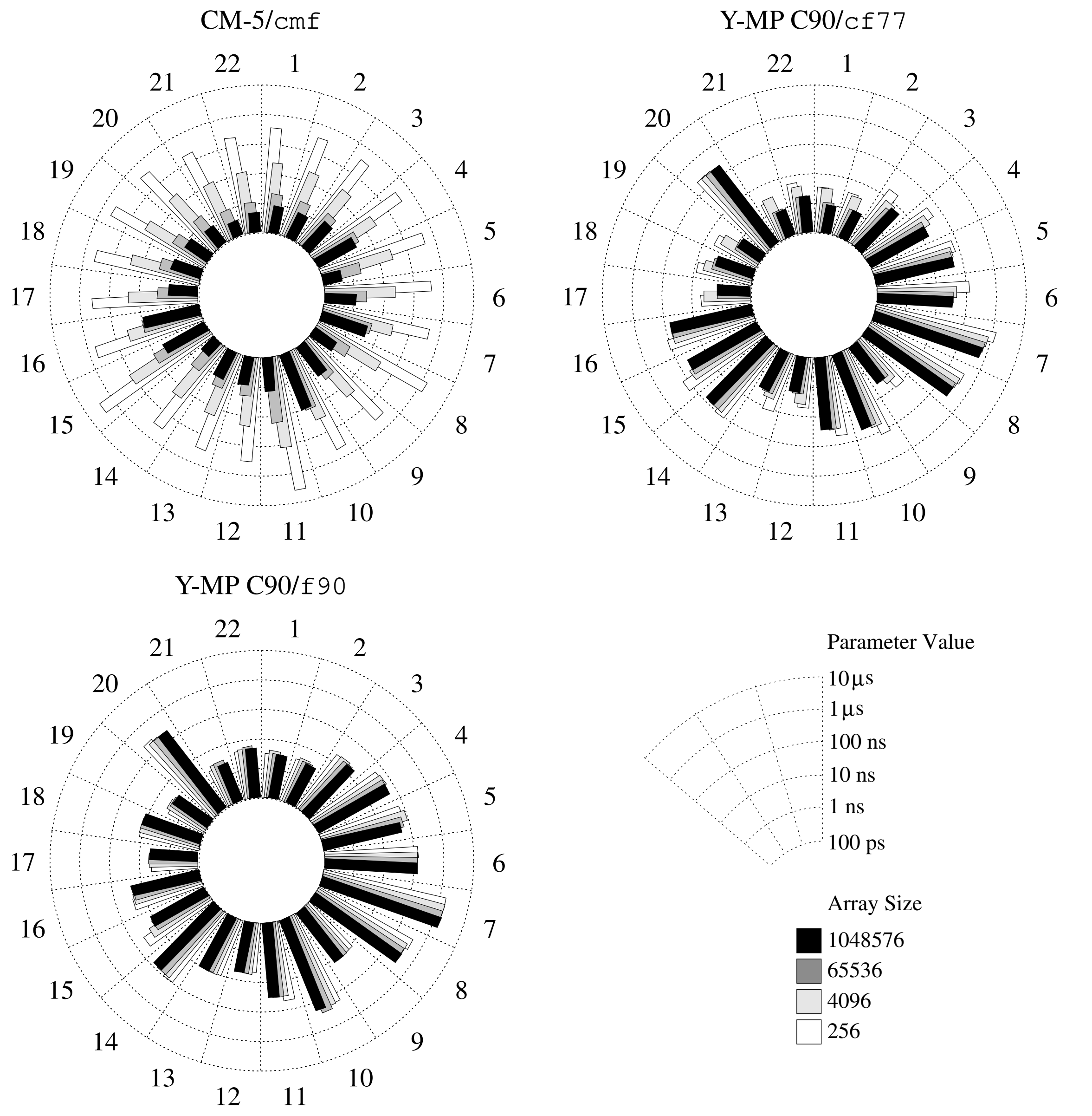


Figure 5: Normalized Simple-Statement Reduced Parameters, for Various Array Sizes

Naive	*Opt*

`j=0`
`do i=1,n`
`   j=j+1`
`   a(j)=a(j)+1.0`
`end do`
	`a=a+1.0`

Figure 6: *Naive* and *Opt* which determine whether induction variable elimination is applied.

size. A notable exception are matrix operations, which the Y-MP C90/`f90` pair multiprocesses and the Y-MP C90/`cf77` pair does not.

6.2 Optimization Characterizer

Compiler optimizations can radically change the execution time of a program; on many supercomputers, a serially-executed loop may execute several orders of magnitude slower than the corresponding vectorized or parallelized version. Several studies have investigated the ability of Fortran compilers to vectorize (e.g. [10]) and parallelize (e.g. [9]) a set of code segments selected in an ad-hoc manner; however, we want to determine the *specific* cases in which the compiler applies a given optimization, such that we can use this information to predict how a compiler will optimize a specific program.

Many of the optimizations we measure, such loop collapsing and subroutine inlining, will improve the execution time of a given code segment when it is executed serially. Many studies have examined optimizations in the serial domain (for example, [26]); we, however, are interested in the ability of the compiler to apply a given optimization, and then *parallelize/vectorize* the result. In the remainder of this section, when referring to a optimization O, we use the term "applies O" to mean that the compiler transforms the code using optimization O and then parallelizes/vectorizes the result, where possible.

We use a simple method to determine whether the compiler applies an optimization O. We create and time the execution of two code segments, which we call *Naive* and *Opt*. *Naive* is a code segment which is amenable to the optimization O, and *Opt* is a code segment which has a run time similar or equal to the optimized version of *Naive* - i.e. *Opt* is a source language optimized version of *Naive*. We design *Naive* so it is not amenable to any other optimization, and will run much slower than *Opt* if the compiler does not apply optimization O. If the run times of *Naive* and *Opt* are similar, we conclude that the compiler applies optimization O, otherwise, that it does not.

As an example of our optimization measurement strategy, suppose we want to see if the compiler applies induction variable elimination. Figure 6 shows the *Naive* and *Opt* code segments for this test. If the compiler does not apply induction variable elimination, the `do`-loop in the *Naive* segment must be executed serially, due to the data dependencies induced between loop iterations by the induction variable `i`. The run times of the two segments will differ significantly; the *Naive* segment will incur several cycles of loop and induction variable computation overhead per element processed, while the *Opt* segment will be parallelized, resulting in nearly linear speedup, and/or vectorized, resulting in an execution rate on a chained vector of slightly more than one cycle per element in the worst case. In contrast, if the compiler applies induction variable elimination, both the *Naive* and *Opt* code segments will be parallelized and their run times will be similar.

More formally, the measured mean run times of the code segments *Naive* and *Opt* are called t_{Naive} and t_{Opt}, respectively. Since our measurements are noisy due to various random variations in run times and to timer inaccuracies and resolution problems, (see Appendix A for details), our measured means are estimations of the actual means. We define the $100(1-\alpha)$-percent confidence intervals[8] of our mean

[8] The $100p$*-percent confidence interval* is the interval which contains the actual measurement with probability p.

measurements as $t_{Naive} \pm c_{Naive}$ and $t_{Opt} \pm c_{Opt}$, respectively, where the values of c_{Naive} and c_{Opt} are dependent upon the value of α, the number of observations, and the variance of our measurements. We define R as the ratio of t_{Naive} to t_{Opt}, and R_p as the lowest predicted ratio of t_{Naive} to t_{Opt} in the case where O was not applied to *Naive*.

An approximation of the $100(1-\alpha)^2$-percent[9] confidence interval of the measured value of R is

$$\left(\frac{t_{Naive} - c_{Naive}}{t_{Opt} + c_{Opt}}, \frac{t_{Naive} + c_{Naive}}{t_{Opt} - c_{Opt}}\right)$$

If the confidence interval of R is less than R_p, we conclude that the compiler applied O to *Naive* successfully, if the confidence interval of R is greater than R_p, we conclude that the was not able to apply O to *Naive*, and if the confidence interval of R includes R_p, we are not sure.

At first glance, it might seem easy to calculate a value of R_p, by timing *Naive* in the absence of optimization. Depending upon the compiler, however, disabling a given optimization O may disable other optimizations which may effect the execution speed of *Naive*, such as software pipelining, loop unrolling, and peephole optimization. Thus, we would compute an inflated value of R_p, which would be invalid, since R_p is defined as the *lowest* predicted ratio of t_{Naive} to t_{Opt} in the case where O was not applied to *Naive*. For example, on the Y-MP C90/`f90` pair, the run time of the *Naive* code in Figure 6 (where `n` is 1000000) is 0.56 seconds with all optimizations disabled, and 0.046 seconds with full scalar optimization enabled and all other optimizations disabled. Instead, we must select R_p solely with our knowledge of the characteristics of *Naive* and *Opt*, and the target machine type (some multiprocessor supercomputer). In most of our test cases, we assume that unoptimized *Naive* code will run at least twice as slow as the parallelized/vectorized code, and thus $R_p = 2$. Specific values of R_p are listed in Appendix E.

Some optimizations produce results which do not have a Fortran 90 analogue. For example, consider the case where the compiler substitutes efficient code when it encounters a *bin summation*, a computation which tallies the number of array elements which fall into each of a set of bins defined by some criteria. There is no way to specify a bin summation efficiently in Fortran 90, so our standard *Naive*/*Opt* code segment approach will not work. Instead, we compare the run time of a *Naive* code segment (defined as before) to the run time of a code segment called *Cripple*. *Cripple* is a version of *Naive* which has been slightly modified so that it cannot be optimized and has approximately the same run time as the unoptimized *Naive* segment. If the run time of *Naive* is significantly less than that of *Cripple*, we conclude that the optimization was applied, otherwise not.

6.2.1 Measurements

Table 4 summarizes our measurements; see Appendix F for a summary of each basic type of optimization. As expected, `cmf` performs poorly, since it is not an automatically parallelizing compiler. `cmf-cmax`, however, performs moderately well, although it generates suboptimal parallelized/vectorized code in some situations. `cf77` performs well, except for its surprising inability to apply the *code motion* and *dead code elimination* optimizations. `f90`, which was partially based on `cf77`, includes all optimization capabilities supported by `cf77`, with the exception of *subroutine inlining*. However, the quality of the optimized code generated by `f90` is inferior to that generated by `cf77`, since it is vectorized but not parallelized.

It is important to note that a very large number of factors, some of which are unpredictable, determine whether a compiler will apply an optimization to an amenable section of code; for example, limited internal table sizes may inhibit loop collapsing for very deep loops. Thus, the results of our tests should be used as heuristic guides to the performance prediction system, not as absolute indicators.

[9] This is a lower bound; the actual probability that $R = x/y$ will fall in the interval (a, b) is $\int_a^b \int_{-\infty}^{\infty} f_x(x) f_y(x/z)\,dx\,dz$, where f_x and f_y are the probability density functions of x and y, respectively. This integral becomes messy when f_x or f_y are Student's t distribution, and its solution is not particularly important. The lower bound we have calculated suits our needs adequately.

Optimization/Capability	cmf	cmf-cmax	cf77	f90
loop collapsing	no	**yes**	**yes**	**yes**
code motion	no	no	no	no
common subexpression elimination	partial	**yes**	**yes**	**yes**
dead code elimination	partial	**yes**	no	**yes**
forward substitution	no	**yes**	**yes**	**yes**
induction variable elimination	no	partial	partial	partial
subroutine inlining	no	no	partial	no
recurrence substitution	no	no	no	no
reduction substitution	no	partial	**yes**	**yes**
scalar expansion	no	**yes**	**yes**	**yes**
semantic analysis	no	no	**yes**	**yes**
dependency analysis	no	**yes**	**yes**	**yes**
idiom recognition	no	partial	**yes**	**yes**

Table 4: Summary of Compiler Optimization Capabilities

6.3 Architecture Characterizer

The Architecture Characterizer (AC) measures various parameters of the *compiler-constrained* architecture, which is the actual architecture as visible through executable code generated the compiler. The compiler-constrained architecture will have some subset of the features of the actual architecture and will perform in a manner equivalent to or slower than the actual architecture.

The following sections explain the quantities measured by each module composing the AC, and the models we use to predict run time; note that each heading indicates the name of the test, followed by its abbreviation. For specific descriptions of the parameters measured, see Appendix D, and for details of the mechanics of specific tests, see Appendix B.

6.3.1 Memory Hierarchy Access Characteristics - `access`

The `access` test measures remote or shared-memory read and write latencies, as well as gather and scatter overhead, per element, and overhead due to random accesses in a shared memory system.

Ideally, the `access` test would characterize the average access time between each processor/processor pair (in a distributed memory system) or each processor/memory location pair (in a shared memory system). Unfortunately, it is very difficult to write a Fortran 90 benchmark which produces such a characterization, since we have no control over the mapping of data and computation to processors. Instead, we measure the latency of the *average* remote (or shared-memory) read or write; as a result, the flatter the communications network in question, the better we characterize the latency of a random remote or shared memory access. Several recent supercomputer architectures have communications networks which appear to be flat; processor to processor latencies over the CM-5 fat tree range from 3 to 7 microseconds [3], and the Y-MP C90 provides fairly uniform access times to its shared memory. However, on machines such as the T3D, which provide hardware to accelerate block transfers and prefetch data [4], our characterization might cause significant errors in the performance prediction of programs which make heavy use of such features.[10]

On some systems, serial code may be executed on a host processor, and this code may reference data distributed across the processing nodes (or stored in the shared memory). Often, the host processor is attached to the interconnection network in a suboptimal manner, and the latency of accesses from the host processor to the processing nodes (or shared memory) may differ significantly from the latency of accesses

[10]Future versions of our measurement suite should characterize the average overhead for several different *types* of remote accesses (random single element accesses, block transfers, etc.).

between processing nodes (or between processing nodes and shared memory). To account for the potential disparity, we include simple tests which measure the latencies of array references issued by serial code. We term this type of access a "host remote access".

6.3.2 Stride Effects - `stride`

The `stride` test measures the effect of the stride of memory accesses in a given computation upon performance. We account for stride effects as overhead incurred upon each load. We define the following quantities:

T_S, the predicted run time of a stride s load from (or store to) an m-element array.
T_L, the measured run time of a stride 1 load from (or store to) an $\frac{m}{s}$ element array.
$O_{i,n}$, the measured stride-induced overhead per element of a stride i reference to an n-element array.

We predict T_S as follows:

$$T_S = T_L + \frac{m}{s} O_{s,m}$$

where $O_{s,m}$ is defined by one of the two following models, and measured accordingly.

Cache Model:

Depending upon the stride, the size of the array operands, and the characteristics of the caches(s), the execution rate of a given operation will vary widely. By measuring and analyzing of the execution rate over a wide range of array and stride sizes, we can reconstruct the parameters of the cache, and we can use these parameters to predict the effect of the cache upon the run time of a given program. We measure $O_{i,n}$ over a sparse range of array sizes and strides, using the method described in [14].

Three problems arise when attempting to measure cache performance on a parallel architecture, using a Fortran 90 benchmark similar to that described in [14]:

- *Inability to restrict measurements to a single node*: The Fortran 90 benchmark is parallelizable. In [16], Saavedra measures the cache performance of the KSR-1, but he does so by timing a single node using an assembly language benchmark, which violates our portability constraint.
- *Operating system-induced effects*: On a system which lacks a virtually-addressed cache, the operating system may map adjacent pages into conflicting physical addresses in the cache, producing unpredictable behavior.[11]
- *Other effects*: Since we cannot restrict our measurements to a single node, we must use the aggregate performance of the system to infer cache characteristics. Unfortunately, it is difficult to factor out other effects from our measurements, such as those due to vector processing, parallel operation startup time, and poor compiler support.

As a result, our measurements may be too noisy to infer the exact organization of the memory hierarchy. For example, the `cmf` compiler produces executable code in which the computation `a(1:n)=a(1:n)+b(1:n)` runs faster than the corresponding non-unit stride computation `a(1:n:`s`)= a(1:n:`s`)+b(1:n:`s`)`, even though much less work is performed in the latter computation.

Since we cannot confidently derive memory hierarchy characteristics, we must approximate overhead for unmeasured strides and array sizes by linearly interpolating from measured values.

[11] Saavedra overcame this problem by rebooting before each measurement, but this approach is not practical when measuring heavily-utilized supercomputers.

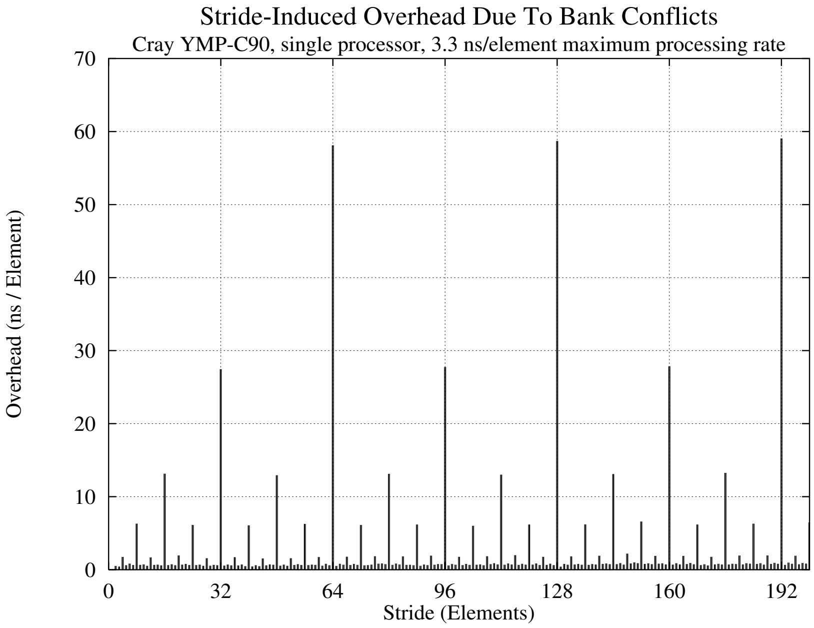


Figure 7: Stride-Induced Overhead Due To Bank Conflicts On The Cray YMP-C90

Bank Model:

Banked memory consists of a set of N individual memory systems, or *banks*, each of which is linked together to form a larger memory, such that address i in the large memory is address i/N in bank i mod N. Successive references to banked memory can proceed rapidly as long as the interval between two references to a given bank is greater than the cycle time; otherwise, the two references will *conflict* and the computation will stall until the bank can service the request. If we have N banks, and we reference memory with stride k, we will utilize $\lceil \frac{N}{gcd(N,k)} \rceil$ banks, and the others will remain idle. Thus, if the cycle time is T_C, the access time is T_A, and the additional overhead per element is T_O,[12] the time to reference an element, c, including overhead, is

$$c = \max(T_A, T_C - (T_A + T_O)\lceil \frac{N}{\text{gcd}(N,k)} \rceil)$$

In most systems, the number of banks is a power of two and $T_A \ll T_C$; therefore, all computations with large power of two strides perform at much less than peak efficiency.

For example, Figure 7 shows the stride-induced overhead due to bank conflicts, on a single processor of a Cray Y-MP C90, for a simple array operation involving a load, arithmetic operation, and store. The unit stride operation ran at a rate of 3.3 ns per element. We can infer from the figure that 64 banks are referenced, and that the cycle time of each bank is approximately half the maximum overhead (the operation in question contains a load and a store), 30 nanoseconds.

We measure B_s, the overhead per element induced by an s-element stride, for all power of two values of s less than or equal to M, where M is the power of two stride after which successive measured values of B do not significantly increase. Since bank effects do not vary with array size, for any stride i or array size n not measured, $O_{i,n} = B_{\text{gcd}(i,N)}$.

[12]Such additional overhead might be due to logic in the vector load/store module.

6.3.3 Distributed Memory Array Layout - `layout`

The `layout` test measured quantities which allow us to determine if the machine measured is a shared memory or distributed memory system, as well as the default array layout in a distributed memory system.

6.3.4 Mask Effects - `mask`

The `mask` test measures the effects of a mask upon a computation. We define the following quantities:

T_A, the measured run time of the unmasked computation A,
$T_R(r)$, the measured run time of the masked computation `where(`X`)` A,
where a fraction r of the elements in mask X are true, and the true elements are randomly distributed,
n is the number of elements in the mask X.

We predict T_M, the run time of an m-element masked computation M in which a fraction x of the elements in the mask are true, using one of the following models, assuming we have measured T_U, the run time of the corresponding unmasked computation U.

Partial Model:

Element i of the destination array is calculated and stored only if element i in the mask is true. All other calculations are skipped:

$$T_M = xT_U + \frac{m(T_R(x) - xT_A)}{n}$$

Full Model:

The entire result of the masked computation is computed as it would be in the corresponding unmasked computation, and results are stored in a temporary array. Subsequently, the elements corresponding to true elements in the mask are copied from the temporary array to the destination array:

$$T_M = T_U + \frac{m(T_R(x) - T_A)}{n}$$

We use the *full* model when our measurements show that $T_R(r) > T_A$; otherwise, we use the *partial* model. Unmeasured values of $T_R(r)$ are linearly interpolated from measured values.

6.3.5 Block Effects - `block`

The `block` test measures the difference between the run time of a simple array load/store and a *block* load/store, where a *block* is defined as an array section with variable bounds or constant bounds which do not correspond to the beginning or end of the source array. Efficient code can be generated for constant-bound block loads/stores, so the `block` test also includes a test which determines whether block overhead is incurred in such cases. We define several quantities:

T_B, the predicted run time of an n-element block load from (or store to) an m-element array.
T_L, the measured run time of a stride 1 load from (or store to) an n-element array.

We predict T_B per one of the following models.

Partial Model:

We model overhead due to a block load as a function of the number of elements loaded:

$$T_B = T_L + nX_n$$

where X_n is the measured overhead per element loaded.

Full Model:

We model overhead due to a block load as a function of the size of the source array and the percentage of elements loaded:

$$T_B = T_L + mY_{\frac{n}{m}}$$

where Y_p is the measured overhead per element in the source array, where $100p$-percent of the elements are loaded. In both models, unmeasured values of overhead are linearly interpolated from measured values.

6.3.6 Measurements

Table 5 shows the derived values of compiler-constrained architecture-specific parameters for both architectures. Two interesting characteristics of the CM-5 emerge:[13]

- The overhead of a gather/scatter is more than 10 times the transfer rate, indicating poor compiler/architecture support for these operations.
- A host remote read[14] costs a factor of 5 more than a normal remote read, and a sequential remote write costs a factor of 10 more. Obviously, the CM-5 host processor is shoe-horned onto the communications network quite sub-optimally.

Function/Parameter	CM-5/`cmf`
Memory Transfer$*$	34.6 ns/real
Gather Overhead$*$	540. ns/real
Scatter Overhead$*$	476. ns/real
Remote Read Latency	10.7 μs/real
Remote Write Latency	3.03 μs/real
Host Remote Read Latency	59.6 μs/real
Host Remote Write Latency	34.6 μs/real
Default Array Layout	blocked
Mask Effect Model	full
Stride Effect Model	cache
Block Effect Model	full

Function	Y-MP C90/`cf77`	Y-MP C90/`f90`
Memory Transfer$*$	4.83 ns/real	2.48 ns/real
Gather Overhead$*$	0.79 ns/real	0.73 ns/real
Scatter Overhead$*$	7.32 ns/real	0.55 ns/real
Random Read Overhead$*$	4.41 ns/real	2.69 ns/real
Random Write Overhead$*$	17.6 ns/real	5.11 ns/real
Serial Read/Write Overhead	negligible	negligible
Mask Effect Model	full	full
Stride Effect Model	bank	bank
Block Effect Model	partial	partial

Table 5: Summary of Derived Architectural Parameters ($*$ = normalized to a single processor; for the purpose of measurement normalization, we consider the Y-MP C90/`f90` pair to be a uniprocessor, since we know `f90` generates uniprocessor code for our test cases.)

Similarly, we can draw the following conclusions from the comparison of derived parameters of the Y-MP C90/`cf77` and Y-MP C90/`f90` pairs:

- The overhead of the scatter operation on the Y-MP C90/`cf77` pair is much larger than that on the Y-MP C90/`f90` pair, indicating that `cf77` supports the scatter operation poorly.
- Since the Y-MP C90/`cf77` pair multiprocesses our benchmarking code while the Y-MP C90/`f90` pair generally does not, a greater number of bank conflicts are induced on the former system, resulting in a slower memory transfer rate and higher overhead due to sets of reads/writes to random locations.

[13] Use of `cmax` did not affect architectural results, so the CM-5/`cmf` results are identical to the CM-5/`cmf-cmax` results.

[14] As defined in Section 6.3.1, a *host remote access* is an access from or to the parallel nodes from the host processor.

7 Verification of Predictive Capabilities

In this section, we present our prediction method and demonstrate that the results produced by our Measurement Suite are useful for performance prediction.

7.1 Prediction Method

We have not yet fully automated the run-time prediction process, such as was accomplished for scalar programs in [8]. Instead, here we predict the run time of all kernels by hand using the following method. To predict the one-time run time of an array operation A, we first predict the run time of the *basic* array operation B. The basic array operation B is defined as the unmasked operation in which operands are manipulated in the same manner as in A. Given that array operands and results in A are n-element sections, the operands and results in B are n-element *arrays* aligned optimally across the system. We replace reduction calculations with scalars, and transformation operations with array loads. For example, if

A : `where (a) b(1:j:i)=sqrt(c(n:m)*(0.5+d(e(f(k:l)))))`, then
B : `s=sqrt(t*(0.5+u(v(w))))`

We define the run time of B as T_B. We calculate T_B as

$$T_B = \sum D_i R_i = D \cdot R$$

where D_i is the number of times operation i is executed in one execution of B, and R_i is the run time of operation i. We then add overhead terms to T_B as follows:

1. The overhead of any embedded reductions or transformation operations in A is computed recursively via the method being described, and added.
2. We analyze each array operand and result in A, and if the operand/result contains a gather/scatter or non-unit stride, we add the overhead per the appropriate model. Additionally, for each block load/store, we compare the block parameters (start, finish, stride) to the blocks we have already considered, and if they are different, we add block overhead for the load/store.
3. When the system is shared memory, for any array operands which are the result of a gather, we add random read/write overhead if the indexing arrays do not contain linearly-related elements.
4. When the system is distributed memory, we assume an owner computes rule. We determine the number of remote reads and writes by combining our knowledge of the default array layout, our assumption of an owner-computes rule, and the values of the indexing arrays used to gather/scatter and the bounds/stride of any array sections.
5. If the computation is masked, we add mask overhead per the appropriate model, and add the run time of the mask computation.

We predict the run time of a program P on a compiler C and architecture M as follows:

1. We convert all sequential constructs in P parallelizable by C into their Fortran 90 array forms, yielding a new program Q.
2. We count the number of times each line is executed in Q, and we determine the values of the masks, indexing arrays, and array section parameters involved in each execution of any array operation.
3. We calculate the run time of serial code (code which is not an array operation) on a given line using Saavedra's method, with the exception that the latency of array accesses is represented using the host read and write parameters when we determine that a host processor is present. We calculate the total run time contribution of an array code segment S as the sum of the computed one-time run time of S for each execution, per the known values of any involved masks, indexing arrays, etc.

7.2 Test Kernels

We used the following five kernels to test our prediction system:

- ave, a Fortran 90 kernel which tests for errors produced by floating point roundoff. ave calculates the sum of values in an array, repeatedly applies an averaging function to the array's contents, and then calculates the new sum of the values in the array. The kernel has high potential for efficient parallel execution, since each step involves a data parallel operation or reduction on one or more large well-aligned array operands; however, the compiler must recognize and parallelize the sum computations, which are expressed sequentially.
- merge, a Fortran 90 implementation of Batcher's odd-even merge sort [1]. The heart of the kernel contains both masked and unmasked gather operations, as well as a large quantity of array integer bit manipulations. Four 32768-element real arrays are sorted into ascending order.
- ep, the slightly modified version of the sample Fortran 90 implementation [19] of the Embarrassingly Parallel Kernel of the NAS Parallel Benchmarks [18]. Using a parallelizable random number generator, the kernel generates a large number of random two dimensional coordinates in a Gaussian distribution about the origin, and tallies the number of coordinates which fall into each of a set of concentric square regions about the origin. All array operations involve 8192-element REAL*8 arrays.
- multi, a kernel which solves for the potentials induced in a square cavity with fixed boundary conditions, using a multigrid implementation of Jacobi relaxation. We represent the potential in the cavity as a 1024 by 1024 grid. No convergence criterion is examined.
- tomcatv, a Fortran 77 SPEC kernel which performs mesh generation using Thompson's Solver [27]. The code consists of a series of singly and doubly-nested loops, all of which are parallelizable and/or vectorizable with varying degrees of difficulty.

Appendix G contains more detailed information about each kernel, as well as the equations used to calculate predicted run times.

7.3 Predicted Run Times

Table 6 shows the observed run time of each kernel, the predicted run time, and the error of each prediction. We report the predicted run time calculated using parameters derived from both mean *and* minimum measurements.[15] We wanted to see if our minimum measurements, which are largely noiseless, produced better predictions than mean measurements; however, in general, the two methods seemed to produce equally accurate predictions. Figure 8 plots the predicted mean run time versus the actual mean run time of each kernel (the mean run times of the tomcatv kernel on the CM-5/cmf and CM-5/cmf-cmax pairs are omitted due to their large run time). Note that there are no clear trends, at least that we can see, either in general or relative to each architecture, in the deviation of our predictions from actual measurements.

We observe that the accuracy of our predictions is far better than can be obtained using Saavedra's method for scalar programs, even under optimistic assumptions.[16] Table 7 compares the run time predictions generated by the two methods for the Y-MP C90/cf77 pair. Saavedra's method measures Fortran 77

[15] Due to the noise in our mean measurements and our linear parameter estimation methods, the predicted mean run time of a program may be less than its predicted minimum run time. For example, consider that parameter X is derived from measurements A and B as $X = A - B$. If the measured minimum of A is 4.0 and B is 3.0, the estimated minimum value of X is 1.0. If the measured mean of A is 4.5 and B is 4.0, respectively, the estimated mean value of X is 0.5, which is less than the estimated minimum value. Thus, for any program which has a predicted run time which heavily depends on X, the predicted mean run time might be less than the predicted minimum run time.

[16] When calculating run times using Saavedra's method, we optimistically assume (a) that array constructs are decomposed into heavily unrolled loops which generated negligible loop and index arithmetic overhead, and (b) that the run time of the IEOR, ISHFT, and IAND intrinsics on local variables is equivalent to the run time of a logical operation on local variables. We ignore index arithmetic overhead for sequential constructs.

Program	Architecture/Compiler	Mean Run Time (s)			Minimum Run Time (s)		
		Predicted	Actual	Error (%)	Predicted	Actual	Error (%)
`ave`	CM-5/`cmf`	61.6	57.7	+6.9	61.1	57.0	+7.2
	CM-5/`cmf-cmax`	1.97	1.35	-16.1	1.97	1.29	-14.0
	Y-MP C90/`cf77`	1.32	1.79	-26.6	1.14	1.19	-3.9
	Y-MP C90/`f90`	5.10	4.75	+7.3	4.95	4.65	+6.3
`merge`	CM-5/`cmf`	5.46	4.77	+14.5	5.39	4.71	+14.5
	CM-5/`cmf-cmax`	0.924	0.916	+0.9	0.899	0.862	+4.3
	Y-MP C90/`cf77`	0.195	0.190	+2.8	0.138	0.148	-7.1
	Y-MP C90/`f90`	0.510	0.703	-27.4	0.457	0.689	-33.4
`ep`	CM-5/`cmf`	0.947	1.34	-29.4	0.946	1.33	-29.1
	CM-5/`cmf-cmax`	0.947	1.31	-27.9	0.946	1.30	-27.0
	Y-MP C90/`cf77`	0.401	0.294	+36.6	0.302	0.200	+51.4
	Y-MP C90/`f90`	0.917	0.915	+0.2	0.824	0.903	-8.7
`multi`	CM-5/`cmf`	16.0	25.5	-37.2	15.8	25.5	-37.6
	CM-5/`cmf-cmax`	16.0	25.3	-37.4	15.8	25.3	-37.5
	Y-MP C90/`cf77`	1.55	1.01	+53.0	1.11	0.993	+11.4
	Y-MP C90/`f90`	4.26	1.12	+280.	4.38	1.07	+310.
`tomcatv`	CM-5/`cmf`‡	16600	> 7200.	–	16500	> 7200.	–
	CM-5/`cmf-cmax`	838.	966.	-13.3	831.	958.	-13.3
	Y-MP C90/`cf77`	1.36	0.980	+38.8	1.28	0.950	+34.5
	Y-MP C90/`f90`	3.16	3.38	-6.5	3.29	3.33	-1.3

Table 6: Predicted And Actual Kernel Run Times (‡ = The run time of `tomcatv` on the CM-5/`cmf` pair exceeded the 120 minute job limit)

Kernel	Predicted Mean Run Time (s)		Actual Mean Run Time (s)
	Saavedra's Method	Our Method	
`ave`	87.5	1.32	1.79
`merge`	3.04	0.195	0.190
`ep`	5.80	0.395	0.294
`multi`	12.5	1.55	1.01
`tomcatv`	16.1	1.36	0.980

Table 7: Comparison of Prediction Methods for the Y-MP C90/`cf77` Pair

operations, often contained in nonvectorizable/unparallelizable loops, and does not include measurements of Fortran 90 array operations. In most cases, array operations are heavily vectorized/parallelized, and execute in a fraction of the time required by their unparallelized/nonvectorized Fortran 77 equivalents. Thus, Saavedra's system grossly overestimates the run time of programs which contain a large number of array operations.

Unfortunately, one prediction, the run time of the `multi` kernel on the Y-MP C90/`f90` pair, is poor. Our suite measures the peak run time of array operations upon one-dimensional array operands, since the involved computations should be the easiest to parallelize. The `f90` compiler, however, generates code which uniprocesses one-dimensional array operations, irregardless of the size of the operands, while parallelizing some two-dimensional array operations. The run time of the `multi` kernel on a single processor of the Y-MP C90/`f90` pair has a mean of 3.87 seconds and minimum of 3.84 seconds, which matches our prediction. Future versions of our suite should include tests which remedy this problem.

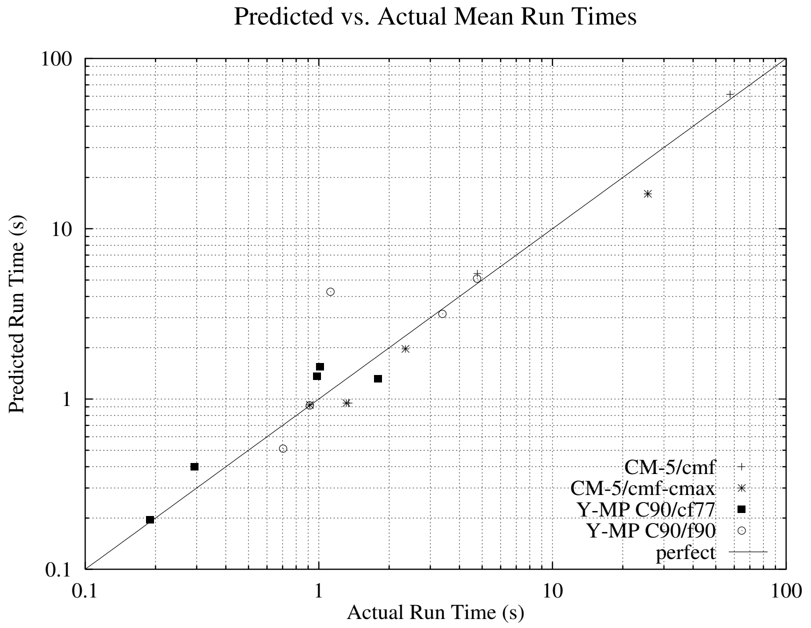


Figure 8: Predicted Versus Actual Mean Run Time of Test Kernels

8 Conclusions and Future Work

As we have shown, we can accurately predict the run time of many Fortran 77 and Fortran 90 programs via a simple combination of serial and array operation execution rates, measurements of compiler optimizations, measurements of program operation execution frequency, and generic architectural measurements.

Since it was intended as a prototype, our measurement suite is not complete. For example, we do not measure the execution rates of some of the more obscure Fortran 90 operations, and no doubt, our architecture measurements can be enhanced to yield results which better characterize the average case. Furthermore, the augmentation of compiler tests will improve the accuracy of our guesses as to which parallelizing/vectorizing optimizations are applied. No doubt, as new optimization technology is introduced to production-quality compilers, such as the pipelining optimizations [24] found in the experimental Fortran D compiler [23], our tests and models will need to be adjusted.

We designed our measurement suite with the goal of creating an automatic performance prediction system. To complete the system, two major components must be implemented: the *Instrumentor*, which adds code to a Fortran 90 program to collect the required dynamic execution measurements, and the *Run Time Predictor*, which combines the original program source, dynamic execution measurements, and machine characterization to produce a run time prediction. The design and implementation of each of these programs will be challenging; the Instrumentor requires efficient statistic collection and storage methods, and the Run Time Predictor requires extensive optimization analysis capabilities (similar in some regards to a production Fortran 90 compiler, sans code generation capabilities). However, the reward for this effort will be a portable automatic performance prediction system which accurately predicts the run time of Fortran 90 programs running on supercomputers.

9 Acknowledgements

I would like to thank Dr. Rafael Saavedra for his permission to use his measurement suite, which proved to be quite valuable. Additionally, we borrowed and extended many of his ideas regarding data presentation.

References

[1] K. Batcher. "Sorting Networks and Their Applications." *Proceedings of the AFIPS Spring Joint Computing Conference*, v. 32, pp. 307-314, 1968.

[2] W. Oed. "Cray Y-MP C90: system features and early benchmark results." *Parallel Computing*, v. 18, n. 8, pp. 947-954, 1992.

[3] W.D. Hillis and L.W. Tucker. "The CM-5 Connection Machine: A Scalable Supercomputer." *Communications of the ACM*, v. 36, n. 11, pp. 30-40, 1993.

[4] R.E. Kessler and J.L. Schwarzmeier. "Cray T3D: A New Dimension For Cray Research." COMPCON Spring '93. Digest of Papers. San Francisco, CA, February 22-26, 1993.

[5] R.G. Convington, S. Madala, V. Mehta, J.R. Jump, and J.B. Sinclair. "The Rice Parallel Processing Testbed" *Proceedings of the 1988 ACM SIGMETRICS Conference on the Measurement and Modelling of Computer Systems*, pp. 4-12, 1988.

[6] D. Menasce and L. Barroso, "A Methodology for Performance Evaluation of Parallel Applications on Multiprocessors" *Journal of Parallel and Distributed Computing*, v. 14, pp. 1-14, 1992.

[7] M. Hall, S. Hiranandani, K. Kennedy, and C. Tseng, "Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines". *Proceedings of Supercomputing '92*, pp. 522-534, 1992.

[8] R. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking.* Ph.D. Thesis, University of California at Berkeley, February 1992.

[9] J. Dongarra, Furtney, Reinhardt, Russell. "Parallel Loops - A test suite for parallelizing compilers: Description and example results." Parallel Computing, v. 17, n.10-1, pp. 1247-1255, 1991.

[10] D. Levine, D. Callahan, J. Dongarra. "A Comparative Study Of Automatic Vectorizing Compilers." Parallel Computing, v. 17, n.10-1, pp. 1223-1244, 1991.

[11] D. Padua and M. Wolfe. "Advanced Compiler Optimizations For Supercomputers" *Communications of the ACM*, v. 29, n 12, pp. 1184-1200, 1988.

[12] M. Metcalf and J. Reid. *Fortran 90 explained*. Oxford [England] ; New York : Oxford University Press, 1990.

[13] ANSI-X3.198 - Programming Language - Fortran - Extended, American National Standards Institute, 1992.

[14] R. H. Saavedra-Barrera and A. J. Smith, *Measuring Cache and TLB Performance and Their Effect of Benchmark Run Times*. Technical Report CSD-93-767, Computer Science Division, University of California at Berkeley, 1993.

[15] W. Schonauer and H. Hafner. "Performance estimates for supercomputers: The responsibilities of the manufacturer and of the user." *Parallel Computing*, v. 17, pp. 1131-1149, 1991.

[16] R. H. Saavedra, R.S. Gains, and M.J. Carlton. "Micro Benchmark Analysis Of The KSR1." *Proceedings SUPERCOMPUTING '93*. SUPERCOMPUTING '93, Portland, OR, 15-19 Nov. 1993.

[17] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1993.

[18] D. Bailey, E. Barszcz, et. al. *The NAS Parallel Benchmarks*. RNR Technical Report RNR-94-007, Applied Research Branch, Numerical Aerodynamic Simulation Group, NASA Ames, 1994.

[19] Sample Fortran 90 and HPF versions of the NAS benchmarks, written by D. Sarafini, are available at URL "`http://www.nas.nasa.gov/Projects/NPB/hpf-f90-NPBs.tar`"

[20] G. Florin, C. Fraize, and S. Natkin. "Stochastic Petri Nets: Properties, Applications, and Tools." *Microelectronics and Reliability*, v. 31, n. 4, pp. 669-697, 1991.

[21] M. A. Marsan, G. Balbo, et. al. "An Introduction to Generalized Stochastic Petri Nets." *Microelectronics and Reliability*, v. 31, n. 4, pp. 699-725, 1991.

[22] W. Blume, R. Eigenmann. "Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs." *IEEE Transactions on Parallel and Distributed Systems*, v. 3, n.6, pp. 643-656, 1992.

[23] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, U. Kremer, et. al. "An overview of the Fortran D Programming System." Languages and Compilers for Parallel Computing. Fourth International Workshop. Santa Clara, CA, USA, 7-9 Aug. 1991.

[24] S. Hiranandani, K. Kennedy, C. Tseng. "Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines." *1992 ACM International Conference on Supercomputing*, Washington, DC, 19-23 July 1992.

[25] High Performance Fortran Language Specification Version 1.0, Draft. Tech Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Jan. 1993.

[26] R. Saavedra-Barrera and A. Smith. *Performance Characterization of Optimizing Compilers.* UCB CSD-92-699, Computer Science Division, University of California at Berkeley, August 1992.

[27] K. Dixit. "New CPU benchmark suites from SPEC." *COMPCON Spring 1992.* Thirty-Seventh IEEE Computer Society International Conference, San Francisco, CA, 24-28 February 1992.

[28] M. Parashar, S. Harari, T. Haupt, G. Fox. "Interpreting the Performance of HPF/Fortran 90D." *Proceedings of Supercomputing '94*, Supercomputing '94, Washington, D.C., 14-18 November, 1994.

Appendix A. Timing Method

To measure each code segment, we utilize timing code similar to that shown in Figure 9. The timing code performs three tasks in sequence:

1. Iteratively determines the number of times we must execute the code segment to reduce the error in our measurements to an acceptable level.
2. Measures a specified number of samples of the execution time of the code segment.
3. Calculates the mean and variance of the samples.

More specifically, to measure the average execution time S of a code segment X, we measure the run time of a loop which contains X, plus data dependency-inducing code which reference operands present in X. This loop iterates N_{body} repetitions. Given that T is the execution time of this loop, C is the timing overhead, and L is the loop overhead, we know that B, the execution time of the body of the loop, is

$$B = (T - C)/N_{body} - L$$

Furthermore, S, the execution time of X, is

$$S = B - D$$

where D is the overhead of the dependency-inducing code included with X. Dependency-inducing code is required to prevent the compiler from applying transformations which eliminate or destructively modify X

```
good=.false.
do ii=1,nobs
   time=0.0
   do while (.not.good.or.time.eq.0.0)
      start=stimer()
      do i=1,nbody

      [ code segment to be measured ]

      [ optimization thwarting dependencies ]

      end do
      finish=etimer()

      time=finish-start
      if (time.ge.clockres*1.0/error) good=.true.
      if (.not.good) then
         nbody=nbody+f(time,clockres,error)
      endif
   end do

   sample(ii)=time/nbody
end do

mean=calcmean(sample,nobs)
variance=calcvariance(sample,nobs)
segmentmean=adjustmean(mean,nbody,...)
segmentvariance=adjustvariance(variance,nbody,...)
```

Figure 9: Skeleton of the Timing Code

or the loop surrounding it. For example, consider that we want to measure the run time of the code segment `a=b+c`. The inner timing loop, sans dependencies, is

```
do i=1,n
   a=b+c
end do
```

Assuming that `a`, `b`, and `c` do not refer to the same storage, the compiler can eliminate the outer loop and maintain semantic equivalence.

Typically, the dependency-inducing code consists of an assignment to a random element of each of the operands, followed by an accumulation from a random element of the result. For example, to prevent the compiler from destructively modifying the above example, we insert the following dependency-inducing code into the loop

```
b(mod(irnd(),n)+1)=rnd()
c(mod(irnd(),n)+1)=rnd()
sum=sum+a(mod(irnd(),n)+1)
```

We print the value of `sum` at the end of each procedure containing such dependencies. We measure the overhead of the the dependency-inducing code using a method similar to that which we employ to measure *S*.

Given clock resolution R, we can accurately model the error introduced into T by the timing routines as a uniformly distributed random variable over the interval $(-R, R)$. Thus, the mean and variance of S are

$$\hat{S} = (\hat{T} - \hat{C})/N_{body} - \hat{L} - \hat{D}$$

$$\text{Var}(S) = (\text{Var}(T) + \text{Var}(C))/N_{body}^2 + \text{Var}(L) + \text{Var}(D) + R^2/3N_{body}^2$$

We select the number of observations, N_{obs}, as 35 or more so that we may utilize the normal distribution as an approximation of Student's t distribution in all confidence interval calculations. N_{body} is determined by an iterative process, such that if Var(S) is small relative to S, it is likely that the clock error will increase p-percent confidence interval of S by a maximum of FS on either side, assuming that the clock error's contribution to Var(D) is low. Where $p = 95$, we selected values of N_{obs} and N_{body} based on the criticality of the code segment measurement to our operation parameter estimations, such that F varied from a few percent to 10 percent.

Two major problems impeded our benchmarking process: buggy or inconsistent timers, and low-frequency workload variations caused by other system users. We now discuss each in detail.

Buggy or inconsistent timers were a consistent impediment to our work. To determine the resolution of a given set of timers, we measured the maximum reported elapsed time between neighboring timer calls. The high resolution timers on the CM-5 usually incremented by approximately 60 microseconds per tick, but would occasionally advance by up to 0.10 seconds, even when measured upon a dedicated machine. Similarly, the Cray high resolution clock incremented by as much as 5 milliseconds, as reported by the `timef` routine. The inconsistencies were not frequent, so we subjectively estimated the clock resolution as the largest frequently-encountered clock increment, and we removed outliers which are a factor of Y above the minimum measurement. Y was architecture-dependent and based on preliminary estimates of the variability of operation run times.

On certain architectures, low-frequency workload variations complicate the measurement process. For example, upon shared memory multiprocessors with banked memory, such as members of the Cray Y-MP family, contention for the interconnection network and memory banks causes wide fluctuations in the execution time of memory-intensive vector operations. Such contention stems from two sources (in a non-dedicated machine): *self-induced* contention, in which the processors involved in the vector computation reference memory sub-optimally, and *user-induced contention*, in which the reference patterns of other jobs composing the system workload cause our job to stall. On the Cray Y-MP/8, an 8-processor system

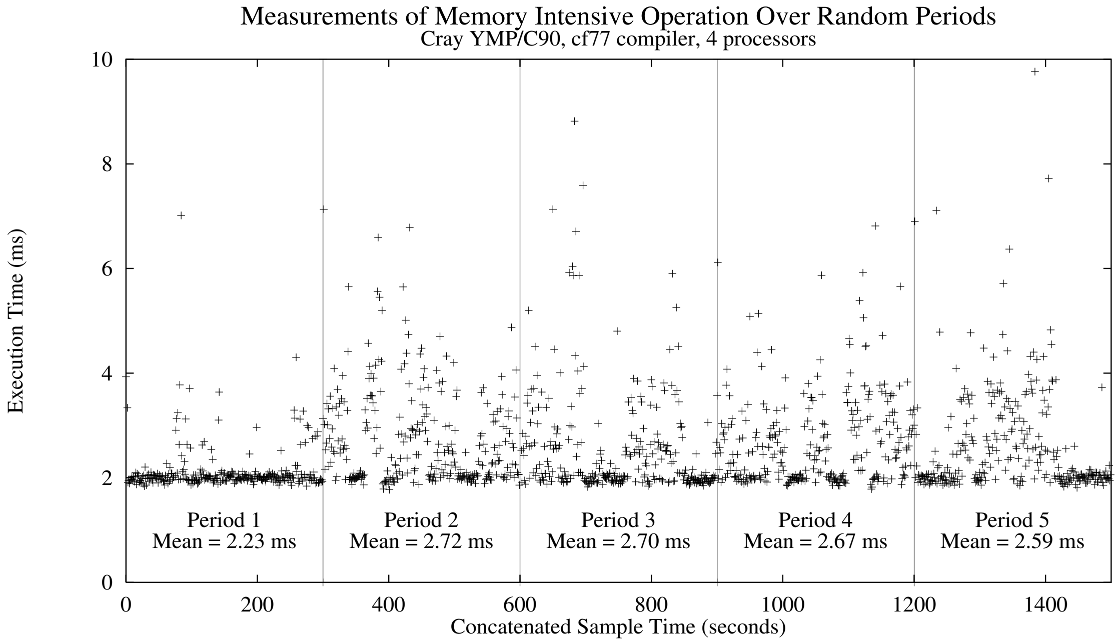


Figure 10: Sampled Run Time of Two Million-Element Array Addition and Storage, Over Various Periods

with 256 memory banks, both types of contention affect the run time of memory intensive operations significantly[15]. As a result, to accurately measure the run time of code segments which are contention-sensitive, we must be careful to collect many independent samples randomly distributed across the space of possible workload-induced contention patterns.

Figure 10 illustrates the run time of a two million-element array addition and store, running upon the Y-MP C90/`cf77` pair, sampled during several randomly spaced five-minute periods at 1-second intervals. The sample profiles are composed of runs of relatively undisturbed computation, interspersed among regions with highly irregular sampled run time; as evidenced by the sample means calculated for each period, workload-induced effects are significant and vary with very low frequency. As a result, the accurate characterization of the average run time of contention-sensitive operations upon a non-dedicated Y-MP C90 may not be possible using our benchmarking strategy, without long-term measurements which might potentially violate our tractability constraints. Thus, for all parameters measured, in addition to the average value, we record the minimum value, for use in predicting the minimum run time of a program, in the case that the average values are not adequately determinable.

Appendix B. Design Details of the Architecture Characterizer

This Appendix details the specific mechanics of several of the tests which compose the Architecture Characterization Module.

B.1 Memory Hierarchy Access Characteristics - `access`

We use a simple method to measure gather and scatter overhead. First, we measure the run time of an identity gather; that is, a gather using an identity array (element i of the array has value i) as the indexing array. Given that an identity gather and store or an n-element array has an average run time of T_{GI}, and the time to copy an n-element array is T_C, the total overhead imposed by the gather is $T_{GR} - T_C$, since the gather is performing the same work as the transfer. We use a similar method to measure scatter overhead.

To determine the average cost of a single remote read, we measure the average run time of a random gather; that is, a gather using an array which is some random permutation of an identity array. We assume P processors, an average run time of a random gather and store, T_{GR}, and an array size of n (where n is large). On the average, $n(P-1)/P^2$ references per processor in a random gather will be remote, so

$$T_{GR} = T_{GI} + (n(P-1)/P^2)(T_{RD} - T_L)$$

where T_{RD} is the cost of a remote read, and T_L is the cost of a local read. If we assume that $T_L \ll T_{RD}$, manipulating the above equation, we have

$$T_{RD} = \frac{P^2(T_{GR} - T_{GI})}{n(P-1)}$$

Note that the costs we calculate are approximations. Since the remote accesses are randomly distributed across the processors, some processors will have to perform more remote accesses than the expected number; if we assume an owner computes rule, this means some processors will go idle while others are still gathering. This means that our measured T_R is slightly inflated, but given a large n with respect to P, the difference is negligible; for confirmation of this conjecture, see Appendix C.

B.2 Distributed Memory Data Layout - `layout`

The `layout` test uses a simple method to deduce the default layout of arrays and the number of processors in a distributed memory system, assuming an owner-computes rule. We define a as an n-element array, where n is large and a power of two, i as the n-element identity array, and p as an n-element indexing array.

The layout of a a will be one of three types:[17] blocked, cyclic, or blocked-cyclic. Given P, the number of processors, we can describe any one of the above layouts with one parameter: B, the block size, in elements. Element i is stored on on processor i/B mod P. A blocked layout corresponds to $B = n/P$, a cyclic layout to $B = 1$, and a blocked-cyclic layout may be described by any of the intermediate values.

We define $T(r)$ as the run time of the following computation (we assume that array 0 is the first element in each array):

$$p = \text{mod}(i + \lfloor rn \rfloor, n)$$
$$a = a * a(p)$$

p is the identity array i, rolled upwards $\lfloor rn \rfloor$ elements. Assuming an owner-computes rule, the reference patterns of the computation $a = a * a(p)$ will change as r increases; to compute a_i, a processor will reference

[17] In the absence of any bizarre compiler bugs or unusual optimizations.

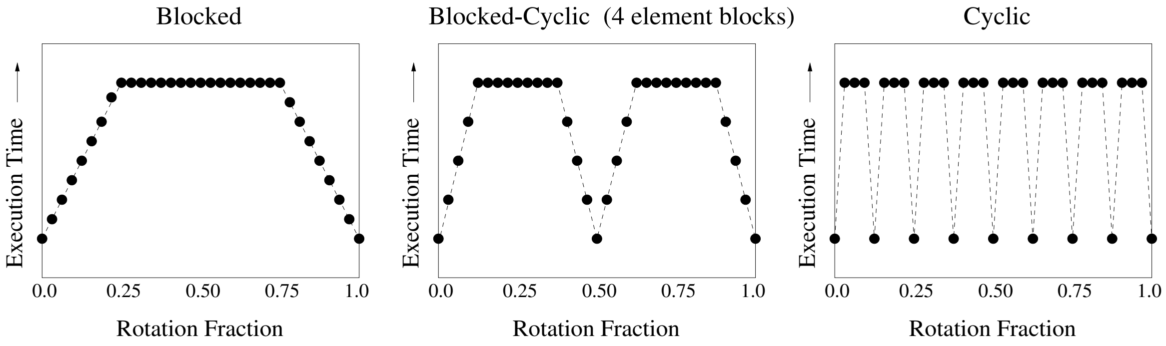


Figure 11: Ideal rotation fraction vs. execution time profiles for a 32-element array distributed across a 4-node system.

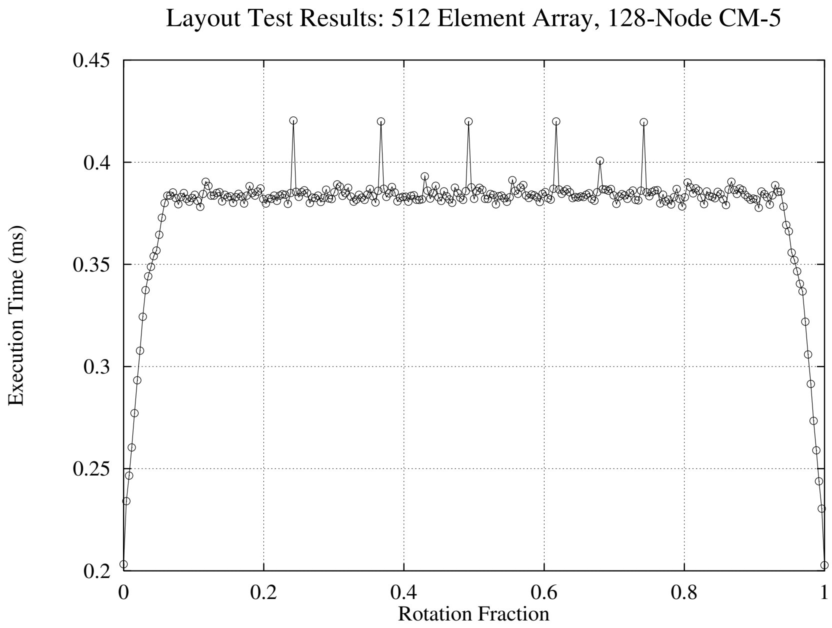


Figure 12: Rotation fraction vs. execution time profile for a 512-element array distributed across an 128-node CM-5.

its local storage for the current value of a_i, and will reference local or remote storage for the current value of $a_{p(i)}$. Essentially, the value of $T(r)$ represents the amount of overlap (in terms of elements allocated to the same processor) of the original data layout and the same data layout rotated upward $\lfloor rn \rfloor$ elements. We can imagine the layouts as two identical grates, one on top of the other, with holes in the grates corresponding to the elements in a residing upon a certain processor. The amount of light shining through the grates corresponds to the amount of overlap between the original and rotated layouts. Initially, the grates are aligned at their ends, and the most light possible shines through the grate holes. As we move the top grate to the right (corresponding to the rotation of the original layout upward), the holes in the grates no longer exactly match up; slowly, the amount of light shining through the grates shrinks, until finally no light can pass through. If the holes in the grate are large, there will be a region of grate positions (corresponding to rotation values) where the amount of light shining through fades full to none, or vice versa.

The profile of $T(r)$ versus r can be used to determine each of these parameters, and thus, the layout. Figure 11 illustrates the ideal profiles corresponding to each layout, assuming that local and remote references each incur a consistent amount of overhead. Note that in the block layout, there is one large dip, corresponding to the overlap of the continuous run of elements allocated on each processor. Conversely, in the cyclic layout, the original and rotated layouts overlap only every P elements, with no overlap elsewhere, yielding the bi-value result, with low values each P elements.

Figure 12 shows the rotation fraction versus execution time profile for a 512-element array distributed upon an 128-node CM-5. We can deduce that the layout is blocked from the presence of a single large dip. Interestingly, the dip is wider than expected, covering a range of rotation values from -0.06 to 0.06. We believe this is due to the communication characteristics of the fat tree communications network; although the network is generally flat, communications with "neighboring" processors tend to have slightly lower latencies. We theorize that the spikes in the flat portion of the profile are due to cache effects local to each processor, since they appear at fixed intervals of 64 elements.

Sampling $T(r)$ over many values of r, where r varies linearly with some induction variable, as in Figure 12, is prohibitively expensive, so we assume that the number of processors and block size will be a power of two, and only measure values of r which correspond to power of two block sizes.

Appendix C. Proofs

The following paragraphs support various conjectures about the distribution of random references across processors in a random gather.

Given P processors involved in a random gather, we expect $100(P-1)/P$-percent of the references on each processor to be remote. In the worst case, all references on a processor are remote, inflating our estimation by of the remote reference latency by a factor of $P/(P-1)$, in the case that the latency of a remote access is much more than that of a local access. Thus, on systems with a small number of processors, such distortion of our measurements may be significant; the following analysis illustrates that the probability of large distortions is negligible.

We first analyze the distribution of remote accesses in an n-element random gather upon a 2-processor system. We define

$$\begin{pmatrix} n \\ r \end{pmatrix} = \frac{n!}{r!(n-r)!}$$

$$h(x; s, n_1, n_2) = \frac{\begin{pmatrix} n_1 \\ x \end{pmatrix}\begin{pmatrix} n_2 \\ s-x \end{pmatrix}}{\begin{pmatrix} n_1+n_2 \\ s \end{pmatrix}}$$

as the hypergeometric probability density function, which computes the probability that exactly x elements of type 1 in a sample of s elements are selected, without replacement, from a set composed of n_1 elements of type 1 and n_2 elements of type 2. We define

$$H(x; s, n_1, n_2) = \sum_{i=x}^{n} h(i; s, n_1, n_2)$$

The probability that a processor will have a fraction of f remote accesses, where $f \geq 0.5$, is simply

$$y(f, n) = 2H(\lceil fn/2 \rceil; n/2, n/2, n/2)$$

Table 8 illustrates $y(0.55, n)$, the probability of a processor having 110-percent of the expected number of remote accesses, for various values of n. Note that as n increases, $y(0.55, n)$ rapidly decreases; for large values of n, the probability of a significant deviation from the expected number of remote accesses on each processor is negligible.

n	$y(0.55, n)$
50	0.656
100	0.317
150	0.191
200	0.203
250	0.129

Table 8: $y(0.55, n)$ as a function of n.

n	$z(0.825, n, 4)$
100	0.618
200	0.319
300	0.274
400	0.142
500	0.073

Table 9: $z(0.825, n, 4)$ as a function of n.

We continue our analysis for systems with a slightly larger number of processors, modelling the selection of the access type of each element in the random gather as a Bernoulli trial. Given P processors and an n-element random gather, a processor will execute n/P trials, and the probability that each trial will result in a remote access is $(P-1)/P$. We define

$$b(x; n, p) = \binom{n}{x} p^x (1-p)^{n-x}$$

as the binomial probability function, which gives the probability that exactly x Bernoulli trials will succeed out of a set of n, with p probability of success for each trial. We define

$$B(x; n, p) = \sum_{i=x}^{n} b(i; n, p)$$

The probability that one or more processors will have a fraction of f remote accesses is simply

$$z(f, n, P) = 1 - (1 - B(\lceil fn/P \rceil; n/P, (P-1)/P))^P$$

Table 9 illustrates $z(0.825, n, 4)$, the probability of any processor having 110-percent of the expected number of remote accesses, for various values of n. Note that as n increases, $z(0.825, n, 4)$ decreases; for large values of n, the probability of a significant deviation from the expected number of remote accesses on each processor is negligible.

Thus, for any number of processors, for large n, there is negligible probability of a large deviation from the expected number of remote accesses.

Appendix D. Parameter Names and Descriptions

This Appendix contains the names and meanings of parameters calculated in the Operation Measurement and Architecture Characterization modules.

D.1 Operation Measurement Module Parameters

The following tables list each of the name of each parameter, which corresponds to the run time of a given Fortran 90 array operation.

Parameter names are composed of one or more instances of some subset of the following components. A *type specifier* establishes the type of operand(s) manipulated by a given operation, or the type of the result; Table 10 lists the possible type specifiers. An *operation specifier* denotes the specific function of an operation, which is usually independent of type (for instance, an array summation, or sine function). Finally, a *size specifier* specifies the array size corresponding to a given parameter.

Specifier	Operand Type
CS	complex single-precision
L	logical
I	integer single-precision
RS	real single-precision
RD	real double-precision

Table 10: Type Specifiers

Individual parameters are systematically named and may be of one of two forms:

- T-O_n
  where O is an operation specifier, T is a type specifier, and n is a size specifier.
  Specifies a non-conversion array operation O of type T, for array size n.
- A-B_n
  where A and B are type specifiers, and n is a size specifier.
  Specifies a conversion array operation, from type A to type B, for array size n.

Note that when the array size omitted, we are referring to the characteristics of the parameter over some range of array sizes.

Table 11 lists the possible non-conversion operation parameters, specifying each possible operation specifier, legal type specifiers, and a description of the operation.

Operation	Legal Types	Description
ABS	CS I RD RS	absolute value
ADD	CS I RD RS	addition
ADD2D	CS I RD RS	addition (2-d array)
ALL	L	all elements true?
AND	L I	AND operation
ANY	L	any elements true?
BITS	I	extract fixed set of bits
BITSV	I	extract varying set of bits
BSET	I	set fixed bit
BSETV	I	set varying bit
BTEST	I	test fixed bit
BTESTV	I	test varying bit
CONJG	CS	complex conjugate
COS	CS RD RS	cosine
COUNT	L	number of elements true
CSHIFT	CS I RD RS	circular shift
DIV	CS I RD RS	division
DOTPRODUCT	CS I RD RS	dot product
EOSHIFT	CS I RD RS	end-off shift
EQ	CS I RD RS	equivalence
EQCONST	CS I RD RS	equivalence to constant
EOR	I	exclusive-OR operation
EXP	CS RD RS	natural exponential
EXPI	CS I RD RS	exponentiation by integer
EXPS	CS I RD RS	exponentiation by real
GT	I RD RS	greater than
GTCONST	I RD RS	greater than constant
LOAD	I L CS RD RS	sequential load
LOG	CS RD RS	natural logarithm
MATMUL	CS I RD RS	matrix multiplication
MAX	I RD RS	maximum

Operation	Legal Types	Description
MAXLOC	I RD RS	location of max element
MAXVAL	I RD RS	value of max element
MERGEF	RS RD	vector merge (false mask)
MERGEH	RS RD	vector merge (half-true mask)
MERGET	RS RD	vector merge (true mask)
MOD	I RD RS	modulo
MUL	CS I RD RS	multiplication
MUL2D	CS I RD RS	multiplication (2-d arrays)
NOT	L	logical NOT operation
OHS	I L CS RD RS	parallel/store overhead
OR	L I	OR operation
PACKF	RS RD	pack vector (false mask)
PACKH	RS RD	pack vector (half-true mask)
PACKT	RS RD	pack vector (true mask)
PRODUCT	CS I RD RS	vector product
REAL	CS	conversion to real
SIN	CS RD RS	sine
SHFT	I	end-off bit shift
SHFTC	I	circular bit shift
SQRT	CS RD RS	square root
STORE2D	CS I RD RS	store of constant (2-d array)
STORE	CS L I RD RS	store of constant
SUM	CS I RD RS	vector sum
TAN	RD RS	tangent
TRANSFER2D	CS I RD RS	array transfer (2-d arrays)
TRANSFER	CS L I RD RS	array transfer
TRANSPOSE	CS I RD RS	matrix transpose
UNPACKF	RS RD	unpack vector (false mask)
UNPACKH	RS RD	unpack vector (half-true mask)
UNPACKT	RS RD	unpack vector (true mask)

Table 11: Individual Parameter Names and Descriptions

Table 12 describes the weighted averages used to calculate each of the 22 reduced parameters. We selected the weight for each component parameter so that it roughly corresponds to the average number of times we would expect the corresponding operation to be executed in the *average* program, relative to the other operations used to calculate the reduced parameter. Of course, since the *average* program is workload dependent, we select commonly-executed scientific codes as our workload. Our weights, which were gleaned from Saavedra-Barrera's work [8] and a highly-regarded computer architecture textbook [17], should roughly correspond to the operation frequencies in the average scientific code.

D.2 Architecture Characterization Module Parameters

The following list describes the parameters measured by the Architecture Characterization Module. Note that all overhead is per element, over the entire machine.

REM-READ, REM-WRITE: run time of the average remote read (write).

SREM-READ, SREM-WRITE: run time of the average sequential remote read (write).

OH-GATHER, OH-SCATTER: overhead (relative to the corresponding array access), per element, of a gather (scatter).

OH-RAND-READ, OH-RAND-WRITE: overhead (relative to a linear read (write)), per element, of a set of random reads (writes) to shared memory.

OH-CACHESTRIDE$_{s,n}$: overhead, per element in the cache stride model (relative to the corresponding stride 1 reference), due to a stride s reference to an n-element array.

R1	
Parameter	Weight
RS-ADD	1.000

R2	
Parameter	Weight
RS-MUL	1.000

R3	
Parameter	Weight
RS-DIV	0.800
RS-EXPI	0.180
RS-EXPS	0.020

R4	
Parameter	Weight
RS-LOG	0.125
RS-EXP	0.125
RS-SIN	0.125
RS-TAN	0.125
RS-ABS	0.125
RS-SQRT	0.125
RS-MOD	0.125
RS-MAX	0.125

R5	
Parameter	Weight
RS-SUM	0.400
RS-PRODUCT	0.200
RS-MAXVAL	0.200
RS-DOTPRODUCT	0.200

R6	
Parameter	Weight
RD-ADD	0.650
RD-MUL	0.250
RD-DIV	0.080
RD-EXPI	0.016
RD-EXPS	0.004

R7	
Parameter	Weight
RD-LOG	0.125
RD-EXP	0.125
RD-SIN	0.125
RD-TAN	0.125
RD-ABS	0.125
RD-SQRT	0.125
RD-MOD	0.125
RD-MAX	0.125

R8	
Parameter	Weight
RD-SUM	0.400
RD-PRODUCT	0.200
RD-MAXVAL	0.200
RD-DOTPRODUCT	0.200

R9	
Parameter	Weight
CS-ADD	0.650
CS-MUL	0.250
CS-DIV	0.080
CS-EXPI	0.016
CS-EXPS	0.004

R10	
Parameter	Weight
CS-LOG	0.200
CS-EXP	0.200
CS-SIN	0.200
CS-ABS	0.200
CS-SQRT	0.200

R11	
Parameter	Weight
CS-SUM	0.250
CS-PRODUCT	0.250
CS-DOTPRODUCT	0.500

R12	
Parameter	Weight
I-ADD	0.625
I-MUL	0.250
I-DIV	0.125

R13	
Parameter	Weight
I-MOD	0.334
I-MAX	0.333
I-ABS	0.333

R14	
Parameter	Weight
I-SUM	0.400
I-PRODUCT	0.200
I-MAXVAL	0.200
I-DOTPRODUCT	0.200

R15	
Parameter	Weight
RS-MATMUL	0.500
RD-MATMUL	0.250
CS-MATMUL	0.250

R16	
Parameter	Weight
RS-ADD2D	0.400
RS-TRANSPOSE	0.100
RD-ADD2D	0.200
RD-TRANSPOSE	0.050
CS-ADD2D	0.200
CS-TRANSPOSE	0.050

R17	
Parameter	Weight
RS-CS	0.100
RS-RD	0.100
I-RS	0.150
I-RD	0.100
I-CS	0.100
RS-I	0.150
RD-I	0.100
CS-REAL	0.100
CS-CONJG	0.100

R18	
Parameter	Weight
RS-EQ	0.0928
RS-GT	0.0927
RS-EQCONST	0.0928
RS-GTCONST	0.0927
RD-EQ	0.0625
RD-GT	0.0625
RD-EQCONST	0.0625
RD-GTCONST	0.0625
I-EQ	0.0625
I-GT	0.0625
I-EQCONST	0.0625
I-GTCONST	0.0625
CS-EQ	0.0625
CS-EQCONST	0.0625

R19	
Parameter	Weight
L-AND	0.334
L-OR	0.333
L-NOT	0.333

R20	
Parameter	Weight
L-ALL	0.334
L-ANY	0.333
L-COUNT	0.333

R21	
Parameter	Weight
RS-TRANSFER	1.000

R22	
Parameter	Weight
RD-TRANSFER	0.500
CS-TRANSFER	0.500

Table 12: Reduced Parameter Compositions

OH-BANKSTRIDE$_s$: overhead per element in the bank stride model (relative to the corresponding stride 1 reference), due to a stride s reference.
OH-PARTIALMASK$_r$, OH-FULLMASK$_r$: overhead, per element, due to the specification of masked computation (relative to the corresponding unmasked computation), where a fraction r of the elements in the mask are true, using the partial (full) mask model.
OH-PARTIALBLOCK$_n$: overhead, per element, due to the specification of n-element block load/store (relative to an unblocked load/store), where n elements are loaded/stored, using the partial block model.
OH-FULLBLOCK$_p$: overhead, per element, due to the specificatin of a block block load/store (relative to an unblocked load/store), where p-percent of the elements are loaded from the source array, per the full block model.

Appendix E. Detailed Measurements

The following sections detail the specific values of various measurements.

E.1 Array Operation Characterizer Measurements

The following sections contain the specific values of array operation parameters measured for use in our performance predictions. We measure values for array sizes of 64, 256, 1024, 4096, 16384, 65536, 262144, and 1048576. We omit double-precision real measurements for the Y-MP C90/`cf77` and Y-MP C90/`f90` pairs, since they were not used in our predictions.

Note that the values of some parameters may be computed as negative; such parameters are negligible when compared to other related parameters.

64-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	8.6757e-05	(8.5874e-05,8.7640e-05)	8.5696e-05
RS-LOAD	2.4821e-06	(1.8585e-06,3.1057e-06)	2.5957e-06
RS-ADD	-2.4490e-06	(-3.3247e-06,-1.5732e-06)	-2.3043e-06
RS-MUL	-2.2201e-06	(-3.2076e-06,-1.2325e-06)	-2.5461e-06
RS-SQRT	2.7734e-07	(-9.1508e-07,1.4698e-06)	5.9158e-07
RS-LOG	2.4817e-05	(2.3463e-05,2.6171e-05)	2.4116e-05
RS-EXPI	-2.2642e-07	(-1.4978e-06,1.0450e-06)	-4.2648e-07
RS-EXPS	4.8687e-05	(4.7164e-05,5.0209e-05)	4.7852e-05
RS-SIN	1.2940e-05	(1.1679e-05,1.4201e-05)	1.2812e-05
RS-GT	5.1652e-06	(3.3931e-06,6.9373e-06)	5.2890e-06
RS-I	1.4158e-05	(1.2827e-05,1.5489e-05)	1.4549e-05
RS-MAX	-2.1954e-06	(-3.8520e-06,-5.3889e-07)	-2.2904e-06
RS-GTCONST	6.6533e-06	(5.3372e-06,7.9694e-06)	7.1793e-06
RD-OHS	9.1657e-05	(9.0512e-05,9.2802e-05)	8.9681e-05
RD-LOAD	4.5231e-07	(-1.2805e-07,1.0327e-06)	3.1394e-07
RD-ADD	-2.3910e-07	(-1.1775e-06,6.9930e-07)	3.1540e-07
RD-MUL	-6.0767e-07	(-1.6705e-06,4.5517e-07)	6.0555e-07
RD-SQRT	1.1879e-06	(-2.1524e-07,2.5910e-06)	2.1921e-06
RD-LOG	1.8703e-05	(1.7284e-05,2.0122e-05)	1.9965e-05
RD-EXPI	6.1405e-06	(4.7504e-06,7.5305e-06)	7.6538e-06
RD-EXPS	5.3672e-05	(5.2127e-05,5.5218e-05)	5.4525e-05
RD-SIN	2.1289e-05	(1.9815e-05,2.2764e-05)	2.2318e-05
RD-GT	1.4326e-05	(1.2666e-05,1.5986e-05)	1.5037e-05
RD-I	3.4055e-06	(2.1274e-06,4.6837e-06)	4.1924e-06
RD-MAX	-1.2717e-06	(-2.9793e-06,4.3588e-07)	-1.8270e-07
RD-GTCONST	9.1655e-06	(7.8817e-06,1.0449e-05)	9.9113e-06
L-OHS	8.7526e-05	(8.6515e-05,8.8537e-05)	8.6198e-05
L-LOAD	2.4821e-06	(1.8585e-06,3.1057e-06)	2.5957e-06
L-OR	4.4673e-06	(2.7709e-06,6.1637e-06)	5.0327e-06
L-NOT	2.5419e-06	(8.4403e-07,4.2397e-06)	2.7798e-06
I-OHS	8.7526e-05	(8.6515e-05,8.8537e-05)	8.6198e-05
I-LOAD	2.4821e-06	(1.8585e-06,3.1057e-06)	2.5957e-06
I-GT	6.2826e-06	(4.5416e-06,8.0235e-06)	6.2791e-06
I-AND	2.4395e-08	(-1.7344e-06,1.7832e-06)	-5.2608e-07
I-SHFT	1.5291e-06	(2.2029e-07,2.8379e-06)	1.8063e-06
I-ADD	-1.7560e-06	(-2.7520e-06,-7.5992e-07)	-1.6705e-06
I-RS	1.1842e-06	(-6.5386e-08,2.4338e-06)	1.0814e-06

256-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	8.7692e-05	(8.6479e-05,8.8906e-05)	8.5258e-05
RS-LOAD	2.6823e-06	(2.0804e-06,3.2841e-06)	2.7846e-06
RS-ADD	-2.3475e-06	(-3.4355e-06,-1.2594e-06)	-1.9880e-06
RS-MUL	-2.8087e-06	(-3.8978e-06,-1.7195e-06)	-1.8513e-06
RS-SQRT	5.7254e-08	(-1.4042e-06,1.5187e-06)	1.7909e-06
RS-LOG	2.4651e-05	(2.3156e-05,2.6145e-05)	2.6201e-05
RS-EXPI	-9.0152e-07	(-2.3421e-06,5.3904e-07)	7.6935e-07
RS-EXPS	4.8032e-05	(4.6399e-05,4.9664e-05)	4.9333e-05
RS-SIN	1.3543e-05	(1.2014e-05,1.5071e-05)	1.4680e-05
RS-GT	5.5381e-06	(3.7502e-06,7.3259e-06)	6.4240e-06
RS-I	1.2326e-05	(1.0913e-05,1.3740e-05)	1.3785e-05
RS-MAX	-3.1011e-06	(-4.8867e-06,-1.3155e-06)	-1.7136e-06
RS-GTCONST	8.2488e-06	(6.8186e-06,9.6790e-06)	9.3137e-06
RD-OHS	9.2408e-05	(9.1198e-05,9.3619e-05)	9.0144e-05
RD-LOAD	7.8015e-07	(1.7089e-07,1.3894e-06)	7.1736e-07
RD-ADD	-6.0428e-07	(-1.6351e-06,4.2655e-07)	-2.1673e-08
RD-MUL	-9.3112e-07	(-2.0270e-06,1.6476e-07)	1.7162e-07
RD-SQRT	1.4785e-06	(-2.7703e-09,2.9597e-06)	2.9927e-06
RD-LOG	2.0056e-05	(1.8498e-05,2.1615e-05)	2.1494e-05
RD-EXPI	5.0714e-06	(3.6191e-06,6.5238e-06)	6.8901e-06
RD-EXPS	5.3921e-05	(5.2209e-05,5.5633e-05)	5.4755e-05
RD-SIN	2.2680e-05	(2.1072e-05,2.4288e-05)	2.3627e-05
RD-GT	1.4197e-05	(1.2349e-05,1.6044e-05)	1.5451e-05
RD-I	1.9755e-06	(5.8269e-07,3.3683e-06)	3.4142e-06
RD-MAX	-1.2992e-08	(-1.8183e-06,1.7923e-06)	1.3286e-06
RD-GTCONST	1.0527e-05	(9.1091e-06,1.1944e-05)	1.2073e-05
L-OHS	8.7021e-05	(8.5870e-05,8.8171e-05)	8.4932e-05
L-LOAD	2.6823e-06	(2.0804e-06,3.2841e-06)	2.7846e-06
L-OR	5.9969e-06	(4.2482e-06,7.7456e-06)	7.3217e-06
L-NOT	4.1247e-06	(2.3802e-06,5.8692e-06)	5.4153e-06
I-OHS	8.7021e-05	(8.5870e-05,8.8171e-05)	8.4932e-05
I-LOAD	2.6823e-06	(2.0804e-06,3.2841e-06)	2.7846e-06
I-GT	7.6207e-06	(5.8201e-06,9.4214e-06)	7.9739e-06
I-AND	-2.5948e-06	(-4.3452e-06,-8.4442e-07)	-1.4722e-06
I-SHFT	-1.2435e-06	(-2.6321e-06,1.4503e-07)	9.2775e-08
I-ADD	-2.4289e-06	(-3.3969e-06,-1.4609e-06)	-2.2119e-06
I-RS	-1.3790e-06	(-2.8717e-06,1.1370e-07)	7.1264e-08

1024-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	8.5778e-05	(8.4647e-05,8.6909e-05)	8.4217e-05
RS-LOAD	2.5029e-06	(1.8995e-06,3.1063e-06)	2.4898e-06
RS-ADD	-2.7667e-06	(-3.8046e-06,-1.7287e-06)	-2.2661e-06
RS-MUL	-2.5229e-06	(-3.5189e-06,-1.5268e-06)	-2.1934e-06
RS-SQRT	1.0480e-06	(-3.8006e-07,2.4760e-06)	1.7361e-06
RS-LOG	2.5015e-05	(2.3523e-05,2.6508e-05)	2.5395e-05
RS-EXPI	-8.2556e-07	(-2.2151e-06,5.6393e-07)	2.8825e-08
RS-EXPS	4.8508e-05	(4.7015e-05,5.0001e-05)	4.8907e-05
RS-SIN	1.3242e-05	(1.1774e-05,1.4709e-05)	1.3835e-05
RS-GT	4.6139e-06	(2.9103e-06,6.3176e-06)	5.7366e-06
RS-I	1.1066e-05	(9.7076e-06,1.2424e-05)	1.1994e-05
RS-MAX	-2.2039e-06	(-3.9315e-06,-4.7621e-07)	-1.3000e-06
RS-GTCONST	6.4807e-06	(5.1373e-06,7.8241e-06)	7.6329e-06
RD-OHS	8.8601e-05	(8.7439e-05,8.9763e-05)	8.7390e-05
RD-LOAD	7.0400e-07	(7.4455e-08,1.3335e-06)	4.0150e-07
RD-ADD	-3.5959e-07	(-1.3927e-06,6.7355e-07)	2.3607e-07
RD-MUL	-3.0590e-07	(-1.4025e-06,7.9069e-07)	4.7364e-07
RD-SQRT	2.9004e-06	(1.4033e-06,4.3975e-06)	3.4187e-06
RD-LOG	2.0464e-05	(1.8995e-05,2.1932e-05)	2.1194e-05
RD-EXPI	6.1677e-06	(4.7404e-06,7.5950e-06)	7.0056e-06
RD-EXPS	5.4548e-05	(5.3007e-05,5.6089e-05)	5.5133e-05
RD-SIN	2.2810e-05	(2.1353e-05,2.4268e-05)	2.3620e-05
RD-GT	1.3501e-05	(1.1744e-05,1.5257e-05)	1.5176e-05
RD-I	6.8147e-07	(-7.0905e-07,2.0720e-06)	1.7547e-06
RD-MAX	-3.8972e-07	(-2.1819e-06,1.4024e-06)	7.0919e-07
RD-GTCONST	9.0221e-06	(7.6394e-06,1.0405e-05)	1.0283e-05
L-OHS	8.7752e-05	(8.6676e-05,8.8829e-05)	8.5980e-05
L-LOAD	2.5029e-06	(1.8995e-06,3.1063e-06)	2.4898e-06
L-OR	4.9092e-06	(3.2082e-06,6.6102e-06)	6.1579e-06
L-NOT	2.8685e-06	(1.1573e-06,4.5798e-06)	3.8809e-06
I-OHS	8.7752e-05	(8.6676e-05,8.8829e-05)	8.5980e-05
I-LOAD	2.5029e-06	(1.8995e-06,3.1063e-06)	2.4898e-06
I-GT	6.0525e-06	(4.3388e-06,7.7662e-06)	6.9927e-06
I-AND	-2.9104e-06	(-4.6166e-06,-1.2041e-06)	-1.8150e-06
I-SHFT	-1.6848e-06	(-3.0085e-06,-3.6105e-07)	-5.2308e-07
I-ADD	-2.2248e-06	(-3.2558e-06,-1.1937e-06)	-1.7689e-06
I-RS	-1.8731e-07	(-1.6018e-06,1.2272e-06)	6.1861e-07

4096-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	8.6948e-05	(8.5718e-05,8.8178e-05)	8.4612e-05
RS-LOAD	2.7318e-06	(2.1107e-06,3.3528e-06)	2.8017e-06
RS-ADD	-3.6298e-06	(-4.7167e-06,-2.5429e-06)	-2.5524e-06
RS-MUL	-3.4809e-06	(-4.5827e-06,-2.3790e-06)	-3.0071e-06
RS-SQRT	-1.1406e-06	(-2.6378e-06,3.5666e-07)	3.7579e-07
RS-LOG	2.2989e-05	(2.1470e-05,2.4508e-05)	2.4138e-05
RS-EXPI	-2.8443e-06	(-4.3433e-06,-1.3452e-06)	-1.2381e-06
RS-EXPS	4.6919e-05	(4.5343e-05,4.8494e-05)	4.8168e-05
RS-SIN	1.1242e-05	(9.7176e-06,1.2767e-05)	1.2562e-05
RS-GT	3.3881e-06	(1.7027e-06,5.0736e-06)	3.3234e-06
RS-I	9.3744e-06	(8.0737e-06,1.0675e-05)	1.0049e-05
RS-MAX	-4.7394e-06	(-6.5502e-06,-2.9286e-06)	-2.9631e-06
RS-GTCONST	5.4493e-06	(4.1457e-06,6.7528e-06)	5.3369e-06
RD-OHS	9.2860e-05	(9.1708e-05,9.4012e-05)	9.0749e-05
RD-LOAD	3.9387e-07	(-2.1042e-07,9.9815e-07)	3.4181e-07
RD-ADD	-9.6156e-08	(-1.0461e-06,8.5374e-07)	3.2211e-07
RD-MUL	-8.0090e-07	(-1.9147e-06,3.1288e-07)	4.7025e-07
RD-SQRT	1.1570e-06	(-3.1305e-07,2.6271e-06)	2.1392e-06
RD-LOG	1.8998e-05	(1.7562e-05,2.0434e-05)	2.0266e-05
RD-EXPI	5.0961e-06	(3.6835e-06,6.5086e-06)	6.7084e-06
RD-EXPS	5.3875e-05	(5.2334e-05,5.5417e-05)	5.5041e-05
RD-SIN	2.1793e-05	(2.0279e-05,2.3306e-05)	2.2916e-05
RD-GT	1.3064e-05	(1.1403e-05,1.4724e-05)	1.3670e-05
RD-I	-2.8095e-07	(-1.5412e-06,9.7929e-07)	4.6836e-07
RD-MAX	-8.2296e-07	(-2.6207e-06,9.7475e-07)	3.8344e-07
RD-GTCONST	8.3694e-06	(7.0805e-06,9.6583e-06)	9.0206e-06
L-OHS	9.0036e-05	(8.9047e-05,9.1025e-05)	8.8766e-05
L-LOAD	2.7318e-06	(2.1107e-06,3.3528e-06)	2.8017e-06
L-OR	3.2948e-06	(1.6121e-06,4.9776e-06)	3.6203e-06
L-NOT	2.5168e-06	(8.3380e-07,4.1997e-06)	3.0204e-06
I-OHS	9.0036e-05	(8.9047e-05,9.1025e-05)	8.8766e-05
I-LOAD	2.7318e-06	(2.1107e-06,3.3528e-06)	2.8017e-06
I-GT	4.3475e-06	(2.6565e-06,6.0384e-06)	4.5903e-06
I-AND	-4.3375e-06	(-6.0039e-06,-2.6712e-06)	-3.8807e-06
I-SHFT	-2.3250e-06	(-3.5939e-06,-1.0561e-06)	-1.6549e-06
I-ADD	-2.6372e-06	(-3.6953e-06,-1.5791e-06)	-2.4053e-06
I-RS	-4.4465e-07	(-1.9111e-06,1.0218e-06)	1.1557e-06

16384-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	9.3528e-05	(9.2379e-05,9.4676e-05)	9.2174e-05
RS-LOAD	3.4715e-06	(2.8440e-06,4.0991e-06)	3.3104e-06
RS-ADD	-2.1778e-06	(-3.2802e-06,-1.0755e-06)	-1.6621e-06
RS-MUL	-2.3915e-06	(-3.3742e-06,-1.4088e-06)	-1.7208e-06
RS-SQRT	8.7352e-06	(7.2759e-06,1.0194e-05)	9.5632e-06
RS-LOG	8.3767e-05	(8.2072e-05,8.5463e-05)	8.4261e-05
RS-EXPI	3.7475e-06	(2.3277e-06,5.1673e-06)	4.5748e-06
RS-EXPS	1.7355e-04	(1.7135e-04,1.7574e-04)	1.7305e-04
RS-SIN	5.7816e-05	(5.6244e-05,5.9387e-05)	5.8260e-05
RS-GT	7.2992e-06	(5.5099e-06,9.0886e-06)	8.5027e-06
RS-I	1.2137e-05	(1.0718e-05,1.3557e-05)	1.3071e-05
RS-MAX	-1.6639e-06	(-3.4486e-06,1.2088e-07)	-6.9138e-07
RS-GTCONST	1.0217e-05	(8.6576e-06,1.1777e-05)	1.0673e-05
RD-OHS	1.0056e-04	(9.9379e-05,1.0175e-04)	9.8406e-05
RD-LOAD	1.3829e-06	(7.0439e-07,2.0613e-06)	1.5377e-06
RD-ADD	-2.3023e-07	(-1.3778e-06,9.1734e-07)	4.9683e-07
RD-MUL	-1.9580e-07	(-1.1829e-06,7.9133e-07)	1.5259e-07
RD-SQRT	1.2934e-05	(1.1440e-05,1.4428e-05)	1.4253e-05
RD-LOG	8.4699e-05	(8.2992e-05,8.6407e-05)	8.5199e-05
RD-EXPI	1.0154e-05	(8.6506e-06,1.1657e-05)	1.1223e-05
RD-EXPS	2.0074e-04	(1.9851e-04,2.0296e-04)	2.0114e-04
RD-SIN	9.4511e-05	(9.2634e-05,9.6387e-05)	9.5014e-05
RD-GT	1.6548e-05	(1.4628e-05,1.8468e-05)	1.6796e-05
RD-I	1.3451e-06	(-1.3687e-07,2.8270e-06)	1.5141e-06
RD-MAX	9.1424e-07	(-9.8495e-07,2.8134e-06)	2.1078e-06
RD-GTCONST	1.3055e-05	(1.1588e-05,1.4522e-05)	1.3474e-05
L-OHS	9.3162e-05	(9.2030e-05,9.4295e-05)	9.1732e-05
L-LOAD	3.4715e-06	(2.8440e-06,4.0991e-06)	3.3104e-06
L-OR	5.7279e-06	(3.9446e-06,7.5113e-06)	6.9128e-06
L-NOT	2.8709e-06	(1.0833e-06,4.6584e-06)	3.8599e-06
I-OHS	9.3162e-05	(9.2030e-05,9.4295e-05)	9.1732e-05
I-LOAD	3.4715e-06	(2.8440e-06,4.0991e-06)	3.3104e-06
I-GT	9.5120e-06	(7.7228e-06,1.1301e-05)	1.0541e-05
I-AND	-2.0856e-06	(-3.8697e-06,-3.0145e-07)	-1.0934e-06
I-SHFT	-2.5237e-07	(-1.6531e-06,1.1483e-06)	5.1835e-07
I-ADD	-1.5022e-06	(-2.4583e-06,-5.4602e-07)	-1.0680e-06
I-RS	-2.4376e-06	(-3.8557e-06,-1.0194e-06)	-1.6008e-06

65536-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	1.1917e-04	(1.1744e-04,1.2089e-04)	1.1678e-04
RS-LOAD	5.4898e-06	(4.5886e-06,6.3909e-06)	5.2954e-06
RS-ADD	2.1534e-06	(6.3651e-07,3.6704e-06)	2.7581e-06
RS-MUL	1.9737e-06	(2.7802e-07,3.6693e-06)	3.1633e-06
RS-SQRT	4.6305e-05	(4.4099e-05,4.8510e-05)	4.8099e-05
RS-LOG	3.2887e-04	(3.2519e-04,3.3255e-04)	3.2890e-04
RS-EXPI	2.5583e-05	(2.3476e-05,2.7691e-05)	2.7246e-05
RS-EXPS	6.7697e-04	(6.7129e-04,6.8264e-04)	6.7462e-04
RS-SIN	2.4265e-04	(2.3982e-04,2.4549e-04)	2.4339e-04
RS-GT	1.8590e-05	(1.5850e-05,2.1329e-05)	2.0420e-05
RS-I	1.3831e-05	(1.1717e-05,1.5944e-05)	1.5980e-05
RS-MAX	9.3883e-06	(6.7817e-06,1.1995e-05)	1.1238e-05
RS-GTCONST	2.3538e-05	(2.1384e-05,2.5692e-05)	2.5378e-05
RD-OHS	1.3977e-04	(1.3754e-04,1.4200e-04)	1.3518e-04
RD-LOAD	5.4879e-06	(4.5219e-06,6.4540e-06)	5.0990e-06
RD-ADD	1.5658e-06	(-3.7560e-07,3.5072e-06)	3.6179e-06
RD-MUL	1.7380e-06	(-2.4366e-07,3.7196e-06)	3.6578e-06
RD-SQRT	6.2223e-05	(5.9541e-05,6.4906e-05)	6.5888e-05
RD-LOG	3.4656e-04	(3.4292e-04,3.5020e-04)	3.4900e-04
RD-EXPI	2.6533e-05	(2.3929e-05,2.9138e-05)	3.0759e-05
RD-EXPS	7.8777e-04	(7.8200e-04,7.9354e-04)	7.8942e-04
RD-SIN	3.8189e-04	(3.7809e-04,3.8570e-04)	3.8516e-04
RD-GT	2.3863e-05	(2.1033e-05,2.6694e-05)	2.5971e-05
RD-I	1.3298e-06	(-8.0359e-07,3.4631e-06)	3.5499e-06
RD-MAX	9.1184e-06	(5.9314e-06,1.2305e-05)	1.2956e-05
RD-GTCONST	2.4031e-05	(2.1856e-05,2.6206e-05)	2.5997e-05
L-OHS	1.2171e-04	(1.1996e-04,1.2347e-04)	1.1909e-04
L-LOAD	5.4898e-06	(4.5886e-06,6.3909e-06)	5.2954e-06
L-OR	7.1847e-06	(4.5297e-06,9.8397e-06)	9.0118e-06
L-NOT	1.1252e-06	(-1.5048e-06,3.7553e-06)	3.3121e-06
I-OHS	1.2171e-04	(1.1996e-04,1.2347e-04)	1.1909e-04
I-LOAD	5.4898e-06	(4.5886e-06,6.3909e-06)	5.2954e-06
I-GT	2.0408e-05	(1.7756e-05,2.3061e-05)	2.2028e-05
I-AND	-8.0259e-07	(-3.4168e-06,1.8116e-06)	1.4750e-06
I-SHFT	1.4348e-07	(-1.9464e-06,2.2334e-06)	2.0734e-06
I-ADD	2.5153e-06	(9.8427e-07,4.0464e-06)	3.3471e-06
I-RS	2.2022e-06	(1.3908e-07,4.2653e-06)	4.0033e-06

262144-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	2.3825e-04	(2.3454e-04,2.4197e-04)	2.3200e-04
RS-LOAD	1.4424e-05	(1.2621e-05,1.6227e-05)	1.4571e-05
RS-ADD	7.9185e-06	(4.4421e-06,1.1395e-05)	9.5869e-06
RS-MUL	1.3752e-05	(1.0645e-05,1.6859e-05)	1.6022e-05
RS-SQRT	1.7827e-04	(1.7343e-04,1.8310e-04)	1.8299e-04
RS-LOG	1.2894e-03	(1.2778e-03,1.3010e-03)	1.2880e-03
RS-EXPI	9.6252e-05	(9.1533e-05,1.0097e-04)	1.0051e-04
RS-EXPS	2.6699e-03	(2.6504e-03,2.6894e-03)	2.6647e-03
RS-SIN	9.6961e-04	(9.6130e-04,9.7791e-04)	9.7154e-04
RS-GT	7.5462e-05	(7.0092e-05,8.0831e-05)	7.8462e-05
RS-I	2.4413e-05	(2.0180e-05,2.8646e-05)	2.7487e-05
RS-MAX	3.5358e-05	(2.9706e-05,4.1010e-05)	3.9182e-05
RS-GTCONST	8.4084e-05	(7.9657e-05,8.8512e-05)	8.6945e-05
RD-OHS	2.9458e-04	(2.9014e-04,2.9901e-04)	2.8790e-04
RD-LOAD	2.7369e-05	(2.5188e-05,2.9549e-05)	2.6737e-05
RD-ADD	-2.8921e-07	(-4.3816e-06,3.8032e-06)	3.6469e-06
RD-MUL	4.2106e-06	(2.1261e-07,8.2086e-06)	6.2581e-06
RD-SQRT	2.4194e-04	(2.3609e-04,2.4779e-04)	2.4759e-04
RD-LOG	1.3831e-03	(1.3702e-03,1.3961e-03)	1.3825e-03
RD-EXPI	8.3219e-05	(7.7762e-05,8.8677e-05)	8.8315e-05
RD-EXPS	3.1285e-03	(3.1048e-03,3.1522e-03)	3.1185e-03
RD-SIN	1.5369e-03	(1.5228e-03,1.5510e-03)	1.5342e-03
RD-GT	6.1167e-05	(5.5106e-05,6.7228e-05)	6.3884e-05
RD-I	-3.0514e-07	(-4.6870e-06,4.0767e-06)	3.2002e-06
RD-MAX	2.8698e-05	(2.2160e-05,3.5237e-05)	3.5194e-05
RD-GTCONST	7.2199e-05	(6.7709e-05,7.6688e-05)	7.6122e-05
L-OHS	2.2755e-04	(2.2406e-04,2.3103e-04)	2.2284e-04
L-LOAD	1.4424e-05	(1.2621e-05,1.6227e-05)	1.4571e-05
L-OR	2.2371e-05	(1.7105e-05,2.7637e-05)	2.5545e-05
L-NOT	4.0195e-06	(-1.2203e-06,9.2593e-06)	7.2127e-06
I-OHS	2.2755e-04	(2.2406e-04,2.3103e-04)	2.2284e-04
I-LOAD	1.4424e-05	(1.2621e-05,1.6227e-05)	1.4571e-05
I-GT	7.4800e-05	(6.9423e-05,8.0178e-05)	7.6948e-05
I-AND	2.2818e-05	(1.7568e-05,2.8068e-05)	2.5687e-05
I-SHFT	1.4409e-05	(1.0189e-05,1.8628e-05)	1.7025e-05
I-ADD	1.4489e-05	(1.1251e-05,1.7726e-05)	1.6731e-05
I-RS	-1.5973e-07	(-4.5885e-06,4.2691e-06)	4.5015e-06

1048576-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	6.5853e-04	(6.4766e-04,6.6940e-04)	6.4727e-04
RS-LOAD	9.6175e-05	(9.0458e-05,1.0189e-04)	9.5450e-05
RS-ADD	2.4317e-05	(1.4058e-05,3.4577e-05)	2.6931e-05
RS-MUL	2.2516e-05	(1.2228e-05,3.2804e-05)	2.7129e-05
RS-SQRT	7.2238e-04	(7.0685e-04,7.3791e-04)	7.2617e-04
RS-LOG	5.1812e-03	(5.1485e-03,5.2140e-03)	5.1759e-03
RS-EXPI	3.9136e-04	(3.7716e-04,4.0556e-04)	3.9854e-04
RS-EXPS	1.0742e-02	(1.0679e-02,1.0804e-02)	1.0719e-02
RS-SIN	3.9052e-03	(3.8708e-03,3.9395e-03)	3.9007e-03
RS-MAX	1.4786e-04	(1.3113e-04,1.6459e-04)	1.5814e-04
RD-OHS	8.5811e-04	(8.4623e-04,8.6998e-04)	8.5215e-04
RD-LOAD	1.3975e-04	(1.3275e-04,1.4676e-04)	1.3627e-04
RD-ADD	-8.1615e-06	(-1.9691e-05,3.3677e-06)	-5.3154e-06
RD-MUL	-1.0660e-05	(-2.2703e-05,1.3817e-06)	-4.7231e-06
RD-SQRT	9.9204e-04	(9.7417e-04,1.0099e-03)	9.9359e-04
RD-LOG	5.6172e-03	(5.5651e-03,5.6693e-03)	5.5801e-03
RD-EXPI	3.4376e-04	(3.2781e-04,3.5970e-04)	3.4682e-04
RD-EXPS	1.2582e-02	(1.2489e-02,1.2676e-02)	1.2533e-02
RD-SIN	6.1532e-03	(6.1039e-03,6.2024e-03)	6.1362e-03
RD-MAX	1.1691e-04	(9.7158e-05,1.3666e-04)	1.2462e-04
L-LOAD	9.6175e-05	(9.0458e-05,1.0189e-04)	9.5450e-05
I-LOAD	9.6175e-05	(9.0458e-05,1.0189e-04)	9.5450e-05
I-RS	2.7585e-06	(-1.0783e-05,1.6300e-05)	7.2701e-06

E.1.2 Y-MP C90/`cf77` Operation Measurements

64-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	1.8957e-07	(1.3049e-07,2.4865e-07)	1.0918e-07
RS-LOAD	-4.4621e-08	(-1.2780e-07,3.8554e-08)	2.7141e-08
RS-ADD	2.0926e-07	(1.2593e-07,2.9259e-07)	1.3698e-07
RS-MUL	1.9573e-07	(1.1236e-07,2.7910e-07)	1.2253e-07
RS-SQRT	2.0584e-06	(1.9554e-06,2.1615e-06)	2.0623e-06
RS-LOG	5.7688e-06	(5.6609e-06,5.8766e-06)	5.7133e-06
RS-EXPI	1.4724e-06	(1.3698e-06,1.5751e-06)	1.4765e-06
RS-EXPS	2.1615e-05	(2.1454e-05,2.1776e-05)	2.1552e-05
RS-SIN	8.6851e-06	(8.5712e-06,8.7991e-06)	8.5917e-06
RS-GT	2.4085e-07	(6.3995e-08,4.1771e-07)	1.6467e-07
RS-I	1.5045e-07	(4.7635e-08,2.5327e-07)	1.5009e-07
RS-MAX	2.1381e-07	(3.7194e-08,3.9042e-07)	1.4591e-07
RS-GTCONST	1.6238e-07	(6.0103e-08,2.6465e-07)	1.6040e-07
L-OHS	9.2865e-08	(3.3601e-08,1.5213e-07)	2.8678e-08
L-LOAD	-4.4621e-08	(-1.2780e-07,3.8554e-08)	2.7141e-08
L-OR	3.2513e-07	(1.4844e-07,5.0182e-07)	2.4416e-07
L-NOT	2.8580e-07	(1.0912e-07,4.6248e-07)	2.0611e-07
I-OHS	9.2865e-08	(3.3601e-08,1.5213e-07)	2.8678e-08
I-LOAD	-4.4621e-08	(-1.2780e-07,3.8554e-08)	2.7141e-08
I-GT	9.0856e-08	(-8.5802e-08,2.6751e-07)	1.5404e-08
I-AND	3.5080e-07	(1.7363e-07,5.2797e-07)	2.5952e-07
I-SHFT	1.2790e-07	(2.5635e-08,2.3017e-07)	1.2688e-07
I-ADD	1.5250e-07	(6.9189e-08,2.3581e-07)	7.3385e-08
I-RS	9.2744e-08	(-9.4059e-09,1.9489e-07)	9.7033e-08

256-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	2.6542e-07	(2.5265e-07,2.7818e-07)	2.4425e-07
RS-LOAD	7.7806e-08	(6.7260e-08,8.8353e-08)	6.7476e-08
RS-ADD	4.3638e-07	(4.2188e-07,4.5089e-07)	4.4816e-07
RS-MUL	4.3765e-07	(4.2369e-07,4.5162e-07)	4.3776e-07
RS-SQRT	1.1339e-05	(1.1241e-05,1.1438e-05)	1.0435e-05
RS-LOG	1.8250e-05	(1.8139e-05,1.8362e-05)	1.8028e-05
RS-EXPI	5.5006e-06	(5.4404e-06,5.5609e-06)	5.3942e-06
RS-EXPS	8.1431e-05	(7.9710e-05,8.3151e-05)	7.4981e-05
RS-SIN	2.4514e-05	(2.4367e-05,2.4661e-05)	2.4244e-05
RS-GT	6.5245e-07	(6.2374e-07,6.8115e-07)	6.3238e-07
RS-I	6.3574e-07	(6.1429e-07,6.5718e-07)	6.0960e-07
RS-MAX	1.0019e-06	(9.7479e-07,1.0290e-06)	1.0118e-06
RS-GTCONST	6.2735e-07	(6.0565e-07,6.4905e-07)	5.9691e-07
L-OHS	2.0309e-07	(1.8657e-07,2.1961e-07)	2.1867e-07
L-LOAD	7.7806e-08	(6.7260e-08,8.8353e-08)	6.7476e-08
L-OR	7.5509e-07	(7.2629e-07,7.8389e-07)	7.2466e-07
L-NOT	6.2686e-07	(5.9857e-07,6.5515e-07)	6.0995e-07
I-OHS	2.0309e-07	(1.8657e-07,2.1961e-07)	2.1867e-07
I-LOAD	7.7806e-08	(6.7260e-08,8.8353e-08)	6.7476e-08
I-GT	4.2269e-07	(3.9471e-07,4.5067e-07)	4.0949e-07
I-AND	1.0699e-06	(9.0204e-07,1.2378e-06)	7.7523e-07
I-SHFT	1.4437e-06	(1.2813e-06,1.6061e-06)	1.1517e-06
I-ADD	4.9497e-07	(4.7040e-07,5.1954e-07)	4.6206e-07
I-RS	5.8343e-07	(5.6457e-07,6.0230e-07)	5.9059e-07

1024-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	7.0445e-07	(6.3486e-07,7.7404e-07)	7.2761e-07
RS-LOAD	3.0453e-07	(2.2567e-07,3.8338e-07)	2.3639e-07
RS-ADD	1.6262e-06	(1.5394e-06,1.7130e-06)	1.6417e-06
RS-MUL	1.7520e-06	(1.6590e-06,1.8451e-06)	1.6978e-06
RS-SQRT	1.5689e-05	(1.5191e-05,1.6187e-05)	1.5054e-05
RS-LOG	4.0483e-05	(4.0227e-05,4.0740e-05)	4.0071e-05
RS-EXPI	2.2276e-05	(2.2104e-05,2.2447e-05)	2.2288e-05
RS-EXPS	3.0427e-04	(3.0258e-04,3.0596e-04)	2.9965e-04
RS-SIN	6.0197e-05	(5.9829e-05,6.0564e-05)	5.9334e-05
RS-GT	1.8271e-06	(1.6487e-06,2.0056e-06)	1.8125e-06
RS-I	1.8206e-06	(1.7065e-06,1.9347e-06)	1.7824e-06
RS-MAX	4.8031e-06	(4.6261e-06,4.9801e-06)	4.8294e-06
RS-GTCONST	1.8443e-06	(1.7295e-06,1.9592e-06)	1.7692e-06
L-OHS	1.3828e-06	(1.3035e-06,1.4621e-06)	1.4488e-06
L-LOAD	3.0453e-07	(2.2567e-07,3.8338e-07)	2.3639e-07
L-OR	1.9999e-06	(1.8205e-06,2.1794e-06)	1.9666e-06
L-NOT	1.5625e-06	(1.3841e-06,1.7409e-06)	1.5060e-06
I-OHS	1.3828e-06	(1.3035e-06,1.4621e-06)	1.4488e-06
I-LOAD	3.0453e-07	(2.2567e-07,3.8338e-07)	2.3639e-07
I-GT	1.8863e-06	(1.6990e-06,2.0735e-06)	1.8067e-06
I-AND	2.2691e-06	(2.0604e-06,2.4779e-06)	2.1463e-06
I-SHFT	3.9184e-06	(3.8003e-06,4.0366e-06)	3.8449e-06
I-ADD	1.7896e-06	(1.6942e-06,1.8849e-06)	1.7419e-06
I-RS	2.5450e-06	(2.4370e-06,2.6529e-06)	2.5486e-06

4096-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	9.0895e-06	(8.6134e-06,9.5656e-06)	8.6533e-06
RS-LOAD	7.6422e-07	(4.8965e-07,1.0388e-06)	3.8191e-07
RS-ADD	1.9478e-06	(1.4009e-06,2.4947e-06)	2.0393e-06
RS-MUL	1.9851e-06	(1.4840e-06,2.4862e-06)	1.8821e-06
RS-SQRT	3.9983e-05	(3.7186e-05,4.2780e-05)	2.9058e-05
RS-LOG	1.7060e-04	(1.6200e-04,1.7920e-04)	1.2937e-04
RS-EXPI	2.2982e-05	(2.2406e-05,2.3558e-05)	2.3693e-05
RS-EXPS	4.6128e-04	(4.4233e-04,4.8022e-04)	3.6827e-04
RS-SIN	2.5970e-04	(2.4088e-04,2.7853e-04)	1.8711e-04
RS-GT	4.2870e-06	(3.1252e-06,5.4488e-06)	1.3613e-06
RS-I	3.8806e-06	(2.8491e-06,4.9122e-06)	1.3161e-06
RS-MAX	7.7663e-06	(6.3032e-06,9.2295e-06)	4.6338e-06
RS-GTCONST	4.2790e-06	(3.1788e-06,5.3793e-06)	1.1725e-06
L-OHS	7.8462e-06	(6.9926e-06,8.6997e-06)	8.9599e-06
L-LOAD	7.6422e-07	(4.8965e-07,1.0388e-06)	3.8191e-07
L-OR	2.3337e-06	(1.3134e-06,3.3539e-06)	1.5594e-06
L-NOT	1.5375e-06	(5.1351e-07,2.5614e-06)	7.2976e-07
I-OHS	7.8462e-06	(6.9926e-06,8.6997e-06)	8.9599e-06
I-LOAD	7.6422e-07	(4.8965e-07,1.0388e-06)	3.8191e-07
I-GT	1.9492e-06	(9.3202e-07,2.9665e-06)	1.4290e-06
I-AND	2.0557e-06	(1.0388e-06,3.0726e-06)	1.6088e-06
I-SHFT	5.6667e-06	(4.7420e-06,6.5914e-06)	4.0772e-06
I-ADD	2.8337e-06	(2.3411e-06,3.3263e-06)	1.7861e-06
I-RS	2.1544e-06	(1.4727e-06,2.8361e-06)	1.5980e-06

16384-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	1.0210e-05	(7.6431e-06,1.2778e-05)	1.1790e-05
RS-LOAD	3.7532e-06	(1.5086e-06,5.9978e-06)	2.5228e-06
RS-ADD	7.6708e-06	(4.9042e-06,1.0437e-05)	4.7340e-06
RS-MUL	9.8271e-06	(6.5282e-06,1.3126e-05)	5.6217e-06
RS-SQRT	2.3694e-04	(2.1969e-04,2.5419e-04)	1.2780e-04
RS-LOG	6.3230e-04	(5.9098e-04,6.7362e-04)	4.7918e-04
RS-EXPI	1.4163e-04	(1.3372e-04,1.4954e-04)	9.5619e-05
RS-EXPS	1.6324e-03	(1.5528e-03,1.7120e-03)	1.4351e-03
RS-SIN	9.6406e-04	(8.7807e-04,1.0501e-03)	6.9099e-04
RS-GT	5.3919e-06	(-2.7617e-06,1.3545e-05)	1.8811e-06
RS-I	-3.0978e-06	(-9.9075e-06,3.7119e-06)	3.4854e-06
RS-MAX	4.1023e-05	(3.4857e-05,4.7188e-05)	2.2303e-05
RS-GTCONST	5.2165e-06	(-1.8835e-06,1.2316e-05)	2.7457e-06
L-OHS	1.9018e-05	(1.2590e-05,2.5446e-05)	1.3582e-05
L-LOAD	3.7532e-06	(1.5086e-06,5.9978e-06)	2.5228e-06
L-OR	1.7938e-06	(-6.1961e-06,9.7837e-06)	2.1211e-06
L-NOT	-2.6882e-06	(-1.0569e-05,5.1922e-06)	9.7461e-07
I-OHS	1.9018e-05	(1.2590e-05,2.5446e-05)	1.3582e-05
I-LOAD	3.7532e-06	(1.5086e-06,5.9978e-06)	2.5228e-06
I-GT	-5.4350e-06	(-1.3277e-05,2.4069e-06)	1.9598e-06
I-AND	-4.3626e-06	(-1.2214e-05,3.4887e-06)	2.6668e-06
I-SHFT	8.1504e-06	(1.3391e-06,1.4962e-05)	1.4531e-05
I-ADD	1.8703e-05	(1.0754e-05,2.6653e-05)	7.1444e-06
I-RS	1.3580e-05	(9.3376e-06,1.7822e-05)	4.3456e-06

65536-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	2.3224e-05	(1.8243e-05,2.8206e-05)	2.4953e-05
RS-LOAD	2.4279e-06	(-1.2627e-06,6.1185e-06)	5.3697e-06
RS-ADD	4.2030e-05	(3.6180e-05,4.7880e-05)	2.5156e-05
RS-MUL	4.3148e-05	(3.6686e-05,4.9610e-05)	2.3486e-05
RS-SQRT	4.6598e-04	(4.5926e-04,4.7270e-04)	4.6042e-04
RS-LOG	1.9272e-03	(1.9116e-03,1.9428e-03)	1.8501e-03
RS-EXPI	3.5811e-04	(3.5156e-04,3.6465e-04)	3.5245e-04
RS-EXPS	5.6589e-03	(5.6144e-03,5.7034e-03)	5.3434e-03
RS-SIN	2.3837e-03	(2.3249e-03,2.4425e-03)	2.2567e-03
RS-GT	4.7137e-05	(3.2713e-05,6.1562e-05)	1.0051e-05
RS-I	2.5213e-05	(1.3284e-05,3.7143e-05)	9.7899e-06
RS-MAX	8.1426e-05	(7.2498e-05,9.0355e-05)	7.1788e-05
RS-GTCONST	4.3637e-05	(3.0562e-05,5.6712e-05)	9.3100e-06
L-OHS	3.3543e-05	(2.2620e-05,4.4466e-05)	3.4450e-05
L-LOAD	2.4279e-06	(-1.2627e-06,6.1185e-06)	5.3697e-06
L-OR	2.9483e-05	(1.5799e-05,4.3167e-05)	1.3276e-05
L-NOT	1.1381e-05	(-1.8056e-06,2.4568e-05)	3.9093e-06
I-OHS	3.3543e-05	(2.2620e-05,4.4466e-05)	3.4450e-05
I-LOAD	2.4279e-06	(-1.2627e-06,6.1185e-06)	5.3697e-06
I-GT	2.8252e-05	(1.4721e-05,4.1784e-05)	1.3562e-05
I-AND	3.6120e-05	(2.2413e-05,4.9828e-05)	1.4412e-05
I-SHFT	6.1873e-05	(5.0328e-05,7.3419e-05)	5.6706e-05
I-ADD	4.1681e-05	(3.5645e-05,4.7718e-05)	2.4970e-05
I-RS	2.3926e-05	(1.7720e-05,3.0132e-05)	1.8624e-05

262144-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	1.2748e-04	(1.0003e-04,1.5493e-04)	7.8420e-05
RS-LOAD	3.1849e-05	(1.6779e-05,4.6918e-05)	2.2872e-05
RS-ADD	1.2514e-04	(1.0143e-04,1.4885e-04)	9.5227e-05
RS-MUL	5.5928e-05	(2.5914e-05,8.5943e-05)	8.5608e-05
RS-SQRT	1.7798e-03	(1.7467e-03,1.8129e-03)	1.8352e-03
RS-LOG	7.9792e-03	(7.6538e-03,8.3046e-03)	7.4670e-03
RS-EXPI	1.4538e-03	(1.3587e-03,1.5490e-03)	1.4087e-03
RS-EXPS	2.2204e-02	(2.2023e-02,2.2385e-02)	2.1322e-02
RS-SIN	9.4417e-03	(9.0362e-03,9.8472e-03)	8.7465e-03
RS-GT	1.8316e-04	(1.3919e-04,2.2714e-04)	1.1449e-04
RS-I	1.8575e-04	(1.5038e-04,2.2112e-04)	1.1263e-04
RS-MAX	2.2105e-04	(1.8022e-04,2.6188e-04)	2.8372e-04
RS-GTCONST	2.3534e-04	(1.9695e-04,2.7373e-04)	1.1344e-04
L-OHS	-4.4123e-05	(-7.6098e-05,-1.2148e-05)	3.5179e-05
L-LOAD	3.1849e-05	(1.6779e-05,4.6918e-05)	2.2872e-05
L-OR	2.3621e-04	(1.8950e-04,2.8293e-04)	1.2660e-04
L-NOT	1.8753e-04	(1.4228e-04,2.3279e-04)	9.2014e-05
I-OHS	-4.4123e-05	(-7.6098e-05,-1.2148e-05)	3.5179e-05
I-LOAD	3.1849e-05	(1.6779e-05,4.6918e-05)	2.2872e-05
I-GT	2.4221e-04	(1.9592e-04,2.8849e-04)	1.2599e-04
I-AND	3.3352e-04	(2.7801e-04,3.8903e-04)	1.3160e-04
I-SHFT	4.1418e-04	(3.7183e-04,4.5652e-04)	3.0133e-04
I-ADD	2.1337e-04	(1.7960e-04,2.4714e-04)	1.0626e-04
I-RS	3.3435e-05	(1.1669e-06,6.5704e-05)	7.0426e-05

1048576-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	3.3597e-04	(2.2447e-04,4.4748e-04)	2.8627e-04
RS-LOAD	1.2453e-04	(5.5445e-05,1.9361e-04)	9.8571e-05
RS-ADD	5.9876e-04	(5.0649e-04,6.9102e-04)	3.3131e-04
RS-MUL	3.1657e-04	(1.9080e-04,4.4235e-04)	3.8226e-04
RS-SQRT	7.3616e-03	(7.1680e-03,7.5552e-03)	7.3492e-03
RS-LOG	2.9979e-02	(2.9427e-02,3.0532e-02)	2.8864e-02
RS-EXPI	8.2416e-03	(7.8137e-03,8.6695e-03)	6.0005e-03
RS-EXPS	9.6197e-02	(9.4757e-02,9.7636e-02)	9.0576e-02
RS-SIN	3.5806e-02	(3.5390e-02,3.6221e-02)	3.4005e-02
RS-GT	2.2284e-05	(-2.0339e-04,2.4795e-04)	1.9671e-04
RS-I	-5.8714e-06	(-1.9194e-04,1.8019e-04)	2.2688e-04
RS-MAX	1.0329e-03	(8.5506e-04,1.2106e-03)	1.1155e-03
RS-GTCONST	-3.7153e-05	(-2.2303e-04,1.4873e-04)	2.1043e-04
L-OHS	5.7968e-04	(4.0716e-04,7.5221e-04)	3.4965e-04
L-LOAD	1.2453e-04	(5.5445e-05,1.9361e-04)	9.8571e-05
L-OR	5.1054e-05	(-1.7442e-04,2.7653e-04)	2.2410e-04
L-NOT	3.5663e-05	(-1.8978e-04,2.6111e-04)	1.2543e-04
I-OHS	5.7968e-04	(4.0716e-04,7.5221e-04)	3.4965e-04
I-LOAD	1.2453e-04	(5.5445e-05,1.9361e-04)	9.8571e-05
I-GT	-8.8978e-06	(-2.3000e-04,2.1220e-04)	2.4771e-04
I-AND	4.1133e-05	(-1.8272e-04,2.6499e-04)	2.7285e-04
I-SHFT	7.4080e-04	(5.5465e-04,9.2695e-04)	9.7303e-04
I-ADD	5.9888e-04	(5.0816e-04,6.8959e-04)	4.8942e-04
I-RS	6.2676e-04	(4.7089e-04,7.8262e-04)	3.0081e-04

E.1.3 Y-MP C90/`f90` Operation Measurements

64-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	7.7672e-08	(6.9121e-08,8.6223e-08)	6.7078e-08
RS-LOAD	4.0519e-09	(2.2964e-09,5.8073e-09)	5.9411e-09
RS-ADD	1.6310e-07	(1.5533e-07,1.7086e-07)	1.4902e-07
RS-MUL	1.4519e-07	(1.3717e-07,1.5321e-07)	1.4513e-07
RS-SQRT	2.1033e-06	(2.0863e-06,2.1204e-06)	2.1083e-06
RS-LOG	5.7956e-06	(5.7606e-06,5.8306e-06)	5.7265e-06
RS-EXPI	1.5136e-06	(1.4990e-06,1.5282e-06)	1.5186e-06
RS-EXPS	2.2003e-05	(2.1874e-05,2.2133e-05)	2.1846e-05
RS-SIN	8.7512e-06	(8.6970e-06,8.8053e-06)	8.6477e-06
RS-GT	1.9750e-07	(1.8770e-07,2.0729e-07)	1.8043e-07
RS-I	1.5364e-07	(1.4449e-07,1.6278e-07)	1.4236e-07
RS-MAX	1.7024e-07	(1.5948e-07,1.8100e-07)	1.7020e-07
RS-GTCONST	1.5424e-07	(1.4513e-07,1.6335e-07)	1.4347e-07
L-OHS	-1.1960e-09	(-8.6040e-09,6.2119e-09)	3.1954e-09
L-LOAD	4.0519e-09	(2.2964e-09,5.8073e-09)	5.9411e-09
L-OR	2.6774e-07	(2.5772e-07,2.7776e-07)	2.5213e-07
L-NOT	2.1347e-07	(2.0368e-07,2.2327e-07)	2.0190e-07
I-OHS	-1.1960e-09	(-8.6040e-09,6.2119e-09)	3.1954e-09
I-LOAD	4.0519e-09	(2.2964e-09,5.8073e-09)	5.9411e-09
I-GT	8.1406e-08	(7.1950e-08,9.0862e-08)	7.2393e-08
I-AND	2.7590e-07	(2.6593e-07,2.8586e-07)	2.6086e-07
I-SHFT	3.9982e-07	(3.9008e-07,4.0955e-07)	3.8827e-07
I-ADD	1.0116e-07	(9.4887e-08,1.0744e-07)	8.6644e-08
I-RS	9.7921e-08	(8.7777e-08,1.0807e-07)	1.0247e-07

256-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	2.3064e-07	(2.1526e-07,2.4603e-07)	1.6144e-07
RS-LOAD	5.5955e-09	(1.0082e-09,1.0183e-08)	1.0464e-08
RS-ADD	5.5062e-07	(5.3647e-07,5.6476e-07)	5.8095e-07
RS-MUL	5.5383e-07	(5.4136e-07,5.6629e-07)	5.4445e-07
RS-SQRT	7.2001e-06	(7.1557e-06,7.2444e-06)	7.2477e-06
RS-LOG	2.0731e-05	(2.0597e-05,2.0866e-05)	2.0367e-05
RS-EXPI	5.5696e-06	(5.4922e-06,5.6471e-06)	5.4879e-06
RS-EXPS	7.7840e-05	(7.7239e-05,7.8440e-05)	7.6359e-05
RS-SIN	3.2928e-05	(3.2723e-05,3.3132e-05)	3.2105e-05
RS-GT	4.9745e-07	(4.7913e-07,5.1578e-07)	5.0206e-07
RS-I	2.7352e-07	(2.1528e-07,3.3176e-07)	3.9123e-07
RS-MAX	1.1707e-06	(1.1494e-06,1.1920e-06)	1.2027e-06
RS-GTCONST	3.7144e-07	(3.5575e-07,3.8714e-07)	3.8910e-07
L-OHS	2.8072e-07	(2.6783e-07,2.9362e-07)	2.4208e-07
L-LOAD	5.5955e-09	(1.0082e-09,1.0183e-08)	1.0464e-08
L-OR	3.6186e-07	(3.4355e-07,3.8018e-07)	3.6616e-07
L-NOT	2.9736e-07	(2.8005e-07,3.1467e-07)	3.1818e-07
I-OHS	2.8072e-07	(2.6783e-07,2.9362e-07)	2.4208e-07
I-LOAD	5.5955e-09	(1.0082e-09,1.0183e-08)	1.0464e-08
I-GT	3.6241e-07	(3.4499e-07,3.7983e-07)	3.8438e-07
I-AND	3.5406e-07	(3.3423e-07,3.7389e-07)	3.7446e-07
I-SHFT	1.0847e-06	(1.0518e-06,1.1177e-06)	1.0807e-06
I-ADD	5.3569e-07	(5.2673e-07,5.4464e-07)	5.2732e-07
I-RS	3.0919e-07	(1.9897e-07,4.1941e-07)	4.8935e-07

1024-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	1.1593e-06	(8.3619e-07,1.4824e-06)	1.0046e-06
RS-LOAD	7.5807e-08	(2.0874e-09,1.4953e-07)	7.9707e-08
RS-ADD	1.8123e-06	(1.7281e-06,1.8965e-06)	1.8674e-06
RS-MUL	2.0565e-06	(1.4358e-06,2.6772e-06)	1.7346e-06
RS-SQRT	2.9593e-05	(2.9004e-05,3.0183e-05)	2.8949e-05
RS-LOG	8.5242e-05	(8.3922e-05,8.6561e-05)	8.3398e-05
RS-EXPI	2.2634e-05	(2.2147e-05,2.3121e-05)	2.2155e-05
RS-EXPS	3.0984e-04	(3.0728e-04,3.1241e-04)	3.0278e-04
RS-SIN	1.3575e-04	(1.3343e-04,1.3808e-04)	1.3155e-04
RS-GT	5.2837e-07	(2.2542e-07,8.3133e-07)	1.1093e-06
RS-I	-1.8072e-07	(-4.4197e-07,8.0522e-08)	7.2623e-07
RS-MAX	5.1078e-06	(4.7408e-06,5.4749e-06)	4.9860e-06
RS-GTCONST	-1.5979e-07	(-4.2251e-07,1.0294e-07)	7.0162e-07
L-OHS	2.5926e-06	(2.3433e-06,2.8419e-06)	1.6138e-06
L-LOAD	7.5807e-08	(2.0874e-09,1.4953e-07)	7.9707e-08
L-OR	7.1564e-08	(-2.2110e-07,3.6422e-07)	7.5858e-07
L-NOT	-3.4195e-07	(-6.3239e-07,-5.1505e-08)	5.6735e-07
I-OHS	2.5926e-06	(2.3433e-06,2.8419e-06)	1.6138e-06
I-LOAD	7.5807e-08	(2.0874e-09,1.4953e-07)	7.9707e-08
I-GT	1.4946e-07	(-1.4143e-07,4.4034e-07)	1.0251e-06
I-AND	5.2872e-07	(-2.7671e-07,1.3342e-06)	1.0036e-06
I-SHFT	2.3287e-06	(2.0671e-06,2.5904e-06)	3.2921e-06
I-ADD	1.9353e-06	(1.8498e-06,2.0208e-06)	1.8795e-06
I-RS	1.2846e-06	(9.4953e-07,1.6197e-06)	1.3533e-06

4096-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	8.4964e-06	(3.1835e-06,1.3809e-05)	3.8770e-06
RS-LOAD	-6.7174e-08	(-6.0143e-07,4.6708e-07)	7.1384e-09
RS-ADD	7.9178e-06	(3.2910e-06,1.2545e-05)	7.5798e-06
RS-MUL	7.0129e-06	(1.4081e-06,1.2618e-05)	7.5633e-06
RS-SQRT	1.5322e-04	(1.0140e-04,2.0504e-04)	1.1637e-04
RS-LOG	5.2656e-04	(3.5685e-04,6.9627e-04)	3.2739e-04
RS-EXPI	9.5011e-05	(7.9826e-05,1.1020e-04)	8.9182e-05
RS-EXPS	1.2117e-03	(1.2029e-03,1.2205e-03)	1.2012e-03
RS-SIN	5.1972e-04	(5.1363e-04,5.2580e-04)	5.2123e-04
RS-GT	1.5243e-06	(-2.7014e-07,3.3188e-06)	3.9267e-06
RS-I	-8.6923e-07	(-2.4038e-06,6.6530e-07)	1.8139e-06
RS-MAX	1.9246e-05	(1.2959e-05,2.5534e-05)	2.0787e-05
RS-GTCONST	-1.0850e-06	(-2.6194e-06,4.4952e-07)	1.8161e-06
L-OHS	1.0679e-05	(9.2416e-06,1.2116e-05)	7.7471e-06
L-LOAD	-6.7174e-08	(-6.0143e-07,4.6708e-07)	7.1384e-09
L-OR	3.6952e-06	(1.7893e-06,5.6012e-06)	3.9318e-06
L-NOT	4.1730e-07	(-1.4740e-06,2.3086e-06)	1.8227e-06
I-OHS	1.0679e-05	(9.2416e-06,1.2116e-05)	7.7471e-06
I-LOAD	-6.7174e-08	(-6.0143e-07,4.6708e-07)	7.1384e-09
I-GT	3.9264e-06	(1.9719e-06,5.8808e-06)	4.0790e-06
I-AND	1.4667e-06	(-3.3259e-07,3.2660e-06)	3.8207e-06
I-SHFT	9.0555e-06	(7.5181e-06,1.0593e-05)	1.1882e-05
I-ADD	8.3629e-06	(7.7523e-06,8.9735e-06)	7.8707e-06
I-RS	1.2659e-06	(-4.0741e-06,6.6059e-06)	5.6902e-06

16384-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	1.6212e-05	(1.4848e-05,1.7575e-05)	1.1504e-05
RS-LOAD	2.3990e-06	(1.4722e-06,3.3258e-06)	1.1039e-06
RS-ADD	2.7846e-05	(2.6385e-05,2.9308e-05)	3.0279e-05
RS-MUL	2.8202e-05	(2.7012e-05,2.9391e-05)	3.1273e-05
RS-SQRT	4.6423e-04	(4.6122e-04,4.6725e-04)	4.6911e-04
RS-LOG	1.3482e-03	(1.3401e-03,1.3563e-03)	1.3276e-03
RS-EXPI	3.5637e-04	(3.5377e-04,3.5898e-04)	3.6113e-04
RS-EXPS	4.8315e-03	(4.8056e-03,4.8573e-03)	4.7998e-03
RS-SIN	2.1120e-03	(2.0995e-03,2.1244e-03)	2.1015e-03
RS-GT	1.1789e-05	(9.7013e-06,1.3877e-05)	1.3615e-05
RS-I	1.5673e-05	(1.4188e-05,1.7158e-05)	8.7971e-06
RS-MAX	8.0986e-05	(7.8609e-05,8.3363e-05)	8.6655e-05
RS-GTCONST	5.9360e-06	(4.6395e-06,7.2325e-06)	7.8178e-06
L-OHS	3.0538e-05	(2.9662e-05,3.1415e-05)	2.9713e-05
L-LOAD	2.3990e-06	(1.4722e-06,3.3258e-06)	1.1039e-06
L-OR	1.0746e-05	(8.6657e-06,1.2827e-05)	1.2865e-05
L-NOT	3.2969e-06	(1.2352e-06,5.3586e-06)	6.4630e-06
I-OHS	3.0538e-05	(2.9662e-05,3.1415e-05)	2.9713e-05
I-LOAD	2.3990e-06	(1.4722e-06,3.3258e-06)	1.1039e-06
I-GT	1.3333e-05	(1.1247e-05,1.5419e-05)	1.5231e-05
I-AND	1.2824e-05	(1.0744e-05,1.4904e-05)	1.4606e-05
I-SHFT	4.5363e-05	(4.4021e-05,4.6705e-05)	4.7317e-05
I-ADD	2.7769e-05	(2.6652e-05,2.8887e-05)	2.8087e-05
I-RS	2.0458e-05	(1.8779e-05,2.2137e-05)	2.5728e-05

65536-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	6.9076e-05	(3.9834e-05,9.8319e-05)	5.3843e-05
RS-LOAD	4.0849e-05	(1.8241e-05,6.3456e-05)	7.7244e-06
RS-ADD	6.6897e-05	(3.7205e-05,9.6588e-05)	1.1370e-04
RS-MUL	6.3056e-05	(3.4066e-05,9.2046e-05)	1.0990e-04
RS-SQRT	1.8179e-03	(1.7794e-03,1.8563e-03)	1.8622e-03
RS-LOG	5.4343e-03	(5.2354e-03,5.6332e-03)	5.2403e-03
RS-EXPI	1.3872e-03	(1.3494e-03,1.4250e-03)	1.4313e-03
RS-EXPS	2.0260e-02	(1.8599e-02,2.1922e-02)	1.9055e-02
RS-SIN	8.4210e-03	(8.3158e-03,8.5262e-03)	8.2525e-03
RS-GT	2.8216e-05	(-2.3114e-05,7.9545e-05)	4.9073e-05
RS-I	3.0313e-05	(1.8284e-07,6.0443e-05)	3.6394e-05
RS-MAX	2.8294e-04	(2.2838e-04,3.3750e-04)	3.2485e-04
RS-GTCONST	2.7125e-05	(-3.0021e-06,5.7252e-05)	3.3122e-05
L-OHS	8.5923e-05	(6.6031e-05,1.0582e-04)	1.1078e-04
L-LOAD	4.0849e-05	(1.8241e-05,6.3456e-05)	7.7244e-06
L-OR	2.7910e-05	(-2.2220e-05,7.8040e-05)	5.7355e-05
L-NOT	-1.2924e-05	(-6.2330e-05,3.6482e-05)	2.6783e-05
I-OHS	8.5923e-05	(6.6031e-05,1.0582e-04)	1.1078e-04
I-LOAD	4.0849e-05	(1.8241e-05,6.3456e-05)	7.7244e-06
I-GT	2.3737e-05	(-2.5705e-05,7.3179e-05)	5.4065e-05
I-AND	2.5732e-05	(-2.3693e-05,7.5158e-05)	5.6031e-05
I-SHFT	1.8802e-04	(1.5781e-04,2.1822e-04)	1.9340e-04
I-ADD	7.8611e-05	(5.4785e-05,1.0244e-04)	1.1550e-04
I-RS	4.6014e-05	(9.0392e-06,8.2989e-05)	9.2025e-05

262144-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	2.9302e-04	(2.6837e-04,3.1768e-04)	2.5529e-04
RS-LOAD	8.1290e-06	(-2.2393e-05,3.8651e-05)	2.0708e-05
RS-ADD	4.6972e-04	(4.3729e-04,5.0216e-04)	4.4554e-04
RS-MUL	4.7771e-04	(4.4461e-04,5.1081e-04)	4.5045e-04
RS-SQRT	7.4224e-03	(7.3718e-03,7.4731e-03)	7.4302e-03
RS-LOG	2.1447e-02	(2.1338e-02,2.1556e-02)	2.1186e-02
RS-EXPI	5.8146e-03	(5.7363e-03,5.8930e-03)	5.7052e-03
RS-EXPS	7.9228e-02	(7.8859e-02,7.9598e-02)	7.7632e-02
RS-SIN	3.3626e-02	(3.3389e-02,3.3862e-02)	3.3001e-02
RS-GT	1.7263e-04	(1.0140e-04,2.4385e-04)	1.9483e-04
RS-I	7.3693e-05	(2.6193e-05,1.2119e-04)	1.2684e-04
RS-MAX	1.3426e-03	(1.2599e-03,1.4253e-03)	1.3001e-03
RS-GTCONST	5.1389e-05	(3.9373e-06,9.8840e-05)	1.1569e-04
L-OHS	5.5494e-04	(5.1876e-04,5.9113e-04)	4.7513e-04
L-LOAD	8.1290e-06	(-2.2393e-05,3.8651e-05)	2.0708e-05
L-OR	2.0283e-04	(1.3169e-04,2.7397e-04)	2.3420e-04
L-NOT	5.0269e-05	(-2.0780e-05,1.2132e-04)	9.9762e-05
I-OHS	5.5494e-04	(5.1876e-04,5.9113e-04)	4.7513e-04
I-LOAD	8.1290e-06	(-2.2393e-05,3.8651e-05)	2.0708e-05
I-GT	2.0211e-04	(1.3080e-04,2.7341e-04)	2.3021e-04
I-AND	1.7438e-04	(1.0321e-04,2.4555e-04)	2.0328e-04
I-SHFT	6.8593e-04	(6.3809e-04,7.3376e-04)	7.5148e-04
I-ADD	4.5213e-04	(4.1617e-04,4.8809e-04)	4.5194e-04
I-RS	3.1995e-04	(2.8053e-04,3.5938e-04)	3.3668e-04

1048576-Element Array Size:

Parameter	Mean Value (seconds)	95-Percent Confidence Interval (seconds)	Minimum Value (seconds)
RS-OHS	1.0534e-03	(1.0103e-03,1.0965e-03)	8.5912e-04
RS-LOAD	8.9373e-05	(7.1107e-05,1.0764e-04)	1.1358e-04
RS-ADD	1.8633e-03	(1.8195e-03,1.9070e-03)	1.8296e-03
RS-MUL	1.8621e-03	(1.8198e-03,1.9045e-03)	1.8606e-03
RS-SQRT	2.9715e-02	(2.9608e-02,2.9822e-02)	2.9849e-02
RS-LOG	8.5948e-02	(8.5743e-02,8.6153e-02)	8.5314e-02
RS-EXPI	2.2836e-02	(2.2728e-02,2.2943e-02)	2.2920e-02
RS-EXPS	3.1034e-01	(3.1008e-01,3.1059e-01)	3.0884e-01
RS-SIN	1.3435e-01	(1.3416e-01,1.3453e-01)	1.3337e-01
RS-GT	8.5067e-04	(7.5063e-04,9.5072e-04)	8.5305e-04
RS-I	4.0610e-04	(2.6966e-04,5.4254e-04)	4.8973e-04
RS-MAX	5.2289e-03	(5.1623e-03,5.2956e-03)	5.3146e-03
RS-GTCONST	3.2674e-04	(2.4352e-04,4.0997e-04)	5.0020e-04
L-OHS	2.0961e-03	(2.0165e-03,2.1757e-03)	1.8702e-03
L-LOAD	8.9373e-05	(7.1107e-05,1.0764e-04)	1.1358e-04
L-OR	7.2839e-04	(6.3806e-04,8.1872e-04)	8.1305e-04
L-NOT	2.1452e-04	(1.2580e-04,3.0325e-04)	3.7012e-04
I-OHS	2.0961e-03	(2.0165e-03,2.1757e-03)	1.8702e-03
I-LOAD	8.9373e-05	(7.1107e-05,1.0764e-04)	1.1358e-04
I-GT	7.4184e-04	(6.5142e-04,8.3226e-04)	8.3083e-04
I-AND	7.7592e-04	(6.8448e-04,8.6736e-04)	8.1798e-04
I-SHFT	2.8067e-03	(2.7216e-03,2.8918e-03)	3.0026e-03
I-ADD	1.8760e-03	(1.7453e-03,2.0067e-03)	1.8173e-03
I-RS	1.3353e-03	(1.2864e-03,1.3841e-03)	1.4862e-03

E.2 Optimization Characterizer Measurements

The following tables describe the results of compiler tests on each of our architectures, except for the CM-5/`cmf` pair, which did not pass any tests, except for the elimination of dead code and common subexpression elimination from array statements.

Section 6.2 explains our testing method. The following tables give the 95-percent confidence intervals of the run time of the *Naive* and *Opt* code segments, the confidence interval of R computed from these run times, R_p, the lowest predicted value of R in the case that *Naive* was not optimized, and our conclusion as to whether or not the compiler applied the optimization in the specific test case.

Note that all tests marked with a bold-faced asterix (*****) use a *Naive*/*Cripple* test. Additionally, some tests crashed the compiler in question and are indicated by dashes in the corresponding rows.

E.2.1 CM-5/`cmf-cmax` Compiler Optimization Results

Test	*Naive* Run Time 95% Conf. Int. (seconds)	*Opt* /*Cripple* Run Time 95% Conf. Int. (seconds)	R 90% Conf. Int. (seconds)	R_p	Applied?
collapses depth 2 loops	(2.47e-5, 3.68e-5)	(3.13e-5,3.95e-5)	(0.626,1.17)	1.2	**yes**
collapses depth 3 loops	(2.38e-5, 3.52e-5)	(3.16e-5,3.62e-5)	(0.657,1.11)	1.2	**yes**
collapses depth 4 loops	(3.00e-5, 3.53e-5)	(2.59e-5,3.85e-5)	(0.78,1.36)	1.2	maybe
common subexpression elimination in non-F90 array operations	(6.38e-5, 7.04e-5)	(5.64e-5,6.50e-5)	(0.981,1.25)	1.3	**yes**
common subexpression elimination F90 array operations	(5.59e-5, 6.42e-5)	(5.69e-5,6.30e-5)	(0.886,1.13)	1.3	**yes**
dead scalar code eliminated	(-1.59e-5,-4.65e-6)	(2.71e-5,3.17e-5)	(-5.03e-1,-1.71e-1)	1.1	**yes**
dead array code eliminated	(2.76e-5, 3.14e-5)	(2.71e-5,3.17e-5)	(0.87,1.16)	1.5	**yes**
substitutes value for variable assigned the constant 0, when the variable is assigned in the same basic block	(3.51e-5, 3.94e-5)	(3.38e-5,3.54e-5)	(0.993,1.16)	2	**yes**
substitutes value for variable assigned the constant 1, when the variable is assigned in the same basic block	(1.96e-4, 2.13e-4)	(1.95e-4,2.05e-4)	(0.953,1.09)	2	**yes**
substitutes value for variable assigned the constant 0, when the variable is assigned outside the basic block	(1.43e-4, 1.67e-4)	(3.38e-5,3.54e-5)	(4.04,4.92)	2	no
substitutes value for variable assigned the constant 1, when the variable is assigned outside the basic block	(1.92e-4, 2.07e-4)	(1.95e-4,2.05e-4)	(0.937,1.06)	2	**yes**
substitutes value for variable assigned a constant symbolic expression, when the variable is assigned in the same basic block	(3.70e-4, 4.14e-4)	(3.63e-4,3.86e-4)	(0.959,1.14)	2	**yes**
substitutes value for variable assigned a constant symbolic expression, when the variable is assigned outside the basic block	(6.02e-4, 6.15e-4)	(3.73e-4,4.14e-4)	(1.45,1.65)	2	**yes**
semantic analysis - produces efficient code when the substituted variable has value 0	(2.10, 2.14)	(3.38e-5,3.54e-5)	(5.95e+4,6.32e+4)	2	no
semantic analysis - produces efficient code when the substituted variable has value 1	(2.10, 2.14)	(1.95e-4,2.05e-4)	(1.02e+4,1.1e+4)	2	no
eliminates an induction variable indexing a 1-d array in a simple loop	(2.11e-5, 2.76e-5)	(2.18e-5,2.62e-5)	(0.805,1.26)	2	**yes**
eliminates an induction variable in a 2-d loop indexing a 1-d array	-	-	-	-	-
eliminates an induction variable in a 2-d loop indexing a 1-d array, where the induction variable is assigned a value in the outer loop	-	-	-	-	-
eliminates an induction variable indexing a 2-d array	(1.55e-5, 8.78e-5)	(3.94e-5,4.78e-5)	(0.325,2.22)	2	maybe
inline substitution of functions returning a real value - 1 level deep	(1.63e-1, 1.65e-1)	(3.02e-5,3.40e-5)	(4.81e+3,5.46e+3)	2	no
inline substitution of functions returning a real value - 2 levels deep	(1.71e-1, 1.78e-1)	(2.88e-5,3.09e-5)	(5.55e+3,6.18e+3)	2	no
inline substitution of functions returning an integer value - 1 level deep	(6.52e-2, 6.77e-2)	(3.28e-4,3.54e-4)	(184,206)	2	no

Test	*Naive* Run Time 95% Conf. Int. (seconds)	*Opt* /*Cripple* Run Time 95% Conf. Int. (seconds)	R 90% Conf. Int. (seconds)	R_p	Applied?
inline substitution of functions returning an integer value - 2 levels deep	(3.29e-2, 3.37e-2)	(3.24e-4,3.30e-4)	(99.7,104)	2	no
code motion - code moved upward one level where possible	(2.60e-1, 2.64e-1)	(2.60e-4,2.88e-4)	(902,1.02e+3)	2	no
code motion - code moved upward two levels where possible	(2.61e-1, 2.63e-1)	(2.70e-4,2.84e-4)	(921,974)	2	no
substitutes efficient code for array-based reccurences *	(1.36, 1.38)	(2.17,2.20)	(0.616,0.635)	0.5	no
substitutes efficient code for array sum computations *	(1.78e-4, 1.80e-4)	(8.10e-1,8.21e-1)	(2.16e-4, 2.22e-4)	0.5	**yes**
substitutes efficient code for array product computations *	(1.81e-4, 1.83e-4)	(8.11e-1,8.18e-1)	(2.21e-4, 2.26e-4)	0.5	**yes**
scalar expansion in a single loop nest	(3.20e-5, 3.84e-5)	(3.76e-5,4.55e-5)	(0.704,1.02)	2	**yes**
scalar expansion in a double loop nest	(3.82e-5, 5.34e-5)	(2.01e-5,3.43e-5)	(1.11,2.66)	2	maybe
array privatization, where the temporary array is defined in the same loop as it is used	(2.08e-5, 3.53e-5)	(2.01e-5,3.43e-5)	(0.607,1.76)	2	**yes**
array privatization, where the temporary array is defined in a loop neighboring the loop where it is used	(3.22e-5, 5.45e-5)	(2.01e-5,3.43e-5)	(0.94,2.71)	2	maybe
parallelizes a dependency-free 1-d loop, where the source and sink elements are in seperate regions of the array	(3.92e-4, 4.03e-4)	(3.67e-4,3.93e-4)	(0.996,1.1)	2	**yes**
parallelizes a dependency-free 1-d loop, where the source and sink elements are interleaved	(2.12e-4, 2.17e-4)	(2.17e-4,2.31e-4)	(0.917,1)	2	**yes**
parallelizes a dependency-free 1-d loop, where the source and sink element indices are defined as linear functions of the loop variable	(1.07e-3, 1.10e-3)	(1.13e-3,1.21e-3)	(0.884,0.974)	2	**yes**
parallelizes a dependency-free 2-d loop, where the source and sink elements are defined as linear functions of the loop variable	(9.62e-4, 1.30e-3)	(1.04e-3,1.07e-3)	(0.897,1.25)	2	**yes**
removes stores to temporary arrays	(3.43e-5, 4.29e-5)	(3.47e-5,4.20e-5)	(0.816,1.24)	1.1	maybe
parallelizes the loop assignment of a arithmetic function of the loop variable to an array *	(3.03e-5, 3.96e-5)	(4.19e-2,5.07e-2)	(5.98e-4, 9.45e-4)	0.5	**yes**
parallelizes the loop assignment of a function of the loop variable to an array, where the function involves an intrinsic *	(4.17e-5, 5.10e-5)	(4.65e-2,4.75e-2)	(8.77e-4, 1.10e-3)	0.5	**yes**
parallelizes the loop assignment of a function of the loop variable to an array, where the function involves an integer leaf function *	(6.71e-2, 6.82e-2)	(4.19e-2,4.30e-2)	(1.56,1.63)	0.5	no
parallelizes a serially-expressed gather	(5.19e-4, 5.63e-4)	(5.15e-4,5.23e-4)	(0.992,1.09)	2	**yes**
parallelizes a serially-expressed scatter	(6.26e-2, 6.56e-2)	(2.70e-4,2.85e-4)	(219,243)	2	no
parallelizes a serially-expressed computation involving two permutation arrays	(1.13e-1, 1.14e-1)	(1.27e-3,1.32e-3)	(85.4,89.7)	2	no
parallelizes a serially-expressed bin summation *	(6.21e-2, 6.31e-2)	(1.08e-1,1.10e-1)	(0.567,0.583)	0.5	no
parallelizes a serially-expressed all up to element array sum *	(6.21e-2, 6.39e-2)	(6.22e-2,6.42e-2)	(0.968,1.03)	0.5	no
parallelizes a serially-expressed `where` operation	(2.45e-5, 3.26e-5)	(2.42e-5,3.75e-5)	(0.655,1.35)	2	**yes**

E.2.2 Y-MP C90/`cf77` Compiler Optimization Results

Test	*Naive* Run Time 95% Conf. Int. (seconds)	*Opt* /*Cripple* Run Time 95% Conf. Int. (seconds)	R 90% Conf. Int. (seconds)	R_p	Applied?
collapses depth 2 loops	(1.45e-5, 1.64e-5)	(1.49e-5,1.64e-5)	(0.884,1.1)	1.2	**yes**
collapses depth 3 loops	(1.25e-5, 1.39e-5)	(1.58e-5,1.74e-5)	(0.717,0.88)	1.2	**yes**
collapses depth 4 loops	(1.37e-5, 1.39e-5)	(1.98e-5,2.22e-5)	(0.616,0.703)	1.2	**yes**
common subexpression elimination in non-F90 array operations	(5.47e-3, 5.51e-3)	(5.98e-3,6.04e-3)	(0.905,0.922)	1.3	**yes**
common subexpression elimination F90 array operations	(5.49e-3, 5.68e-3)	(5.99e-3,6.05e-3)	(0.907,0.948)	1.3	**yes**
dead scalar code eliminated	(2.20e-3, 2.41e-3)	(8.51e-4,8.63e-4)	(2.55,2.83)	1.1	no
dead array code eliminated	(2.51e-3, 2.53e-3)	(8.51e-4,8.63e-4)	(2.91,2.98)	1.5	no
substitutes value for variable assigned the constant 0, when the variable is assigned in the same basic block	(8.37e-4, 8.46e-4)	(8.52e-4,8.81e-4)	(0.95,0.994)	2	**yes**
substitutes value for variable assigned the constant 1, when the variable is assigned in the same basic block	(4.74e-3, 4.79e-3)	(4.77e-3,4.82e-3)	(0.984,1.01)	2	**yes**
substitutes value for variable assigned the constant 0, when the variable is assigned outside the basic block	(8.92e-4, 9.85e-4)	(8.52e-4,8.81e-4)	(1.01,1.16)	2	**yes**
substitutes value for variable assigned the constant 1, when the variable is assigned outside the basic block	(4.76e-3, 4.82e-3)	(4.77e-3,4.82e-3)	(0.988,1.01)	2	**yes**
substitutes value for variable assigned a constant symbolic expression, when the variable is assigned in the same basic block	(5.02e-4, 5.95e-4)	(4.95e-4,5.72e-4)	(0.877,1.2)	2	**yes**
substitutes value for variable assigned a constant symbolic expression, when the variable is assigned outside the basic block	(4.29e-4, 4.73e-4)	(4.35e-4,4.81e-4)	(0.892,1.09)	2	**yes**
semantic analysis - produces efficient code when the substituted variable has value 0	(1.40e-3, 1.68e-3)	(8.52e-4,8.81e-4)	(1.59,1.98)	2	**yes**
semantic analysis - produces efficient code when the substituted variable has value 1	(4.81e-3, 4.99e-3)	(4.77e-3,4.82e-3)	(0.999,1.05)	2	**yes**
eliminates an induction variable indexing a 1-d array in a simple loop	(1.25e-3, 1.61e-3)	(1.17e-3,1.37e-3)	(0.913,1.37)	2	**yes**
eliminates an induction variable in a 2-d loop indexing a 1-d array	(1.17e-2, 1.20e-2)	(1.17e-3,1.37e-3)	(8.56,10.3)	2	no
eliminates an induction variable in a 2-d loop indexing a 1-d array, where the induction variable is assigned a value in the outer loop	(1.16e-2, 1.19e-2)	(1.17e-3,1.37e-3)	(8.51,10.2)	2	no
eliminates an induction variable indexing a 2-d array	(2.80e-3, 2.99e-3)	(1.19e-3,1.41e-3)	(1.99,2.51)	2	maybe
inline substitution of functions returning a real value - 1 level deep	(9.42e-5, 1.04e-4)	(7.91e-5,9.50e-5)	(0.992,1.32)	2	**yes**
inline substitution of functions returning a real value - 2 levels deep	(4.01e-2, 4.10e-2)	(1.31e-4,1.56e-4)	(258,312)	2	no
inline substitution of functions returning an integer value - 1 level deep	(2.40e-2, 2.49e-2)	(1.05e-4,1.19e-4)	(201,238)	2	no
inline substitution of functions returning an integer value - 2 levels deep	(2.06e-2, 2.11e-2)	(6.30e-5,7.40e-5)	(279,335)	2	no
code motion - code moved upward one level where possible	(2.58e-1, 2.59e-1)	(7.87e-5,8.01e-5)	(3.21e+3,3.29e+3)	2	no
code motion - code moved upward two levels where possible	(1.60e-3, 1.61e-3)	(9.60e-5,1.04e-4)	(15.3,16.8)	2	no
substitutes efficient code for array-based reccurences *	(5.64e-2, 5.66e-2)	(1.10e-1,1.10e-1)	(0.514,0.517)	0.5	no
substitutes efficient code for array sum computations *	(1.05e-3, 1.22e-3)	(5.48e-2,5.53e-2)	(0.0189,0.0223)	0.5	**yes**
substitutes efficient code for array product computations *	(6.89e-4, 7.19e-4)	(5.36e-2,5.38e-2)	(0.0128,0.0134)	0.5	**yes**
scalar expansion in a single loop nest	(1.17e-3, 1.18e-3)	(1.50e-3,1.62e-3)	(0.723,0.787)	2	**yes**
scalar expansion in a double loop nest	(1.53e-3, 1.54e-3)	(1.53e-3,1.55e-3)	(0.981,1.01)	2	**yes**
array privatization, where the temporary array is defined in the same loop as it is used	(6.09e-3, 6.15e-3)	(1.53e-3,1.55e-3)	(3.92,4.01)	2	no
array privatization, where the temporary array is defined in a loop neighboring the loop where it is used	(4.10e-3, 4.14e-3)	(1.53e-3,1.55e-3)	(2.64,2.7)	2	no

Test	*Naive* Run Time 95% Conf. Int. (seconds)	*Opt* /*Cripple* Run Time 95% Conf. Int. (seconds)	R 90% Conf. Int. (seconds)	R_p	Applied?
parallelizes a dependency-free 1-d loop, where the source and sink elements are in seperate regions of the array	(4.89e-4, 5.41e-4)	(4.89e-4,4.95e-4)	(0.989,1.11)	2	**yes**
parallelizes a dependency-free 1-d loop, where the source and sink elements are interleaved	(7.17e-4, 7.94e-4)	(6.67e-4,6.76e-4)	(1.06,1.19)	2	**yes**
parallelizes a dependency-free 1-d loop, where the source and sink element indices are defined as linear functions of the loop variable	(1.92e-3, 1.94e-3)	(1.92e-3,1.95e-3)	(0.985,1.01)	2	**yes**
parallelizes a dependency-free 2-d loop, where the source and sink elements are defined as linear functions of the loop variable	(5.30e-3, 5.35e-3)	(7.28e-3,7.34e-3)	(0.722,0.734)	2	**yes**
removes stores to temporary arrays	(1.26e-3, 1.27e-3)	(1.16e-3,1.18e-3)	(1.07,1.09)	1.1	**yes**
parallelizes the loop assignment of a arithmetic function of the loop variable to an array *	(7.11e-5, 7.27e-5)	(3.11e-3,3.14e-3)	(0.0226,0.0234)	0.5	**yes**
parallelizes the loop assignment of a function of the loop variable to an array, where the function involves an intrinsic *	(6.76e-4, 7.11e-4)	(4.19e-2,4.20e-2)	(0.0161,0.017)	0.5	**yes**
parallelizes the loop assignment of a function of the loop variable to an array, where the function involves an integer leaf function *	(1.35e-4, 1.36e-4)	(1.55e-3,1.56e-3)	(0.0864,0.0881)	0.5	**yes**
parallelizes a serially-expressed gather	(4.73e-5, 4.85e-5)	(4.71e-5,4.77e-5)	(0.992,1.03)	2	**yes**
parallelizes a serially-expressed scatter	(6.42e-4, 6.49e-4)	(6.46e-4,6.53e-4)	(0.983,1.01)	2	**yes**
parallelizes a serially-expressed computation involving two permutation arrays	(4.34e-3, 4.39e-3)	(4.26e-3,4.31e-3)	(1.01,1.03)	2	**yes**
parallelizes a serially-expressed bin summation *	(4.14e-3, 4.22e-3)	(7.37e-3,7.45e-3)	(0.555,0.573)	0.5	no
parallelizes a serially-expressed all up to element array sum *	(2.03e-3, 2.07e-3)	(3.39e-3,3.43e-3)	(0.591,0.611)	0.5	no
parallelizes a serially-expressed `where` operation	(4.85e-5, 4.90e-5)	(4.85e-5,4.91e-5)	(0.988,1.01)	2	**yes**

E.2.3 Y-MP C90/`f90` Compiler Optimization Results

Test	*Naive* Run Time 95% Conf. Int. (seconds)	*Opt* /*Cripple* Run Time 95% Conf. Int. (seconds)	R 90% Conf. Int. (seconds)	R_p	Applied?
collapses depth 2 loops	(1.65e-5, 1.68e-5)	(1.62e-5,1.64e-5)	(1,1.04)	1.2	**yes**
collapses depth 3 loops	(1.87e-5, 1.91e-5)	(1.89e-5,1.92e-5)	(0.976,1.01)	1.2	**yes**
collapses depth 4 loops	(2.53e-5, 2.57e-5)	(2.53e-5,2.57e-5)	(0.986,1.01)	1.2	**yes**
common subexpression elimination in non-F90 array operations	(2.42e-2, 2.44e-2)	(2.44e-2,2.46e-2)	(0.985,1)	1.3	**yes**
common subexpression elimination F90 array operations	(2.43e-2, 2.45e-2)	(2.42e-2,2.44e-2)	(0.997,1.01)	1.3	**yes**
dead scalar code eliminated	(3.93e-3, 3.98e-3)	(3.87e-3,3.93e-3)	(1,1.03)	1.1	**yes**
dead array code eliminated	(3.91e-3, 3.97e-3)	(3.87e-3,3.93e-3)	(0.997,1.03)	1.5	**yes**
substitutes value for variable assigned the constant 0, when the variable is assigned in the same basic block	(3.89e-3, 3.94e-3)	(3.94e-3,3.99e-3)	(0.975,1)	2	**yes**
substitutes value for variable assigned the constant 1, when the variable is assigned in the same basic block	(4.71e-3, 4.76e-3)	(4.71e-3,4.76e-3)	(0.99,1.01)	2	**yes**
substitutes value for variable assigned the constant 0, when the variable is assigned outside the basic block	(3.93e-3, 3.99e-3)	(3.94e-3,3.99e-3)	(0.986,1.01)	2	**yes**
substitutes value for variable assigned the constant 1, when the variable is assigned outside the basic block	(4.73e-3, 4.79e-3)	(4.71e-3,4.76e-3)	(0.994,1.02)	2	**yes**
substitutes value for variable assigned a constant symbolic expression, when the variable is assigned in the same basic block	(1.94e-3, 1.97e-3)	(1.91e-3,1.94e-3)	(1,1.03)	2	**yes**
substitutes value for variable assigned a constant symbolic expression, when the variable is assigned outside the basic block	(1.93e-3, 1.95e-3)	(1.91e-3,1.94e-3)	(0.993,1.02)	2	**yes**
semantic analysis - produces efficient code when the substituted variable has value 0	(4.63e-3, 4.71e-3)	(3.94e-3,3.99e-3)	(1.16,1.19)	2	**yes**
semantic analysis - produces efficient code when the substituted variable has value 1	(4.69e-3, 4.75e-3)	(4.71e-3,4.76e-3)	(0.985,1.01)	2	**yes**
eliminates an induction variable indexing a 1-d array in a simple loop	(3.88e-3, 3.93e-3)	(3.89e-3,3.94e-3)	(0.985,1.01)	2	**yes**
eliminates an induction variable in a 2-d loop indexing a 1-d array	(3.89e-3, 3.95e-3)	(3.89e-3,3.94e-3)	(0.988,1.02)	2	**yes**
eliminates an induction variable in a 2-d loop indexing a 1-d array, where the induction variable is assigned a value in the outer loop	-	-	-	-	-
eliminates an induction variable indexing a 2-d array	(3.89e-3, 3.95e-3)	(4.04e-3,4.14e-3)	(0.939,0.977)	2	**yes**
inline substitution of functions returning a real value - 1 level deep	(4.84e-2, 4.86e-2)	(2.45e-4,2.48e-4)	(195,198)	2	no
inline substitution of functions returning a real value - 2 levels deep	(8.99e-2, 9.10e-2)	(3.16e-4,3.21e-4)	(281,288)	2	no
inline substitution of functions returning an integer value - 1 level deep	(4.95e-2, 4.98e-2)	(2.80e-4,2.83e-4)	(175,178)	2	no
inline substitution of functions returning an integer value - 2 levels deep	(4.04e-2, 4.09e-2)	(1.70e-4,1.72e-4)	(235,241)	2	no
code motion - code moved upward one level where possible	(6.25e-3, 6.32e-3)	(2.58e-4,2.62e-4)	(23.9,24.5)	2	no
code motion - code moved upward two levels where possible	(6.69e-3, 6.78e-3)	(2.60e-4,2.63e-4)	(25.4,26.1)	2	no
substitutes efficient code for array-based reccurences *	(5.60e-2, 5.62e-2)	(8.32e-2,8.49e-2)	(0.659,0.675)	0.5	no
substitutes efficient code for array sum computations *	(2.32e-3, 2.35e-3)	(5.63e-2,5.67e-2)	(0.0409,0.0416)	0.5	**yes**
substitutes efficient code for array product computations *	(2.32e-3, 2.35e-3)	(5.64e-2,5.67e-2)	(0.041,0.0416)	0.5	**yes**
scalar expansion in a single loop nest	(5.07e-3, 5.15e-3)	(5.11e-3,5.18e-3)	(0.978,1.01)	2	**yes**
scalar expansion in a double loop nest	(5.13e-3, 5.20e-3)	(5.09e-3,5.15e-3)	(0.995,1.02)	2	**yes**
array privatization, where the temporary array is defined in the same loop as it is used	(2.06e-2, 2.08e-2)	(5.09e-3,5.15e-3)	(4,4.09)	2	no
array privatization, where the temporary array is defined in a loop neighboring the loop where it is used	(2.27e-2, 2.30e-2)	(5.09e-3,5.15e-3)	(4.41,4.51)	2	no

Test	*Naive* Run Time 95% Conf. Int. (seconds)	*Opt* /*Cripple* Run Time 95% Conf. Int. (seconds)	R 90% Conf. Int. (seconds)	R_p	Applied?
parallelizes a dependency-free 1-d loop, where the source and sink elements are in seperate regions of the array	(2.02e-3, 2.06e-3)	(1.98e-3,2.01e-3)	(1.01,1.04)	2	**yes**
parallelizes a dependency-free 1-d loop, where the source and sink elements are interleaved	(2.35e-3, 2.38e-3)	(2.37e-3,2.40e-3)	(0.982,1.01)	2	**yes**
parallelizes a dependency-free 1-d loop, where the source and sink element indices are defined as linear functions of the loop variable	(1.92e-3, 1.95e-3)	(3.15e-3,3.18e-3)	(0.605,0.618)	2	**yes**
parallelizes a dependency-free 2-d loop, where the source and sink elements are defined as linear functions of the loop variable	(5.02e-3, 5.07e-3)	(2.38e-3,2.42e-3)	(2.07,2.13)	2	no
removes stores to temporary arrays	(1.11e-2, 1.12e-2)	(9.78e-3,9.92e-3)	(1.12,1.14)	1.1	no
parallelizes the loop assignment of a arithmetic function of the loop variable to an array *	(2.45e-4, 2.47e-4)	(3.07e-3,3.11e-3)	(0.0788,0.0805)	0.5	**yes**
parallelizes the loop assignment of a function of the loop variable to an array, where the function involves an intrinsic *	(2.70e-3, 2.73e-3)	(4.88e-3,4.95e-3)	(0.546,0.558)	0.5	no
parallelizes the loop assignment of a function of the loop variable to an array, where the function involves an integer leaf function *	(3.14e-2, 3.18e-2)	(3.17e-2,3.19e-2)	(0.986,1)	0.5	no
parallelizes a serially-expressed gather	(1.57e-4, 1.59e-4)	(1.60e-4,1.62e-4)	(0.969,0.995)	2	**yes**
parallelizes a serially-expressed scatter	(6.57e-4, 6.64e-4)	(6.54e-4,6.61e-4)	(0.994,1.02)	2	**yes**
parallelizes a serially-expressed computation involving two permutation arrays	(5.23e-3, 5.42e-3)	(5.10e-3,5.15e-3)	(1.02,1.06)	2	**yes**
parallelizes a serially-expressed bin summation *	(4.88e-3, 4.98e-3)	(8.07e-3,8.15e-3)	(0.599,0.617)	0.5	no
parallelizes a serially-expressed all up to element array sum *	(1.31e-3, 1.32e-3)	(1.74e-3,1.77e-3)	(0.74,0.759)	0.5	no
parallelizes a serially-expressed `where` operation	(3.90e-4, 4.77e-4)	(3.13e-4,3.20e-4)	(1.22,1.52)	2	**yes**

E.3 Architecture Characterizer Measurements

The following tables detail our architectural measurements.

E.3.1 CM-5/`cmf` Architecture Results

Full block model, full mask model, cache stride model.
Constant bound block references incur overhead.

Parameter	Mean Value (seconds)	90-Percent Confidence Interval (seconds)	Minimum Value (seconds)
REM-READ	9.9302e-06	(9.8300e-06,1.0030e-05)	9.7152e-06
REM-WRITE	2.9409e-06	(2.9218e-06,2.9600e-06)	2.9001e-06
SREM-READ	5.9581e-05	(5.9017e-05,6.0144e-05)	5.9099e-05
SREM-WRITE	3.4622e-05	(3.4415e-05,3.4829e-05)	3.4267e-05
OH-GATHER	5.4865e-09	(4.7767e-09,6.1964e-09)	5.0840e-09
OH-SCATTER	4.6402e-09	(4.6165e-09,4.6638e-09)	4.6131e-09
OH-CACHESTRIDE$_{2,65536}$	3.9308e-09	(3.8768e-09,3.9849e-09)	3.8483e-09
OH-CACHESTRIDE$_{8,65536}$	1.6277e-08	(1.5757e-08,1.6797e-08)	1.5367e-08
OH-CACHESTRIDE$_{32,65536}$	6.3213e-08	(6.2111e-08,6.4315e-08)	6.1327e-08
OH-CACHESTRIDE$_{128,65536}$	2.4903e-07	(2.4797e-07,2.5009e-07)	2.4572e-07
OH-CACHESTRIDE$_{512,65536}$	1.0066e-06	(9.9084e-07,1.0224e-06)	9.8100e-07
OH-CACHESTRIDE$_{2,262144}$	1.9598e-09	(1.9408e-09,1.9788e-09)	1.9334e-09
OH-CACHESTRIDE$_{8,262144}$	7.8168e-09	(7.7561e-09,7.8776e-09)	7.7202e-09
OH-CACHESTRIDE$_{32,262144}$	3.1672e-08	(3.1206e-08,3.2137e-08)	3.1043e-08
OH-CACHESTRIDE$_{128,262144}$	1.2772e-07	(1.2514e-07,1.3029e-07)	1.2354e-07
OH-CACHESTRIDE$_{512,262144}$	5.0551e-07	(4.9818e-07,5.1285e-07)	4.9344e-07
OH-CACHESTRIDE$_{2,1048576}$	1.6808e-09	(1.6615e-09,1.7001e-09)	1.6540e-09
OH-CACHESTRIDE$_{8,1048576}$	6.6532e-09	(6.6266e-09,6.6798e-09)	6.6229e-09
OH-CACHESTRIDE$_{32,1048576}$	2.6787e-08	(2.6531e-08,2.7044e-08)	2.6495e-08
OH-CACHESTRIDE$_{128,1048576}$	1.1761e-07	(1.0049e-07,1.3473e-07)	1.0593e-07
OH-CACHESTRIDE$_{512,1048576}$	4.3292e-07	(4.2519e-07,4.4065e-07)	4.2368e-07
OH-FULLMASK$_{0}$	4.1653e-10	(4.0040e-10,4.3267e-10)	4.1940e-10
OH-FULLMASK$_{25}$	4.1608e-10	(4.0039e-10,4.3177e-10)	4.1927e-10
OH-FULLMASK$_{50}$	4.1777e-10	(4.0155e-10,4.3400e-10)	4.1949e-10
OH-FULLMASK$_{75}$	4.3051e-10	(4.0595e-10,4.5508e-10)	4.1946e-10
OH-FULLMASK$_{100}$	4.1880e-10	(4.0302e-10,4.3458e-10)	4.1970e-10
OH-FULLBLOCK$_{0}$	1.1488e-09	(1.1382e-09,1.1594e-09)	1.1375e-09
OH-FULLBLOCK$_{12}$	1.1057e-09	(1.0847e-09,1.1266e-09)	1.0890e-09
OH-FULLBLOCK$_{25}$	1.0402e-09	(1.0317e-09,1.0487e-09)	1.0370e-09
OH-FULLBLOCK$_{50}$	9.3605e-10	(9.2399e-10,9.4810e-10)	9.3261e-10
OH-FULLBLOCK$_{100}$	7.2291e-10	(7.0728e-10,7.3854e-10)	7.2353e-10

E.3.2 Y-MP C90/`cf77` Architecture Results

Partial block model, full mask model, bank stride model.
Constant bound block references do not incur significant overhead.

Parameter	Mean Value (seconds)	90-Percent Confidence Interval (seconds)	Minimum Value (seconds)
OH-RAND,READ	4.7707e-09	(4.2404e-09,5.3010e-09)	2.7227e-09
OH-RAND,WRITE	1.7650e-08	(1.5816e-08,1.9484e-08)	1.4404e-08
OH-GATHER	1.9664e-10	(1.2042e-10,2.7286e-10)	1.3150e-10
OH-SCATTER	1.8316e-09	(1.7732e-09,1.8901e-09)	2.3071e-09
OH-BANKSTRIDE_{2}	3.0197e-10	(2.0457e-10,3.9936e-10)	1.8891e-10
OH-BANKSTRIDE_{4}	9.3616e-10	(8.2922e-10,1.0431e-09)	6.7888e-10
OH-BANKSTRIDE_{8}	3.0296e-09	(2.8303e-09,3.2290e-09)	1.9145e-09
$\text{OH-BANKSTRIDE}_{16}$	6.0458e-09	(5.8026e-09,6.2891e-09)	3.5653e-09
$\text{OH-BANKSTRIDE}_{32}$	1.3045e-08	(1.2432e-08,1.3658e-08)	7.1932e-09
$\text{OH-BANKSTRIDE}_{64}$	2.6222e-08	(2.5199e-08,2.7244e-08)	1.5421e-08
$\text{OH-BANKSTRIDE}_{128}$	2.7667e-08	(2.5988e-08,2.9346e-08)	1.8911e-08
$\text{OH-BANKSTRIDE}_{256}$	3.4986e-08	(3.3307e-08,3.6665e-08)	2.2652e-08
$\text{OH-BANKSTRIDE}_{512}$	5.0548e-08	(4.8871e-08,5.2225e-08)	3.9601e-08
OH-FULLMASK_{0}	4.6312e-10	(3.4547e-10,5.8078e-10)	2.4124e-10
OH-FULLMASK_{25}	5.3527e-09	(5.1345e-09,5.5708e-09)	3.7756e-09
OH-FULLMASK_{50}	5.5061e-09	(5.2534e-09,5.7589e-09)	3.7180e-09
OH-FULLMASK_{75}	5.5015e-09	(5.1860e-09,5.8170e-09)	3.4023e-09
OH-FULLMASK_{100}	5.5801e-09	(5.2850e-09,5.8752e-09)	3.5740e-09
$\text{OH-PARTIALBLOCK}_{16}$	8.9994e-08	(1.9464e-08,1.6052e-07)	2.9642e-08
$\text{OH-PARTIALBLOCK}_{256}$	7.7009e-09	(1.5670e-09,1.3835e-08)	3.8264e-09
$\text{OH-PARTIALBLOCK}_{4096}$	1.9227e-09	(1.7613e-09,2.0842e-09)	1.9079e-09
$\text{OH-PARTIALBLOCK}_{65536}$	1.4724e-10	(4.9131e-11,2.4535e-10)	6.8782e-11
$\text{OH-PARTIALBLOCK}_{262144}$	2.1587e-10	(1.0430e-10,3.2745e-10)	1.5179e-10
$\text{OH-PARTIALBLOCK}_{1048576}$	1.7512e-10	(6.4474e-11,2.8578e-10)	9.1809e-12

E.3.3 Y-MP C90/`f90` Architecture Results

Partial block model, full mask model, bank stride model.
Constant bound block references do not incur significant overhead.

Parameter	Mean Value (seconds)	90-Percent Confidence Interval (seconds)	Minimum Value (seconds)
OH-RAND,READ	1.0754e-08	(1.0377e-08,1.1131e-08)	1.0390e-08
OH-RAND,WRITE	2.0473e-08	(1.4788e-08,2.6159e-08)	1.4728e-08
OH-GATHER	7.2863e-10	(6.8146e-10,7.7579e-10)	6.8337e-10
OH-SCATTER	5.4722e-10	(5.0464e-10,5.8981e-10)	5.7459e-10
OH-BANKSTRIDE_{2}	1.1643e-10	(-5.3411e-10,7.6696e-10)	5.4003e-10
OH-BANKSTRIDE_{4}	1.2499e-09	(5.9931e-10,1.9005e-09)	1.5848e-09
OH-BANKSTRIDE_{8}	5.7511e-09	(5.0998e-09,6.4023e-09)	6.0240e-09
$\text{OH-BANKSTRIDE}_{16}$	1.2803e-08	(1.2145e-08,1.3460e-08)	1.2794e-08
$\text{OH-BANKSTRIDE}_{32}$	2.7267e-08	(2.6606e-08,2.7927e-08)	2.7266e-08
$\text{OH-BANKSTRIDE}_{64}$	5.7796e-08	(5.7108e-08,5.8484e-08)	5.7347e-08
$\text{OH-BANKSTRIDE}_{128}$	5.8336e-08	(5.7641e-08,5.9032e-08)	5.7826e-08
$\text{OH-BANKSTRIDE}_{256}$	5.9440e-08	(5.8667e-08,6.0213e-08)	5.7845e-08
$\text{OH-BANKSTRIDE}_{512}$	7.6560e-08	(6.0183e-08,9.2936e-08)	5.9344e-08
OH-FULLMASK_{0}	5.5128e-09	(5.0851e-09,5.9405e-09)	4.8904e-09
OH-FULLMASK_{25}	1.5474e-08	(1.4927e-08,1.6021e-08)	1.3169e-08
OH-FULLMASK_{50}	1.5954e-08	(1.5247e-08,1.6662e-08)	1.2945e-08
OH-FULLMASK_{75}	1.7447e-08	(1.6665e-08,1.8230e-08)	1.3211e-08
OH-FULLMASK_{100}	1.6165e-08	(1.5230e-08,1.7099e-08)	1.3147e-08
$\text{OH-PARTIALBLOCK}_{16}$	4.6877e-08	(2.0399e-08,7.3354e-08)	1.6987e-08
$\text{OH-PARTIALBLOCK}_{256}$	3.5644e-09	(-3.1366e-10,7.4424e-09)	5.5318e-10
$\text{OH-PARTIALBLOCK}_{4096}$	0.0000e+00	(0.0000e+00,0.0000e+00)	7.0828e-11
$\text{OH-PARTIALBLOCK}_{65536}$	0.0000e+00	(0.0000e+00,0.0000e+00)	2.8337e-11
$\text{OH-PARTIALBLOCK}_{262144}$	0.0000e+00	(0.0000e+00,0.0000e+00)	0.0000e+00
$\text{OH-PARTIALBLOCK}_{1048576}$	0.0000e+00	(0.0000e+00,0.0000e+00)	0.0000e+00

Appendix F. Optimization Descriptions

This section describes and gives examples of each of the basic types of optimizations which the Optimization Characterizer detects.

F.1 Code Motion

Segments of code inside a do-loop which calculate values which could be calculated outside the loop are relocated, eliminating redundant computations and exposing other optimizations.

Before	After	Notes

`do i=1,n`
`   a(i)=a(i)+1.0`
`   k=2.0`
`end do`
`a(0)=a(0)+k`
	`k=2.0`
	`do i=1,n`
	`   a(i)=a(i)+1.0`
	`end do`
	`a(0)=a(0)+k`
		none

F.2 Common Subexpression Elimination

As suggested by the name, redundant expressions which perform identical calculations on the same operands are removed and replaced with a reference to the initially calculated value.

Before	After	Notes
`a=a*b*b-f(b*b)`	`c=b*b` `a=a+c-f(c)`	none

F.3 Dead Code Elimination

Occasionally, the application of optimizations (or programmer oversight) creates segments of code which have no semantic effect on the outcome of the program; such code is eliminated.

Before	After	Notes

`do i=1,n`
`   a(i)=a(i)+1.0`
`   b(i)=dead()`
`end do`
	`do i=1,n`
	`   a(i)=a(i)+1.0`
	`end do`
		`b` is not used anywhere in the program, `dead` has no side effects.

F.4 Forward Substitution

At compile-time, any expressions which can be determined to have been assigned to variables in the computation are substituted for the variable references, in the hope that other optimizations will be exposed.

Before	After	Notes

`j=n/2`
`do i=1,n/2`
`   a(i)=a(i+j)+1.0`
`end do`
	`do i=1,n/2`
	`   a(i)=a(i+n/2)+1.0`
	`end do`
		none

F.5 Induction Variable Elimination

do-loops often contain induction variables, which are incremented by a constant factor each iteration of the loop and generally used to serially reference arrays at some given stride. Induction variables produce inter-iteration data dependencies, and can be eliminated by replacing the references to the induction variable by a linear function of the loop variable.

Before	After	Notes

`j=0`
`do i=1,n`
`   j=j+1`
`   a(j)=a(i)*b(i)`
`end do`
	`do i=1,n`
	`   a(i)=a(i)*b(i)`
	`end do`
		none

F.6 Loop Collapsing

Loop nests are condensed where possible, such that two or more nested loops become a single loop; the dimensions of any arrays referenced by such loops are condensed to one dimension. Loop collapsing reduces loop overhead, and sometimes allows more effective vectorization.

Before	After	Notes

`do i1=1,n1`
`   do i2=1,n2`
`      a(i1,i2)=a(i1,i2)*b(i1,i2)`
`   end do`
`end do`
	`do i=1,n1*n2`
	`   acoll(i)=acoll(i)*bcoll(i)`
	`end do`
		`acoll` and `bcoll` correspond to collapsed versions of `a` and `b`, respectively.

F.7 Recurrence Substitution

Solutions or efficient solvers are substituted for recurrences.

Before	After	Notes

`do i=1,n`
`   a=a*b`
`end do`
	`a=a*b**n`	none

F.8 Reduction Substitution

Reductions, such as the sum or product of the elements in a vector, are replaced by finely-tuned code or subroutines.

Before	After	Notes

`sum=0.0`
`do i=1,n`
`   sum=sum+a(i)`
`end do`
	`sum=sumvector(a)`	`sumvector` is an efficient vector sum implementation.

F.9 Idiom Recognition

Operations expressed in a serial manner which typically have special compiler or hardware support, such as gather/scatter or masked operations, are recognized and parallelized/vectorized.

Before	After	Notes

`do i=1,n`
`   if (a(i).gt.0.0) b(i)=a(i)`
`end do`
	`where (a.gt.0.0) b=a`	none

F.10 Scalar Expansion

Programmers often use scalar temporaries inside of `do`-loops to store intermediate values; unfortunately, these temporaries create output dependences. Each temporary is expanded into a vector, each element of which holds the value of the temporary for a certain iteration of the loop, eliminating the write after write dependence and resulting in code which is more likely amenable to parallelization.

Before	After	Notes

`do i=1,n`
`   x=c(i)*b(i)`
`   a(i)=a(i)+x`
`end do`
	`do i=1,n`
	`   x(i)=c(i)*b(i)`
	`   a(i)=a(i)+x(i)`
	`end do`
		none

F.11 Semantic Analysis

The compiler analyzes a computation to determine which values of involved variables will allow optimization, and it produces multiple compiled versions of the code to be executed conditionally at run time. Of course, the compiler does not check all the possible combinations of each possible value of each variable involved in the computation; usually, it uses a pattern matching algorithm to derive regions of variable values which allow other optimizations to be applied.

One version of the compiled code executes serially, and is run when values of involved variables induce data dependencies which inhibit optimizations, and the others are efficient versions in which the values of involved variables allow various optimizations.

Before	After	Notes

`do i=1,n-j`
`   a(i)=a(i+j)*b(i)`
`end do`
	`if (j.ge.0) then`
	`   do i=1,n-j`
	`      a(i)=a(i+j)*b(i)`
	`   end do`
	`else`
	`   do i=1,n-j`
	`      a(i)=a(i+j)*b(i)`
	`   end do`
	`endif`
		The compiler will parallelize the first loop in the After section, since the values which `j` can have in the first loop will not cause optimization-inhibiting data dependencies, but not the second

F.12 Subroutine Inlining

The contents of leaf functions are expanded inline, eliminating subroutine call overhead and exposing other potential optimizations. Inline expansion may proceed many levels deep, or may be limited to small functions or those returning constants.

Before	After	Notes

`real function leaf(x)`
`   leaf=x**3`
`end`
 	
`[...]`
 	
`do i=1,n`
`   a(i)=leaf(a(i))`
`end do`
	`do i=1,n`
	`   a(i)=a(i)**3`
	`end do`
		none

F.13 Data Dependency Analysis

The compiler attempts to recognize loops which are free of true *data dependencies*; such loops can be trivially parallelized/vectorized. For example, simple arithmetic tests exist to determine the absence of data dependencies in loops in which an elements of an array are read and written using indices calculated as linear functions of the loop variable. [17]

Before	After	Notes

`do i=1,n/3`
`   a(3*i)=a(2*i+3)+b(i)`
`end do`
	`a(3:n:3)=a(5:n/3*2+3:2)+`
	`   b(1:n/3)`
		The compiler determines that the original loop contains no data dependencies.

Appendix G. Kernel Run Time Prediction Calculations

The following sections detail each kernel, as well as the equations used to predict their run times. For most of our test kernels, serial computation composes a negligible portion of the run time, and we omit corresponding terms from our calculations except where noted.

We predict that a compiler optimizes a given code segment if the results of our compiler optimizations indicated that it parallelized the most similar segment in our test suite. We optimistically assume that the compiler applies optimizations for which we could not reach a definite conclusion.

Our measurements indicate that constant bound blocks do not incur significant overhead on the Cray YMP-C90, so block overhead is omitted in appropriate calculations.

G.1 `ave` Kernel Predictions

```
01:      before=0.
02:      do i = 1,n
03:         before=before+a(i)
04:      end do
05:
06:      do i = 1,1000
07:          a1=a(1)
08:          a2=a(2)
09:          an1=a(n-1)
10:          an=a(n)
11:          a(2:n-1)=0.25*a(1:n-2)+0.5*a(2:n-1)+0.25*a(3:n)
12:          a(1)=0.25*a2+0.5*a1+0.25*an
13:          a(n)=0.25*a1+0.5*an+0.25*an1
14:      end do
15:
16:      after=0.
17:      do i = 1,n
18:         after=after+a(i)
19:      end do
```

Figure 13: Core of the `ave` Kernel

`ave` is a Fortran 90 kernel which performs a computation consisting of three steps:

1. Summation the elements in the array using an accumulator inside a `do`-loop.
2. 1000 applications of an averaging function, which is essentially Jacobi relaxation in one dimension, to the array, expressed using array notation .
3. Summation the elements in the array using an accumulator inside a `do`-loop.

We pick an array size of 500000 elements. Figure 13 shows the body of the `ave` kernel.

Step 2 in the `ave` computation is trivially parallelizable; the other steps, however, are expressed sequentially, and the compiler must recognize them and parallelize them. Steps 1 and 3 are parallelized by all compiler/architecture pairs with the exception of CM-5/`cmf`.

The following paragraphs detail our run time prediction calculations. SOR-SUMS corresponds to the parallelized sum computations in lines 1 through 4 and 16 through 19.

$$\text{SOR-SUMS} = 2 \cdot \text{RS-SUM}_{500000} + 2 \cdot \text{RS-OHS}_{500000} + 2 \cdot \text{RS-LOAD}_{500000}$$

SOR-MAIN-FULL corresponds to the run time of line 11, in the case that we represent overhead due to block references by the full model.

$$\text{OH-FULLBLOCKTOTAL} = 500000 \cdot \text{OH-FULLBLOCK}_{100}$$

$$\text{SOR-MAIN-FULL} = 3000 \cdot \text{RS-MUL}_{500000} + 2000 \cdot \text{RS-ADD}_{500000} + 3000 \cdot \text{RS-LOAD}_{500000} + 1000 \cdot \text{RS-OHS}_{500000} + 3000 \cdot \text{OH-FULLBLOCKTOTAL}$$

SOR-MAIN-NONE corresponds to the run time of line 11, in the case that no block overhead is accrued.

$$\text{SOR-MAIN-NONE} = 3000 \cdot \text{RS-MUL}_{500000} + 2000 \cdot \text{RS-ADD}_{500000} + 3000 \cdot \text{RS-LOAD}_{500000} + 1000 \cdot \text{RS-OHS}_{500000}$$

SOR-MAIN-REM1 and SOR-MAIN-REM2 correspond to the cost remote references generated in lines 7 through 10 and 12 though 13, respectively.

$$\text{SOR-MAIN-REM1} = 4000 \cdot \text{SREM-READ} + 2000 \cdot \text{SREM-WRITE}$$

$$\text{SOR-MAIN-REM2} = 2000 \cdot \text{REM-READ}$$

SOR-SUMS-REM corresponds to the cost of sequential remote reads generated by the unparallelized sum computations in lines 1 through 4 and 16 through 19.

$$\text{SOR-SUMS-REM} = 1\text{e+}06 \cdot \text{SREM-READ}$$

SOR-CRAY corresponds to the total run time on the Cray architecture, SOR-CMF corresponds to the run time on the CM-5/`cmf` pair (sans contributions due to serial non-access computations, which are negligible), and SOR-CMAX corresponds to the run time on the CM-5/`cmf-cmax` pair.

$$\text{SOR-CRAY} = \text{SOR-SUMS} + \text{SOR-MAIN-NONE} + \text{SOR-MAIN-REM1} + \text{SOR-MAIN-REM2}$$

$$\text{SOR-CMF} = \text{SOR-SUMS-REM} + \text{SOR-MAIN-FULL} + \text{SOR-MAIN-REM1} + \text{SOR-MAIN-REM2}$$

$$\text{SOR-CMAX} = \text{SOR-SUMS} + \text{SOR-MAIN-FULL} + \text{SOR-MAIN-REM1} + \text{SOR-MAIN-REM2}$$

G.2 `merge` Kernel Predictions

```
01:       real a(0:nn-1)
02:       integer off,off2
03:       integer index(0:nn-1)
04:       integer numbers(0:nn-1)
05:       logical mask(0:nn-1)
06:
07:       do i=0,nn-1
08:          numbers(i)=i
09:       end do
10:
11:       do oi=1,n
12:
13:          do i=oi,1,-1
14:
15:             off=2**i
16:             off2=off/2
17:             if (i.eq.oi.and.i.ne.1) then
18:                index=ishft(ishft(numbers,-i),i)+off-1-
19:      +            iand(numbers,off-1)
20:             else
21:                index=ieor(numbers,off2)
22:             endif
23:             mask=.not.(numbers.gt.index.xor.a(index).gt.a)
24:             where (mask)
25:                a=a(index)
26:             end where
27:
28:          end do
29:       end do
```

Figure 14: Core of the `merge` Kernel

`merge` is a Fortran 90 implementation of Batcher's odd-even merge sort, which sorts n numbers in $O(n \log^2 n)$ time. The sort can operate on data in place, is relatively efficient for medium sized arrays, and is not stable.[18] The main loop of our implementation, shown in Figure 14, sorts a four arrays of `nn` quasi-random reals into ascending order.

In our specific benchmark, `nn` is 32768, and `n` is 15. Thus, the code inside the two `do`-loops is executed a total of 120 iterations per array sorted. We assume that on each iteration, an average of half of the elements in the array being sorted are swapped.

MERGE-1-4 and MERGE-1-128 correspond to the total run time of line 23 on a 4-processor shared memory architecture and an 128-processor distributed memory architecture with blocked data layout, respectively.

$$\text{OH-RAND-READ-NORM} = 0.25 \cdot \text{OH-RAND-READ}$$

$$\text{MERGE-1-4} = 120 \cdot \text{L-OHS}_{32768} + 120 \cdot \text{L-NOT}_{32768} + 120 \cdot \text{I-LOAD}_{32768} + 120 \cdot \text{I-GT}_{32768} + 120 \cdot \text{I-LOAD}_{32768} + 120 \cdot \text{L-OR}_{32768} + 120 \cdot \text{RS-LOAD}_{32768} + 120 \cdot \text{RS-GT}_{32768} + 3.93216\text{e+06} \cdot \text{OH-GATHER} + 3.93216\text{e+06} \cdot \text{OH-RAND-READ-NORM} + 120 \cdot \text{RS-LOAD}_{32768}$$

[18] A *stable* sort retains the original ordering of records with the same key.

$\text{MERGE-1-128} = 120 \cdot \text{L-OHS}_{32768} + 120 \cdot \text{L-NOT}_{32768} + 120 \cdot \text{I-LOAD}_{32768} + 120 \cdot \text{I-GT}_{32768} + 120 \cdot \text{I-LOAD}_{32768} + 120 \cdot \text{L-OR}_{32768} + 120 \cdot \text{RS-LOAD}_{32768} + 120 \cdot \text{RS-GT}_{32768} + 3.93216\text{e+}06 \cdot \text{OH-GATHER} + 917504 \cdot \text{REM-READ-NORM} + 92 \cdot \text{RS-LOAD}_{32768}$

MERGE-2 corresponds to the total run time of line 21.

$\text{MERGE-2} = 106 \cdot \text{I-OHS}_{32768} + 106 \cdot \text{I-LOAD}_{32768} + 106 \cdot \text{I-EOR}_{32768}$

MERGE-3 corresponds to the total run time of lines 18 though 19.

$\text{MERGE-3} = 14 \cdot \text{I-OHS}_{32768} + 28 \cdot \text{I-SHFT}_{32768} + 14 \cdot \text{I-LOAD}_{32768} + 14 \cdot \text{I-ADD}_{32768} + 14 \cdot \text{I-AND}_{32768} + 14 \cdot \text{I-LOAD}_{32768}$

MERGE-4-REM corresponds to the total run time of lines 7 though 9, unparallelized.

$\text{MERGE-4-REM} = 32768 \cdot \text{SREM-WRITE}$

MERGE-4-PAR corresponds to the run time of lines 7 though 9, parallelized on 4 processors.

$\text{MERGE-4-PAR} = 8192 \cdot \text{SREM-WRITE}$

MERGE-5-FULL-4 and MERGE-5-FULL-128 correspond to the total run time of the lines 24 though 26, in the case that the `where` statement is executed in a manner corresponding to the *full* mask model, for a 4-processor shared memory machine and an 128-processor distributed memory architecture with blocked data layout, respectively.

$\text{MERGE-5-FULL-4} = 3.93216\text{e+}06 \cdot \text{OH-FULLMASK}_{50} + 120 \cdot \text{RS-OHS}_{32768} + 3.93216\text{e+}06 \cdot \text{OH-GATHER} + 3.93216\text{e+}06 \cdot \text{RAND-READ}$

$\text{MERGE-5-FULL-128} = 3.93216\text{e+}06 \cdot \text{OH-FULLMASK}_{50} + 120 \cdot \text{RS-OHS}_{32768} + 92 \cdot \text{RS-LOAD}_{32768} + 3.93216\text{e+}06 \cdot \text{OH-GATHER} + 917504 \cdot \text{REM-READ-NORM}$

The MERGE-CMF, MERGE-CMAX, MERGE-CF77, and MERGE-F90 parameters correspond to the total run time on the CM-5/`cmf`, CM-5/`cmf-cmax`, Y-MP C90/`cf77`, and Y-MP C90/`f90`pairs, respectively.

$\text{MERGE-CMF} = 4 \cdot \text{MERGE-1-128} + 4 \cdot \text{MERGE-2} + 4 \cdot \text{MERGE-3} + 4 \cdot \text{MERGE-4-REM} + 4 \cdot \text{MERGE-5-FULL-128}$

$\text{MERGE-CMAX} = 4 \cdot \text{MERGE-1-128} + 4 \cdot \text{MERGE-2} + 4 \cdot \text{MERGE-3} + 4 \cdot \text{MERGE-5-FULL-128}$

$\text{MERGE-CF77} = 4 \cdot \text{MERGE-1-4} + 4 \cdot \text{MERGE-2} + 4 \cdot \text{MERGE-3} + 4 \cdot \text{MERGE-4-PAR} + 4 \cdot \text{MERGE-5-FULL-4}$

$\text{MERGE-F90} = 4 \cdot \text{MERGE-1-4} + 4 \cdot \text{MERGE-2} + 4 \cdot \text{MERGE-3} + 4 \cdot \text{MERGE-4-REM} + 4 \cdot \text{MERGE-5-FULL-4}$

G.3 `ep` Kernel Predictions

```
01:   do j = 1 ,2**(M-N)
02:
03:       call ARANLC( A ,y ,x, nn )
04:       call ARANLC( A ,x ,y, nn )
05:
06:       xk = x * R45 - 1.0
07:       yk = y * R45 - 1.0
08:       t = xk**2 + yk**2
09:
10:       select = ( t .LE. 1.0 )
11:       where( select )
12:           t = SQRT( -2 * LOG( t ) / t )
13:           xk = xk * t
14:           yk = yk * t
15:           sumx = sumx + xk
16:           sumy = sumy + yk
17:           ic = INT( MAX( ABS(xk) ,ABS(yk) ) )
18:       endwhere
19:
20:       do i = 0 , NQ-1
21:           where( select .AND. ic .EQ. i )
22:  +            counts(i,:) = counts(i,:) + 1
23:       enddo
24:
25:   enddo
26:
27:   sumx1 = SUM( sumx )
28:   sumy1 = SUM( sumy )
29:   do i=0,nq-1
30:      countslice=counts(i,:)
31:      counts1(i) = SUM(countslice)
32:   end do

  [...]

33:   subroutine ARANLC( A ,Xk ,Xkp1, n )
34:
35:   real A
36:   real Xk(n)
37:   real Xkp1(n)
38:   real t1(n),t2(n),x1(n),x2(n) ,z(n)
39:
40:   a1 = AINT( R23 * A )
41:   a2 = A - T23 * a1
42:   x1 = AINT( R23 * Xk )
43:   x2 = Xk - T23 * x1
44:
45:   t1 = a1 * x2 + a2 * x1
46:   z = t1 - T23 * AINT( R23 * t1 )
47:   t2 = T23 * z + a2 * x2
48:   Xkp1 = t2 - T46 * AINT( R46 * t2 )
49:
50:   return
51:   end
```

Figure 15: Core of the `ep` Kernel

`ep` is the sample version of the Embarrassingly Parallel kernel of the NAS Parallel Benchmark suite, modified so it compiles on all systems in question. Using a parallelizable random number generator, the kernel generates a large number (2^{21}) of random two dimensional coordinates in a Gaussian distribution about the origin, and tallies the number of coordinates which fall into each of a set of concentric square regions also centered on the origin. The random number generator stores and manipulates 48-bit integer quantities using the mantissa of `REAL*8` variables.

We measure the parallel random number and coordinate generation, region testing, and result calculation portions of the kernel. We generate 2097512 (2^{21}) coordinates, in batches of 8196 (2^{13}), and calculate the number of coordinates falling into each of 10 annuli.

The following calculations compose our predictions:
EP-ARANLC-RS-1 is the run time of a single call to the `aranlc` subroutine as in line 3.

$$\text{EP-ARANLC-RS-1} = 256 \cdot \text{RS-OHS}_{8192} + 256 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-I}_{8192} + 256 \cdot \text{I-RS}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-ADD}_{8192} + 256 \cdot \text{RS-MUL}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 256 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-I}_{8192} + 256 \cdot \text{I-RS}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-ADD}_{8192} + 256 \cdot \text{RS-MUL}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 1024 \cdot \text{RS-LOAD}_{8192} + 512 \cdot \text{RS-MUL}_{8192} + 256 \cdot \text{RS-ADD}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 512 \cdot \text{RS-MUL}_{8192} + 256 \cdot \text{RS-ADD}_{8192} +$$

$256 \cdot \text{RS-I}_{8192} + 256 \cdot \text{I-RS}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 512 \cdot \text{RS-MUL}_{8192} + 256 \cdot \text{RS-ADD}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 512 \cdot \text{RS-MUL}_{8192} + 256 \cdot \text{RS-ADD}_{8192} + 256 \cdot \text{RS-I}_{8192} + 256 \cdot \text{I-RS}_{8192}$

EP-RS-1 is the total time spent in the `aranlc` subroutine.

$$\text{EP-RS-1} = 2 \cdot \text{EP-ARANLC-RS-1}$$

EP-RS-2 is the run time of lines 6 through 8.

$$\text{EP-RS-2} = 256 \cdot \text{RS-OHS}_{8192} + 256 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-MUL}_{8192} + 256 \cdot \text{RS-ADD}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 256 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-MUL}_{8192} + 256 \cdot \text{RS-ADD}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 512 \cdot \text{RS-EXPI}_{8192} + 256 \cdot \text{RS-ADD}_{8192}$$

EP-RS-3 is the run time of line 10.

$$\text{EP-RS-3} = 256 \cdot \text{L-OHS}_{8192} + 256 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-GTCONST}_{8192}$$

$$\text{OH-FULLMASK}_{79} = 0.84 \cdot \text{OH-FULLMASK}_{75} + 0.16 \cdot \text{OH-FULLMASK}_{100}$$

EP-RS-4 is the run time of lines 11 through 18.

$$\text{EP-RS-4} = 256 \cdot \text{L-OHS}_{8192} + 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{79} + 256 \cdot \text{RS-OHS}_{8192} + 256 \cdot \text{RS-SQRT}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-LOG}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-MUL}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-MUL}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-ADD}_{8192} + 256 \cdot \text{RS-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-ADD}_{8192} + 256 \cdot \text{I-OHS}_{8192} + 512 \cdot \text{RS-LOAD}_{8192} + 256 \cdot \text{RS-MAX}_{8192} + 256 \cdot \text{RS-ABS}_{8192} + 256 \cdot \text{RS-I}_{8192}$$

$$\text{OH-FULLMASK}_{37} = 0.52 \cdot \text{OH-FULLMASK}_{25} + 0.48 \cdot \text{OH-FULLMASK}_{50}$$

$$\text{OH-FULLMASK}_{34} = 0.64 \cdot \text{OH-FULLMASK}_{25} + 0.36 \cdot \text{OH-FULLMASK}_{50}$$

$$\text{OH-FULLMASK}_{06} = 0.76 \cdot \text{OH-FULLMASK}_{0} + 0.24 \cdot \text{OH-FULLMASK}_{25}$$

EP-RS-5 is the run time of lines 20 through 23.

$$\text{EP-RS-5} = 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{37} + 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{34} + 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{06} + 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+06} \cdot \text{OH-FULLMASK}_{0} + 2560 \cdot \text{L-LOAD}_{8192} + 2560 \cdot \text{L-AND}_{8192} + 5120 \cdot \text{I-LOAD}_{8192} + 256 \cdot \text{I-GT}_{8192} + 2560 \cdot \text{I-OHS}_{8192} + 256 \cdot \text{I-LOAD}_{8192} + 256 \cdot \text{I-ADD}_{8192}$$

EP-RS-6 is the run time of lines 29 through 32.

$$\text{EP-RS-6} = \text{I-SUM}_{8192} + \text{I-LOAD}_{8192} + \text{I-SUM}_{8192} + \text{I-LOAD}_{8192} + 10 \cdot \text{I-OHS}_{8192} + 10 \cdot \text{I-LOAD}_{8192} + 10 \cdot \text{I-SUM}_{8192} + 10 \cdot \text{I-LOAD}_{8192}$$

EP-RS is the total run time of the kernel, using single precision reals as storage for integers.

$$\text{EP-RS} = \text{EP-RS-1} + \text{EP-RS-2} + \text{EP-RS-3} + \text{EP-RS-4} + \text{EP-RS-5} + \text{EP-RS-6}$$

The following calculations compute EP-RD, the total run time of the kernel, using double precision reals as storage for integers. Sans changes from single to double precision, the calculations are the same as those used to calculate EP-RS.

$\text{EP-ARANLC-RD-1} = 256 \cdot \text{RD-OHS}_{8192} + 256 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-I}_{8192} + 256 \cdot \text{I-RD}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-ADD}_{8192} + 256 \cdot \text{RD-MUL}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 256 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-I}_{8192} + 256 \cdot \text{I-RD}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-ADD}_{8192} + 256 \cdot \text{RD-MUL}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 1024 \cdot \text{RD-LOAD}_{8192} + 512 \cdot \text{RD-MUL}_{8192} + 256 \cdot \text{RD-ADD}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 512 \cdot \text{RD-MUL}_{8192} + 256 \cdot \text{RD-ADD}_{8192} + 256 \cdot \text{RD-I}_{8192} + 256 \cdot \text{I-RD}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 512 \cdot \text{RD-MUL}_{8192} + 256 \cdot \text{RD-ADD}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 512 \cdot \text{RD-MUL}_{8192} + 256 \cdot \text{RD-ADD}_{8192} + 256 \cdot \text{RD-I}_{8192} + 256 \cdot \text{I-RD}_{8192}$

$\text{EP-RD-1} = 2 \cdot \text{EP-ARANLC-RD-1}$

$\text{EP-RD-2} = 256 \cdot \text{RD-OHS}_{8192} + 256 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-MUL}_{8192} + 256 \cdot \text{RD-ADD}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 256 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-MUL}_{8192} + 256 \cdot \text{RD-ADD}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 512 \cdot \text{RD-EXPI}_{8192} + 256 \cdot \text{RD-ADD}_{8192}$

$\text{EP-RD-3} = 256 \cdot \text{L-OHS}_{8192} + 256 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-GTCONST}_{8192}$

$\text{EP-RD-4} = 256 \cdot \text{L-OHS}_{8192} + 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{79} + 256 \cdot \text{RD-OHS}_{8192} + 256 \cdot \text{RD-SQRT}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-LOG}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-MUL}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-MUL}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-ADD}_{8192} + 256 \cdot \text{RD-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-ADD}_{8192} + 256 \cdot \text{I-OHS}_{8192} + 512 \cdot \text{RD-LOAD}_{8192} + 256 \cdot \text{RD-MAX}_{8192} + 512 \cdot \text{RD-ABS}_{8192} + 256 \cdot \text{RD-I}_{8192}$

$\text{EP-RD-5} = 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{37} + 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{34} + 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{06} + 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{0} + 2.09715\text{e+}06 \cdot \text{OH-FULLMASK}_{0} + 2560 \cdot \text{L-LOAD}_{8192} + 2560 \cdot \text{L-AND}_{8192} + 5120 \cdot \text{I-LOAD}_{8192} + 256 \cdot \text{I-GT}_{8192} + 2560 \cdot \text{I-OHS}_{8192} + 256 \cdot \text{I-LOAD}_{8192} + 256 \cdot \text{I-ADD}_{8192}$

$\text{EP-RD-6} = \text{I-SUM}_{8192} + \text{I-LOAD}_{8192} + \text{I-SUM}_{8192} + \text{I-LOAD}_{8192} + 10 \cdot \text{I-OHS}_{8192} + 10 \cdot \text{I-LOAD}_{8192} + 10 \cdot \text{I-SUM}_{8192} + 10 \cdot \text{I-LOAD}_{8192}$

$\text{EP-RD} = \text{EP-RD-1} + \text{EP-RD-2} + \text{EP-RD-3} + \text{EP-RD-4} + \text{EP-RD-5} + \text{EP-RD-6}$

G.4 `multi` Kernel Predictions

```
do while (step.ge.1)

   ni=32

   do i=1,ni

      ax=ifirststep
      ay=ax
      bx=n+ifirststep-1
      by=bx

      sss(ax:bx:step,ay:by:step)=
         0.2*sss(ax:bx:step,ay:by:step)+
         0.2*sss(ax+step:bx+step:step,ay:by:step)+
         0.2*sss(ax-step:bx-step:step,ay:by:step)+
         0.2*sss(ax:bx:step,ay-step:by-step:step)+
         0.2*sss(ax:bx:step,ay+step:by+step:step)

   end do

   if (step.ge.2) then

      sss(ax+step/2:bx+step/2:step,ay+step/2:by+step/2:step)=
         0.25*sss(ax:bx:step,ay:by:step)+
         0.25*sss(ax+step:bx+step:step,ay:by:step)+
         0.25*sss(ax:bx:step,ay+step:by+step:step)+
         0.25*sss(ax+step:bx+step:step,ay+step:by+step:step)
      sss(ax:bx:step,ay+step/2:by+step/2:step)=
         0.25*sss(ax:bx:step,ay:by:step)+
         0.25*sss(ax:bx:step,ay+step:by+step:step)+
         0.25*sss(ax-step/2:bx-step/2:step,ay+step/2:
            by+step/2:step)+
         0.25*sss(ax+step/2:bx+step/2:step,ay+step/2:
            by+step/2:step)

      sss(ax+step/2:bx+step/2:step,ay:by:step)=
         0.25*sss(ax:bx:step,ay:by:step)+
            0.25*sss(ax+step:bx+step:step,ay:by:step)+
            0.25*sss(ax+step/2:bx+step/2:step,ay-step/2:
               by-step/2:step)+
            0.25*sss(ax+step/2:bx+step/2:step,ay+step/2:
               by+step/2:step)

   end if

   step=step/2
   ni=ni/2

end do
```

Figure 16: Core of the `multi` Kernel

`multi` solves for the potentials induced in a two-dimensional cavity with specified boundary conditions; more specifically, we compute the solution to Laplace's Equation over a 1024 by 1024 element grid, for 5 random sets of boundary conditions. `multi` uses a *multigrid* version of Jacobi relaxation to speed convergence; in other words, we utilize grids of increasing resolution as the simulation requires. When we have calculated a solution for an n by n grid, we simply interpolate these values into a $2n$ by $2n$ grid, calculate the solution for the higher density grid, and repeat the forementioned steps until we reach the appropriate grid density. In the case of the `multi` kernel, we start by solving for a grid size of 16 by 16, and we increase the dimensions by a factor of two until we reach 1024 by 1024. No convergence criteria are examined, and we iterate 32 steps at each grid resolution.

To avoid copy steps when we increase the grid density, we simply map our coarse grid to a 1024 by 1024 array, where the points in an n by n grid are separated by $1024/n$ elements in the 1024 by 1024 array. To simplify the handling of boundary conditions, we pad our 1024 by 1024 array with 64 elements on each side. These pad elements are set to the appropriate boundary condition values.

$$\text{MULTI-1} = \text{RS-STORE}_{1327104} + 4 \cdot \text{RS-STORE}_{73428}$$

MULTI-2-SBANK$_n$ corresponds to the run time of a set of 32 Jacobi iterations where the stride is n.

$$\text{MULTI-2-SBANK}_n = \text{RS-OHS}_{(1024/n)^2} + 4 \cdot \text{RS-ADD}_{(1024/n)^2} + 5 \cdot \text{RS-MUL}_{(1024/n)^2} + 5 \cdot \text{RS-LOAD}_{(1024/n)^2} + 6(1024/n)^2 \cdot \text{OH-BANKSTRIDE}_\text{n} + 5(1024/n)^2 \cdot \text{OH-PARTIALBLOCK}_{(1024/n)^2}$$

$$\text{MULTI-2-SBANK}_1 = \text{RS-OHS}_{1048576} + 4 \cdot \text{RS-ADD}_{1048576} + 5 \cdot \text{RS-MUL}_{1048576} + 5 \cdot \text{RS-LOAD}_{1048576} + 5.24288\text{e+06} \cdot \text{OH-PARTIALBLOCK}_{1048576}$$

$$\text{MULTI-2-BANK} = 32 \cdot \text{MULTI-2-SBANK}_{64} + 32 \cdot \text{MULTI-2-SBANK}_{32} + 32 \cdot \text{MULTI-2-SBANK}_{16} + 32 \cdot \text{MULTI-2-SBANK}_{8} + 32 \cdot \text{MULTI-2-SBANK}_{4} + 32 \cdot \text{MULTI-2-SBANK}_{2} + 32 \cdot \text{MULTI-2-SBANK}_{1}$$

$\text{MULTI-3-SBANK}_n = 3 \cdot \text{RS-OHS}_{(1024/n)^2} + 12 \cdot \text{RS-MUL}_{(1024/n)^2} + 9 \cdot \text{RS-ADD}_{(1024/n)^2} + 12 \cdot \text{RS-LOAD}_{(1024/n)^2} + 15(1024/n)^2 \cdot \text{OH-BANKSTRIDE}_n + 15(1024/n)^2 \cdot \text{OH-PARTIALBLOCK}_{(1024/n)^2}$

$\text{MULTI-3-BANK} = \text{MULTI-3-SBANK}_{64} + \text{MULTI-3-SBANK}_{32} + \text{MULTI-3-SBANK}_{16} + \text{MULTI-3-SBANK}_8 + \text{MULTI-3-SBANK}_4 + \text{MULTI-3-SBANK}_2$

$\text{MULTI-BANK} = 5 \cdot \text{MULTI-1} + 5 \cdot \text{MULTI-2-BANK} + 5 \cdot \text{MULTI-3-BANK}$

The following are predictions which cover the CM-5/`cmf` and CM-5/`cmf-cmax`pairs:

$\text{OH-CACHESTRIDETIME} = ++ \cdot \text{OH-CACHESTRIDE}\ldots$

$\text{OH-FULLBLOCKTIME} = +\text{frac1} \cdot \text{OH-FULLBLOCK}_{\text{size1}} + \text{frac2} \cdot \text{OH-FULLBLOCK}_{\text{size2}}$

$\text{MULTI-2-SCACHE}_n = \text{RS-OHS-256}_{(1024/n)^2} + 4 \cdot \text{RS-ADD}_{(1024/n)^2} + 5 \cdot \text{RS-MUL}_{(1024/n)^2} + 5 \cdot \text{RS-LOAD}_{(1024/n)^2} + 6 \cdot \text{OH-CACHESTRIDETIME}_{1024/n} + 5 \cdot \text{OH-FULLBLOCKTIME}_{1024/n}$

$\text{MULTI-2-SCACHE}_1 = \text{RS-OHS}_{1048576} + 4 \cdot \text{RS-ADD}_{1048576} + 5 \cdot \text{RS-MUL}_{1048576} + 5 \cdot \text{RS-LOAD}_{1048576} + 5 \cdot \text{OH-FULLBLOCKTIME}_1$

$\text{MULTI-2-CACHE} = 32 \cdot \text{MULTI-2-SCACHE64}_{+} 32 \cdot \text{MULTI-2-SCACHE}_{32} + 16 \cdot \text{MULTI-2-SCACHE}_{32} + 8 \cdot \text{MULTI-2-SCACHE}_{32} + 4 \cdot \text{MULTI-2-SCACHE}_{32} + 2 \cdot \text{MULTI-2-SCACHE}_{32} + 1 \cdot \text{MULTI-2-SCACHE}_{32}$

$\text{MULTI-3-SCACHE}_n = 3 \cdot \text{RS-OHS}_{(1024/n)^2} + 12 \cdot \text{RS-MUL}_{(1024/n)^2} + 9 \cdot \text{RS-ADD}_{(1024/n)^2} + 12 \cdot \text{RS-LOAD}_{(1024/n)^2} + 15 \cdot \text{OH-CACHESTRIDETIME}_{1024/n} + 15 \cdot \text{OH-FULLBLOCKTIME}_{1024/n}$

$\text{MULTI-3-CACHE} = \text{MULTI-3-SCACHE}_{64} + \text{MULTI-3-SCACHE}_{32} + \text{MULTI-3-SCACHE}_{16} + \text{MULTI-3-SCACHE}_8 + \text{MULTI-3-SCACHE}_4 + \text{MULTI-3-SCACHE}_2$

The following approximate the overhead of messages passed during each Jacobi iterations, on an 128-processor system with the array distributed in blocked fashion (144 by 72 blocks).

$\text{MULTI-2-REM-128}_{64} = 6 \cdot \text{REM-READ}$

$\text{MULTI-2-REM-128}_{32} = 13 \cdot \text{REM-READ}$

$\text{MULTI-2-REM-128}_{16} = 27 \cdot \text{REM-READ}$

$\text{MULTI-2-REM-128}_8 = 54 \cdot \text{REM-READ}$

$\text{MULTI-2-REM-128}_4 = 108 \cdot \text{REM-READ}$

$\text{MULTI-2-REM-128}_2 = 216 \cdot \text{REM-READ}$

$\text{MULTI-2-REM-128}_1 = 432 \cdot \text{REM-READ}$

MULTI-2-REM-128 is the overhead of messages passed in the Jacobi steps, per solution generated.

$\text{MULTI-2-REM-128} = 64 \cdot \text{MULTI-2-REM-128}_{32} + 32 \cdot \text{MULTI-2-REM-128}_{32} + 16 \cdot \text{MULTI-2-REM-128}_{32} + 8 \cdot \text{MULTI-2-REM-128}_{32} + 4 \cdot \text{MULTI-2-REM-128}_{32} + 2 \cdot \text{MULTI-2-REM-128}_{32} + 1 \cdot \text{MULTI-2-REM-128}_{32}$

MULTI-3-REM-128 is the overhead of messages generated in the grid interpolation phase, per solution generated.

$\text{MULTI-3-REM-128} = 1296 \cdot \text{REM-READ}$

MULTI-F90, MULTI-F77, MULTI-CMF, and MULTI-CMAX are the predicted run times on the Y-MP C90/`f90`, Y-MP C90/`cf77`, CM-5/`cmf`, and CM-5/`cmf-cmax` pairs, respectively.

MULTI-CRAY = MULTI-BANK

MULTI-F90 = MULTI-CRAY

MULTI-F77 = MULTI-CRAY

MULTI-CM5 = 5 · MULTI-1 + 5 · MULTI-2-CACHE + 5 · MULTI-3-CACHE + 5 · MULTI-2-REM-128 + 5 · MULTI-3-REM-128

MULTI-CMF = MULTI-CM5

MULTI-CMAX = MULTI-CM5

G.5 `tomcatv` Kernel Predictions

```
C
C     J-LOOP
C
      M = 0
      DO    310    J = J1P,J2M
      JP = J+1
      JM = J-1
      M  = M+1
C
C     I-LOOP
C
      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  = 0.25*(XY*XY+YY*YY)
      B  = 0.25*(XX*XX+YX*YX)
      C  = 0.125 *(XX*XY+YX*YY)
      QI = 0.
      QJ = 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)
C
C     CALCULATE RESIDUALS ( EQUIVALENT TO RIGHT HAND SIDES...
C
      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
  310 CONTINUE
C
C     DETERMINE MAXIMUM VALUES OF RESIDUALS
C
      DO    270    J = 1,M
      DO    270    I = I1P,I2M
      IF(ABS(RX(I,J)).LT.ABS(RXM))  GOTO  262
      RXM  = RX(I,J)
      IRXM = I
      JRXM = J
  262 IF(ABS(RY(I,J)).LT.ABS(RYM))  GOTO  270
      RYM  = RY(I,J)
      IRYM = I
      JRYM = J
  270 CONTINUE
C
C     SOLVE TRIDIAGONAL SYSTEMS IN PARALLEL
C
      IF(M-1)601,201,301
  201 CONTINUE
      DO    102    I = I1P,I2M
      RX(I,1) = RX(I,1)/DD(I,1)
      RY(I,1) = RY(I,1)/DD(I,1)
  102 CONTINUE
      GOTO  601
  301 DO    103    I = I1P,I2M
      D(I,1) = 1./DD(I,1)
  103 CONTINUE
      DO    401    J = 2,M
      DO    401    I = I1P,I2M
      R        = AA(I,J)*D(I,J-1)
      D (I,J) = 1./(DD(I,J)-AA(I,J-1)*R)
      RX(I,J) = RX(I,J) - RX(I,J-1)*R
      RY(I,J) = RY(I,J) - RY(I,J-1)*R
  401 CONTINUE
      DO    411    I = I1P,I2M
      RX(I,M) = RX(I,M)*D(I,M)
      RY(I,M) = RY(I,M)*D(I,M)
  411 CONTINUE
      DO    501    J = 2,M
      K = M-J+1
      DO    501    I = I1P,I2M
      RX(I,K) = (RX(I,K)-AA(I,K)*RX(I,K+1))*D(I,K)
      RY(I,K) = (RY(I,K)-AA(I,K)*RY(I,K+1))*D(I,K)
  501 CONTINUE
C
C     ADD CORRECTIONS
C
      L = 0
      DO    290    J = J1P,J2M
      L = L+1
      DO    290    I = I1P,I2M
      X(I,J) = X(I,J)+RX(I,L)
      Y(I,J) = Y(I,J)+RY(I,L)
  290 CONTINUE
C
C     PREPARE OUTPUT OF CONVERGENCE BEHAVIOUR
C
  601 LL = LL+1
      WRITE (6,1300)LL,IXCM,JXCM,DXCM,
     1                  IYCM,JYCM,DYCM,
     2                  IRXM,JRXM, RXM,
     3                  IRYM,JRYM, RYM
      ABX  = ABS(RXM)
      ABY  = ABS(RYM)
      DMAX = AMAX1(ABX,ABY)
      IF(LL.LT.LMAX.AND.DMAX.GT.EPS)  GOTO  190
```

Figure 17: Core of the `tomcatv` Kernel

`tomcatv` performs the generation of a 257 by 257 grid using Thompson's method. The program is a member of the SPEC benchmarking suite and conforms to the Fortran 77 standard. The program consists of a set of singly and doubly-nested loops of depths one and two, each of which can be parallelized with varying degrees of difficulty. We refer to a `do`-loop which includes code up to label n as loop n. The `cf77`, `f90`, and `cmf-cmax` compilers parallelize all inner loops, and the doubly nested loop 310. Loop 270 is not parallelized by any compiler, and its run time composes a significant portion of the run time of `tomcatv` on all platforms.

The following are computations of the run time of the kernel:

TCV-CRAY-310-250 is the parallel run time of loop 310.

$$\text{TCV-CRAY-310-250} = 4 \cdot \text{RS-OHS}_{65025} + 18 \cdot \text{RS-LOAD}_{65025} + 22 \cdot \text{RS-ADD}_{65025} + 22 \cdot \text{RS-MUL}_{65025}$$

TCV-CRAY-103 is the parallel run time of loop 103.

$$\text{TCV-CRAY-103} = \text{RS-OHS}_{255} + \text{RS-LOAD}_{255} + \text{RS-DIV}_{255}$$

TCV-CRAY-401-INNER is the parallel run time of the inner loop of loop 401.

$$\text{TCV-CRAY-401-INNER} = 3 \cdot \text{RS-OHS}_{255} + 8 \cdot \text{RS-LOAD}_{255} + 4 \cdot \text{RS-MUL}_{255} + 3 \cdot \text{RS-ADD}_{255} + \text{RS-DIV}_{255}$$

TCV-CRAY-401 is the parallel run time of loop 401.

$$\text{TCV-CRAY-401} = 255 \cdot \text{TCV-CRAY-401-INNER} + 256 \cdot \text{SEQ-LOIN}$$

TCV-CRAY-411 is the parallel run time of loop 411.

$$\text{TCV-CRAY-411} = \text{RS-OHS}_{255} + 2 \cdot \text{RS-LOAD}_{255} + \text{RS-MUL}_{255}$$

TCV-CRAY-501-INNER is the parallel run time of the inner loop in loop 501.

$$\text{TCV-CRAY-501-INNER} = 2 \cdot \text{RS-OHS}_{255} + 6 \cdot \text{RS-LOAD}_{255} + 4 \cdot \text{RS-MUL}_{255} + 2 \cdot \text{RS-ADD}_{255}$$

TCV-CRAY-501 is the parallel run time of loop 501.

$$\text{TCV-CRAY-501} = 255 \cdot \text{TCV-CRAY-501-INNER} + 256 \cdot \text{SEQ-LOIN}$$

TCV-CRAY-290 is the parallel run time of loop 290.

$$\text{TCV-CRAY-290} = 2 \cdot \text{RS-OHS}_{65025} + 4 \cdot \text{RS-LOAD}_{65025} + 2 \cdot \text{RS-ADD}_{65025}$$

TCV-CRAY-270-ONE is the sequential run time of one iteration of the computation inside loop 270. Based on our measurements, the code inside either `if` statement is run 0.7 percent of the possible number of executions.

$$\text{TCV-CF77-270-ONE} = 4 \cdot \text{SEQ-ABSS} + 2 \cdot \text{SEQ-ARR2} + 2 \cdot \text{SEQ-CRSL} + 2 \cdot \text{RAND-READ} + 1.99 \cdot \text{SEQ-GOTO} + 0.014 \cdot \text{SEQ-ARR2} + 0.042 \cdot \text{SEQ-TRSL}$$

TCV-CRAY-270 is the sequential run time of loop 270.

$$\text{TCV-CRAY-270} = 65025 \cdot \text{TCV-CRAY-270-ONE} + 256 \cdot \text{SEQ-LOIN}$$

TCV-CRAY-ONE is the run time of the a single iteration of the mesh generation process on the Y-MP C90.

$$\text{TCV-CRAY-ONE} = \text{TCV-CRAY-310-250} + \text{TCV-CRAY-103} + \text{TCV-CRAY-401} + \text{TCV-CRAY-411} + \text{TCV-CRAY-501} + \text{TCV-CRAY-290} + \text{TCV-CRAY-270}$$

TCV-CF77 and TCV-F90 are the run times of the kernel on the Y-MP C90/`cf77` and Y-MP C90/`f90` pairs, respectively.

$$\text{TCV-CF77} = 100 \cdot \text{TCV-CF77-ONE}$$

$$\text{TCV-F90} = \text{TCV-CF77}$$

The following are calculations of the run time on the CM-5 platform. TCV-CMF is the run time on the CM-5/`cmf` pair. The `cmf` compiler does not parallelize the code at all, so we approximate the run time as the sum of the overhead host remote reads and writes and ignore other constributions. We calculate that `tomcatv` performs approximately 2.5 million host remote reads and 0.5 million host remote writes.

$$\text{TCV-CMF} = 2500000 \cdot \text{SREM-READ} + 500000 \cdot \text{SREM-WRITE}$$

Note that all parameters TCV-CMAX-n are calculated as the corresponding parameter TCV-CRAY-n, with the exception that all parameters which represent operations on single precision operands are replaced by their double precision counterparts. TCV-CMAX-270-ONE is the run time of one execution of the inner computation of loop 270.

$$\text{TCV-CMAX-270-ONE} = 2 \cdot \text{SREM-READ}$$

TCV-CMAX-270 is the run time of loop 270.

$$\text{TCV-CMAX-270} = 65025 \cdot \text{TCV-CMAX-270-ONE}$$

TCV-CMAX-OH represents the overhead due to block references and remote reads. We assume an 8 by 16 blocked array distribution. We calculate that there are approximate 3666 remote reads per iteration, and 4100 block loads/stores. Since the block overhead on the CM-5 is represented by the *full* block model is largely independent of the number of elements loaded, we approximate all block loads using the OH-FULLBLOCK_0.

$$\text{TCV-CMAX-OHBLOCKTOTAL} = 66049 \cdot \text{OH-FULLBLOCK}_0$$

$$\text{TCV-CMAX-OH} = 3666 \cdot \text{REM-READ}4100 \cdot \text{OH-FULLBLOCKTOTAL}$$

$$\text{TCV-CMAX-ONE} = \text{TCV-CMAX-310-250}+\text{TCV-CMAX-103}+\text{TCV-CMAX-401}+\text{TCV-CMAX-411}+ \\ \text{TCV-CMAX-501} + \text{TCV-CMAX-290} + \text{TCV-CMAX-270} + \text{TCV-CMAX-OH}$$

$$\text{TCV-CMAX-3-ONE} = 3 \cdot \text{SREM-READ} + 2 \cdot \text{SREM-WRITE}$$

$$\text{TCV-CMAX-3} = 65025 \cdot \text{TCV-CMAX-3-ONE}$$

TCV-CMAX is the run time of the kernel on the CM-5/`cmf-cmax` pair.

$$\text{TCV-CMAX} = 100 \cdot \text{TCV-CMAX-ONE} + \text{TCV-CMAX-3}$$