Reducing Parallel Overheads Through Dynamic Serialization*

Michael J. Voss and Rudolf Eigenmann

Purdue University
School of Electrical and Computer Engineering

Abstract

If parallelism can be successfully exploited in a program, significant reductions in execution time can be achieved. However, if sections of the code are dominated by parallel overheads, the overall program performance can degrade. We propose a framework, based on an inspector-executor model, for identifying loops that are dominated by parallel overheads and dynamically serializing these loops. We implement this framework in the Polaris parallelizing compiler and evaluate two portable methods for classifying loops as profitable or unprofitable. We show that for six benchmark programs from the Perfect Club and SPEC 95 suites, parallel program execution times can be improved by but as much as 85% on 16 processors of an Origin 2000.

1 Introduction

Parallel program execution can lead to significant performance gains in a wide range of application areas. For general users it is of importance that parallelism can be exploited in programs written in standard, sequential languages. To this end, significant advances have been made in automatically recognizing parallelism in such programs and translating them into parallel form. However, identifying parallelism is only a first step. Executing parallel code efficiently is not straightforward. We have found that even well-structured parallel applications on current shared-memory machines may run slower than their serial counterparts, if they contain parallel regions that cannot amortize the overheads associated with their parallel execution.

With the new OpenMP shared-memory programming API, applications can now be expressed in a form that allows their parallel execution on most available shared-memory

*This work was supported in part by U. S. Army contract DABT63-92-C-0033, NSF award ASC-9612133, and an NSF CAREER award. This work is not necessarily representative of the positions or policies of the U. S. Army or the Government.

machines. This emerging standard makes it possible to write portable parallel programs. One issue when doing so is that portable programs cannot easily take advantage of advanced knowledge of the machine configurations on which they will run. This means that no decisions can be made by the programmer or the compiler that require knowledge about processor speeds, memory configuration, network topology or even the number of processors. A further issue is that, although many program optimizations can be done that do not require these parameters, the performance of the resulting code is very difficult to predict. Much work is necessary to provide techniques that can lead to portable parallel programs while exploiting key features of the specific machines on which they execute.

The specific goal of the work presented in this paper is to recognize situations in which the parallel execution of a code section would perform less than its original serial version, and to "undo" the parallel execution dynamically at runtime. In our previous work we have measured that significant performance improvement can result from such a scheme. These parallel execution overheads include factors such as fork/join costs, communication overheads, and added memory latency for accessing remote data. Often, the work performed by a program region depends on the input and may vary as the program runs. Thus, the overheads are a function of the program, the program input, the machine configuration, and the execution time. Most of these parameters are only known at run-time. Hence, deciding whether a parallel code region will be profitable must be done in a dynamically adaptive runtime scheme.

Given the input data and machine parameters, it may be possible to model each parallel program region before it executes and determine if it should be replaced with its serial counterpart. However, modeling program performance can be complex and to do so for each invocation of each region would likely introduce significant overhead. Instead, we propose a method similar to an inspector-executor scheme to make the decision. Each loop in a given program is first executed in parallel, assuming that the parallelism will be beneficial. The loop is timed as it executes and this timing is used to decide if the loop is dominated by parallel overheads. This decision guides subsequent executions of the loop and will be reconsidered when necessary.

In Section 1.1, we describe two approaches to dynamically decide whether a parallel loop is profitable. In Section 1.2, we give an overview of Polaris, the parallelizing compiler in which we have implemented our scheme. Section 1.3 describes related work. In Section 2, we present an analytical evaluation of the two proposed classification schemes. It is shown that for six benchmark programs, performance may be improved as much as 85%, and on average 26%, by applying these techniques. In Section 3, our framework for the dynamically adaptive method is described, as well as the two schemes implemented in Polaris that use this framework. Section 4 gives experimental results for 6 benchmarks executed on 16 processors of a Cray Origin 2000 system. Section 5 concludes the paper.

1.1 Two Approaches for Deciding Loop Profitability

We evaluate two approaches that classify parallelism as profitable or unprofitable based upon the measured parallel loop time. The first approach, *scaled-test*, compares the measured parallel time to an estimated serial time. If the parallel time is longer, it must be dominated by overheads and the loop is classified as unprofitable. The strength of this approach lies in the fact that the real parallel time is used, and not an estimate.

In the *scaled-test* approach, it is the serial time of the target machine that is modeled. In our implementation this is done by profiling the application on a base machine, and scaling the loop timings based upon a microbenchmark executed at the start of the application. Therefore, this approach estimates the 1 processor performance of the target machine, which is much simpler than predicting parallel performance. Many machine parameters can be ignored, such as the number of processors, network topology and local versus remote access latency. These parameters are essential for predicting parallel performance, but unnecessary for estimating 1 processor time.

The second approach, *overhead-test*, classifies a loop as unprofitable if the parallel time is below a certain threshold and so the loop must be dominated by overheads. The strength of this approach is that the threshold can be directly measured at the start of the program execution. Figure 1 illustrates the basic idea. We model the loop execution time as a parallel startup overhead (t_s), plus the work done by the loop (t_p), plus the parallel join overhead (t_e). The total parallel time T_p is then $t_s + t_p + t_e$. If we assume a perfect speedup, $T_{serial} = p \times t_p$, where p is the number of processors in the system. Therefore if $p \times t_p < T_p$, no gain is possible from running the loop in parallel.

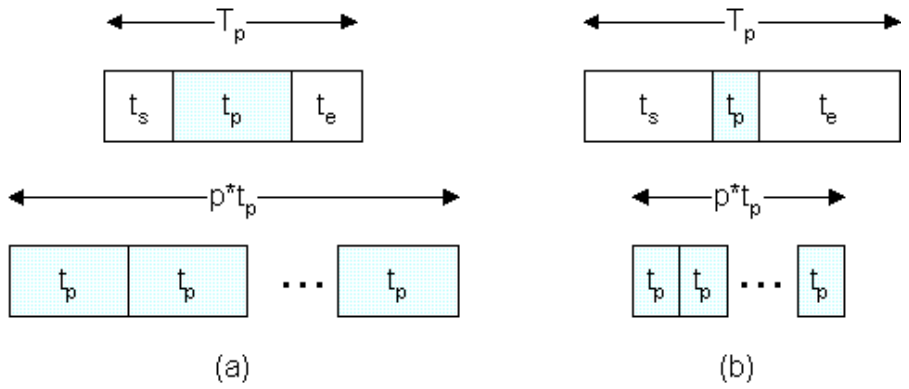

Figure 1: Choosing profitable parallelism using the parallel time: (a) the work is suff icient to amortize the overheads and (b) the overheads dominate.

We cannot measure t_p directly, but we can measure the start/end overheads, $t_{ov} = t_s + t_e$, of an empty parallel loop and the actual parallel time of the loop. Using just t_{ov} and T_p, both measurable run-time quantities, the following inequalities must hold if a parallel loop performs enough work to speed up:

$$t_p \times p > T_p \tag{1}$$

$$(T_p - t_{ov}) \times p > T_p \tag{2}$$

$$T_p > t_{ov} \times \frac{p}{p-1} \tag{3}$$

The right hand side of the inequality, given as Equation 3, can be evaluated once at the start of the program execution, making this test a simple comparison of the measured parallel time and a run-time constant. Unlike the *scaled-test*, the *overhead-test* makes an assumption as to the source of the parallel overheads. It assumes that the fork/join costs are the only source of parallel overhead and thus neglects communication overheads and differences in memory access latencies.

Both tests need only be applied once if the context of the loop remains unchanged. The framework we describe in Section 3, re-applies these tests if the number of iterations that a loop executes changes in a way that may effect its classification. (i.e. an unprofitable loop has an increase workload or a profitable loop has a decreased workload.)

Both schemes present challenges. In the *scaled-test* scheme, a reasonably accurate serial time must be estimated for each parallel loop. In the *overhead-test* scheme, an accurate threshold must be determined. This paper evaluates the merits and practical considerations of each approach. We do this first by presenting an analytical evaluation of the techniques, and then an experimental evaluation using several benchmarks from the Perfect Benchmarks and the SPEC95 floating-point suite.

1.2 The Polaris Parallelizing Compiler

This work is done as an extension of the Polaris parallelizing compiler. Polaris is a source-level restructurer and compiler infrastructure, developed by Purdue University and the University of Illinois at Urbana-Champaign. It can be used to transform sequential programs into one of several parallel forms [1, 2]. Its primary domain is Fortran, although work is currently being done to extend its capabilities to non-Fortran languages [3]. Polaris includes many advanced techniques such as scalar and array privatization, scalar and array reduction recognition, induction variable substitution and interprocedural analysis, which are used to recognize and exploit loop-level parallelism. In this study, we use the OpenMP back-end developed at Purdue University.

Polaris' major strength is detecting parallelism. Currently, it does not include advanced techniques to improve cache behavior or to reduce synchronization. In order to obtain reasonable performance, we found it necessary to add simple optimizations to address these issues. First, loop interchanging is performed to increase the number of stride-1 accesses. Second, if a parallel loop is an inner loop in a nest, the synchronization

point is moved as far out as possible in the nest. Both loop interchanging and synchronization point movement are subject to legality constraints. Figure 2 shows an example of these techniques as they would be applied by Polaris.

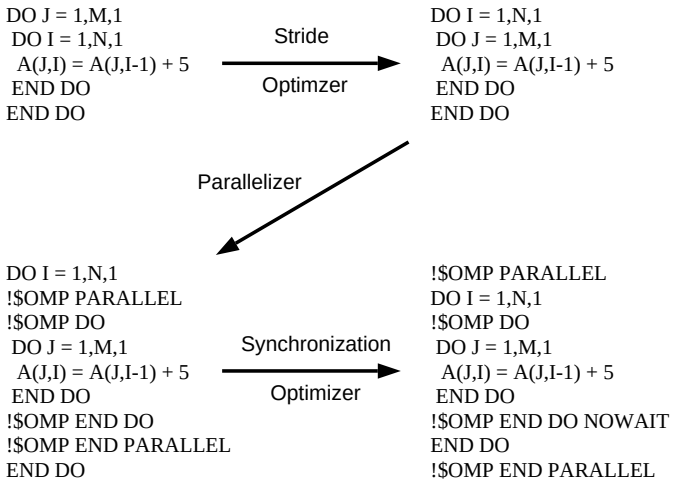

Figure 2: An example of the operation of the Polaris compiler: A sequential DO loop is the input. First the loop nest is interchanged to increase the number of stride-1 accesses. Next, the loop is parallelized. Finally, the synchronization point is moved outward in the loop nest.

In order to evaluate the techniques proposed in Section 3, a pass was added to perform the necessary code modifications and instrumentation. Polaris provides powerful utilities for instrumenting programs as well as collecting program statistics. All instrumentation used to generate the results in Section 4 were automatically inserted by the compiler. Much of the data presented in this paper was gathered and visualized through the use of the Ursa Minor/Major Toolset [4, 5].

1.3 Related Work

Several projects have developed techniques to avoid excessive overheads by serializing parallel loops. At the University of Illinois, work has been done to model each loop based on its operations, and at run-time use this model and the iteration count to serialize loops. This work has also been implemented in the Polaris compiler. In the SUIF compiler, a simple heuristic based on the number of lines in the loop body and the iteration count is used [6]. Both of these techniques are inherently non-portable. The first approach requires detailed knowledge of the time that each operation requires on the target machine. The second approach requires that a single threshold be chosen. In contrast, our approach adapts to the architecture on which the program executes.

Techniques used to serialize unprofitable loops aim at predicting the performance of a parallel loop. Much work has been done in the area of performance prediction [7, 8, 9].

However, these techniques can be computationally intensive and therefore too expensive to include in a run-time test. In contrast, the performance prediction approaches used in this paper are simple and make use of profiling and direct program measurements.

Our technique uses a scheme similar to an inspector-executor approach to identify non-profitable loops. The inspector-executor model has been used by others for scheduling parallel loops, orchestrating communication, and for performing run-time data dependence analysis [10, 11]. In [11], statistics are collected while a loop executes in parallel, and after its completion, it is determined if the loop was in fact parallel. We similarly collect execution statistics, namely the execution time, but use this information to determine if a loop, already known to be parallel, should be executed in parallel.

2 An Analytical Evaluation

Figure 3 shows the normalized execution time of six benchmark programs, and the modeled times of these codes for both the *scaled-test* and *overhead-test* approaches. Both schemes' execution times are modeled by summing individual loop timings selected from a profile of either a 1-processor execution, or a 16-processor execution, of the original parallel program run on an Origin 2000. The *scaled-test* time is calculated by using the 16 processor loop time unless the 1 processor loop time is smaller, in which case the 1 processor time is used. The *overhead-test* time is calculated by using a value of 50 μs as T_{ov}, and evaluating Equation 3. The value of T_{ov} was determined experimentally by timing an empty parallel loop. If Equation 3 shows that the loop cannot speedup, the 1 processor time is used, otherwise the 16 processor time is used.

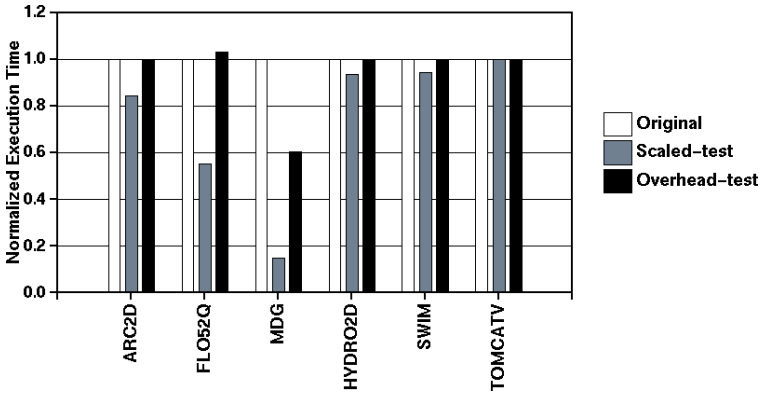

Figure 3: Normalized execution time of benchmarks executed on 16 processors of an Origin 2000. The bars labeled Original are the actual measured time. Scaled-test and Overhead-test are analytical models of the schemes proposed in Section 1.1

These approximations show that improvements are possible on five of the six codes. The *scaled-test* is always able to outperform the *overhead-test*, as should be expected, since the former is assumed to be 100% accurate in identifying loops to serialize. Signif-

icant gains can be seen in the programs flo52 and mdg, while arc2d, hydro2d and swim show small gains for the scaled-test approach. Interestingly, flo52 shows a dramatic gain for the *scaled-test*, while a small decrease in performance is seen when the *overhead-test* is used.

Figure 4 shows the percentage of loops that the *overhead-test* scheme can correctly identify. In the analytical model used, the *scaled-test* approach is assumed to identify correctly all loops that have a shorter 1-processor time than 16-processor time. The assumption of the *overhead-test* model is that the fork/join overhead is the only significant parallel overhead and that loops can be correctly classified using only this metric. It is clear, from Figure 4, that this is an over simplification. The *overhead-test* is seen to correctly classify only 63% of the parallel loops.

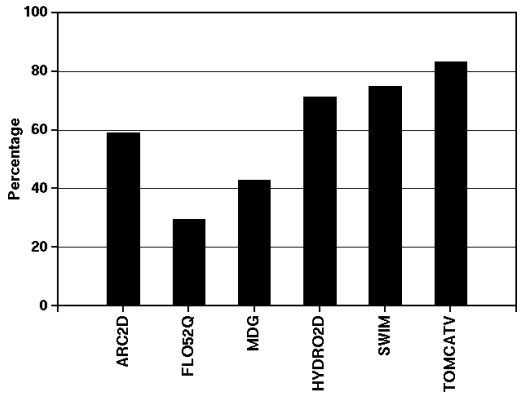

Figure 4: Percentage of loops correctly classified by the analytical model of the *overhead-test* scheme.

The flo52 application, which shows poor performance when the *overhead-test* is used in Figure 3, is shown to have a less than 30% classification accuracy. The mdg benchmark, with an accuracy of slightly over 40%, is able to reduce the execution time by only 40% with *overhead-test*, while the *scaled-test* is able to reduce it by 85%. The average improvement shown for the six codes in Figure 3 is 26% for the *scaled-test* approach, and 6% for the *overhead-test* approach.

The execution times generated by our analytical evaluation may be smaller, or larger, than those seen from an actual implementation. Our framework, described in Section 3, will re-evaluate the profitability of a parallel loop, if its context changes during the course of the program execution. The analytical model presented above, chose a fixed attribute (serial or parallel) for each loop. On the other hand, the actual implementation will incur additional overheads due to the testing and book-keeping required by these schemes.

3 Compiler Implementation of Dynamic Serialization

3.1 The Dynamic Adaptation Framework

The inspector-executor framework must provide a means to correctly classify parallel loops as profitable or unprofitable. It also must provide the capability to dynamically adapt to changes in a loop's context. And finally, it must avoid mis-classifications due to program startup effects. Figure 5 shows the state transition diagram of the framework we propose to meet these requirements.

Initially, each loop is classified as PARALLEL and starts in the WARMUP state. Each loop is allowed to execute all of its iterations once in parallel before any timing is done. This is done so that the timing of the loop is not influenced by cold misses in the cache.

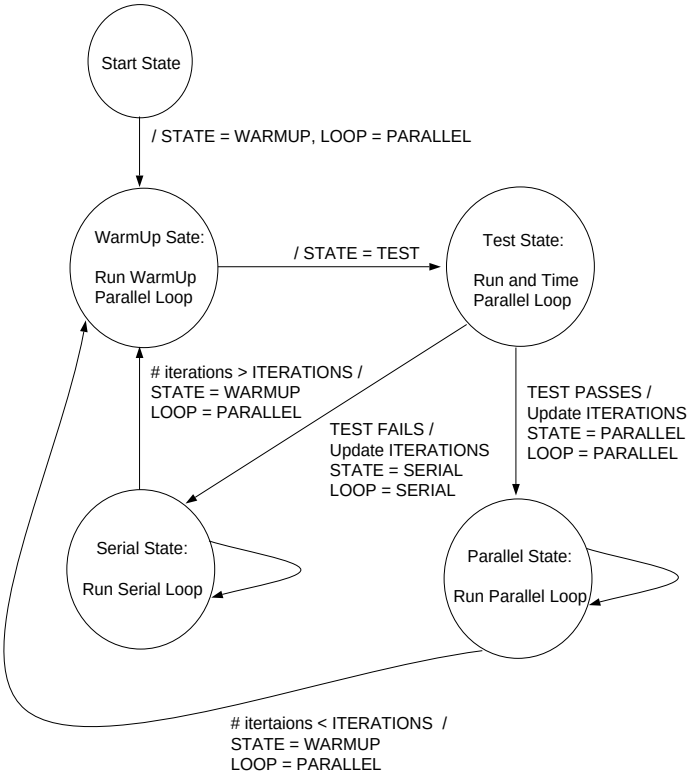

Figure 5: The state transition diagram for classifying loops.

After a loop in the WARMUP state executes once, it moves to the TEST state. The next execution of the loop will be again run in parallel, but this time it will be timed. The timing is then used to decide if the loop should be executed sequentially the next time it is executed. If the test determines that the loop should be serial, the state of the loop is changed to SERIAL, the loop is classified as SERIAL and the current number of

iterations is recorded. If the test determines that the loop should be parallel, the state of the loop is changed to PARALLEL, the loop is classified as PARALLEL and the current number of iterations is recorded.

Once a loop reaches either the SERIAL or PARALLEL state, it will stay in this state until the program ends unless the number of loop iterations changes. If a SERIAL loop increases its number of iterations, the loop may have enough work to now be parallel, so it is reclassified as PARALLEL and is moved back into the WARMUP state. Likewise, if a PARALLEL loop has its number of iterations decreased, the loop may now need to be serialized, and so it is moved back into the WARMUP state.

This paper uses the model in Figure 5 as a basis for a proof of concept. There are many optimizations that can be done to improve the model. For example, the decision to re-test could be refined to reduce unnecessary testing: a loop that executes very quickly, and is therefore serialized, may not need to be re-tested if its iteration count increases by only little. One may note that loops that execute only once may not be correctly classified. However, loops that slow down are usually small, and these loops would only become important if they were executed frequently.

3.2 Compilation Scheme for the Scaled-test Approach.

Figure 6 shows the compilation steps taken to generate a program that uses the *scaled-test* technique. Since we are interested in automatic parallelization, we assume we begin with a sequential Fortran program. This program is run through the Polaris compiler to generate an optimized OpenMP program. This program is instrumented by Polaris in such a way that the average per-iteration loop time will be collected for each parallel loop as the program runs. This parallel program is then run on 1 processor of a *base machine*. The base machine can be any machine available to the developer, it need not be a multiprocessor.

The instrumented program will generate the per-iteration loop times that will be fed back into the Polaris compiler. Polaris provided with these loop timings and the original program, generates another parallel OpenMP program, embedding the state machine, shown in Figure 5, as well as a microbenchmark used to scale the base machine timings. Currently, we use a small matrix multiplication kernel to generate the scaling factor. The time that this kernel takes on the base machine is known (i.e. can be measured once for a given machine). At run-time, the kernel is timed on the target machine. The target machine kernel timing, divided by the base machine kernel timing, is used to scale each measurement from the base machine profile.

We use per-iteration loop times to attempt to compensate for changes in the working-set size. If the number of iterations of a parallel loop is a function of input parameters, this will be correctly scaled by our technique. If, however, a parallel loop has inner loops that dependent on input parameters, an incorrect time may be predicted. In ongoing work we are developing techniques for making the scaling factor more data-set independent. For the benchmarks used in this study, we used the same data-set on the base machine

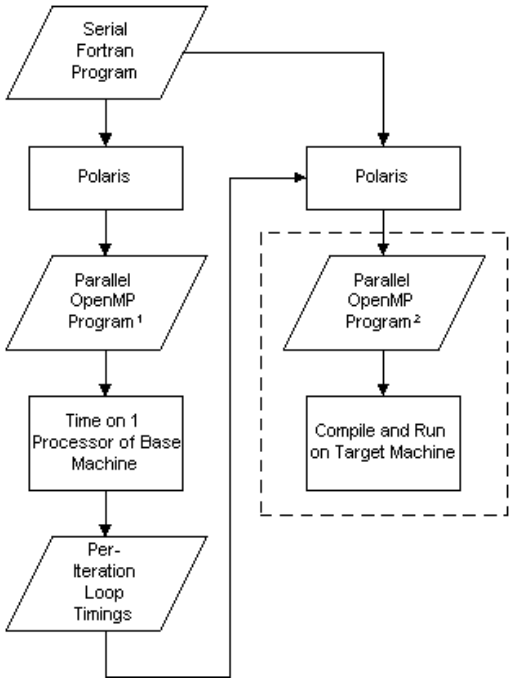

Figure 6: Generating code that chooses parallelism by comparing to a predicted serial time.

and target machine.

3.3 Compilation Scheme for the Overhead-test Approach.

Again, as with the *scaled-approach*, we begin with a sequential Fortran program. Polaris is used to generate a parallel OpenMP program, and embed the state machine and a test to determine the value of t_{ov}. No profiling is required in this approach, so Polaris is only run once on the application. The value of t_{ov} is obtained by measuring the time of an empty parallel loop. Code to perform this measurement is inserted at the beginning of each application. This value is multiplied by $p/(p-1)$ as in Equation 3. The number of processors, p, is obtained by a library call provided by the OpenMP API.

As discussed in Section 1.1, the *overhead-test* approach assumes that the parallel overheads are comprised solely of the fork/join overheads. We approximate these overheads by timing an empty parallel loop. In previous work [12], we have measured the fork overhead to also be linearly dependent on the number of shared variables, because these are passed by reference into the loop body. Here we assume that this effect is negligible compared to the loop scheduling and join overheads.

4 Experimental Results

4.1 Improvements in Normalized Execution Time

Figure 7 shows the execution time of six programs run on 16 processors of an Origin 2000. For the *scaled-test* approach, base timings were collected on a SPARCstation 20. The execution times are normalized to the program as originally parallelized by Polaris without any overhead avoidance scheme (i.e. all parallelizable loops are executed in parallel). The *scaled-test* approach yields an average decrease of 15% and the *overhead-test* yields an average decrease of 6%.

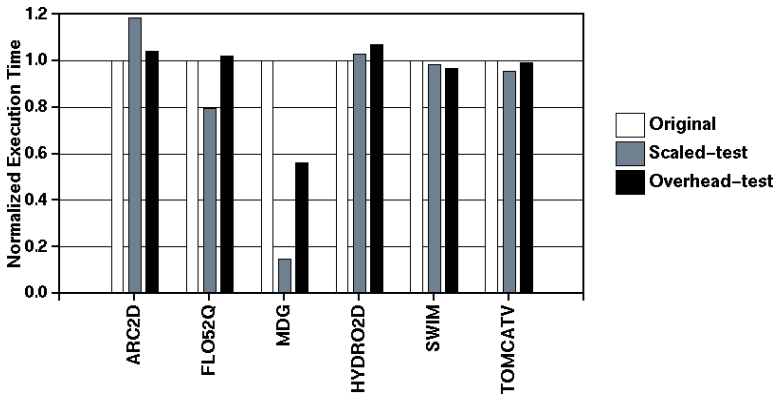

Figure 7: The normalized execution time on 16 processors of an Origin 2000.

Four of the six programs show improvements when the *scaled-test* is used with our inspector-executor framework. The mdg benchmark shows performance comparable to that predicted by our analytical model, decreasing its time by nearly 85%. Flo52 shows a decrease of 20%. Swim and tomcatv show improvements of less than 10%, which is consistent with the analytical model. However, both arc2d and hydro2d show a performance degradation.

The arc2d benchmark, for which the analytical model predicted a nearly 20% decrease in execution time, has a nearly 20% *increase* in execution time. This program consists of 97 parallel loops which contribute to its less than 8 second running time on 16 processors. Many of these loops are executed at least 100 times, so one might suspect that the overhead associated with the framework is the culprit. However, the *overhead-test*, which performs the same number of tests, has only a 2% increase in its execution time. A closer analysis of this program suggests that the performance degradation is an indirect effect of the program transformation. The *scaled-test* is able to identify more non-profitable loops than the *overhead-test*. However, our analysis does not include inter-loop cache effects. If, for example, two consecutive parallel loops access the same data and are distributed in the same manner, data cached by the first loop is reused by the second loop. If the first loop is serialized, this cache behavior is changed, causing misses to occur in both loops. In effect, this is a parallelization benefit not accounted for by our model. The correct

handling of the situation requires global analysis, which is the subject of our ongoing work, but beyond the scope of this paper.

Again examining Figure 7, the *overhead-test* shows an improvement in three programs. In two of these codes, the *scaled-test* performs better, as expected from the analytical model. Again, mdg shows the largest improvement in execution time, a decrease of nearly 45%. Both swim and tomcatv, which the analytical model predicted would not benefit from this scheme, show small decreases in execution time.

4.2 Classification Accuracies of the Two Approaches

To better understand the differing performance of these two approaches, and the variations from the analytical model, we measured the classification accuracies. These accuracies are based upon the loop performance in the original parallel program. If the 1 processor time measured for the original program is smaller than the 16 processor time, loop parallelization is considered to be profitable, otherwise it is unprofitable. In Figure 8, the percentage of loops that are correctly classified by each scheme is shown. The values from Figure 4 are included for the sake of comparison. The *scaled-test* accuracy is equal to, or better than, the *overhead-test* in all cases.

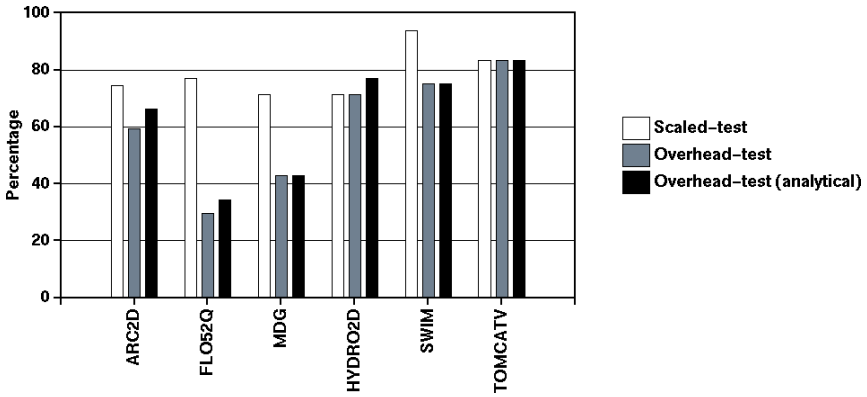

Figure 8: The percentage of loops correctly classified by each test.

The relationship between the accuracies in Figure 8 and the performance improvements shown in Figure 7 are not direct. One clear results is that the *scaled-test* does provide a more reliable means of identifying loops. Figure 9 provides more insights. It shows the percentage of loops dominated by overheads that are correctly classified by each scheme. The percentages shown in Figure 9, are based only on loops that execute more than once. (As described in Section 3, loops that only execute once are not classified by our scheme but can be considered unimportant.)

The only programs that show a loss in performance with the *scaled-test* are arc2d and hydro2d. The percentage of unprofitable loops that are correctly identified in these codes are 60% and 37% respectively. All other applications have classification rates larger than 80% and show decreased execution times. The *overhead-test* approach is unable to

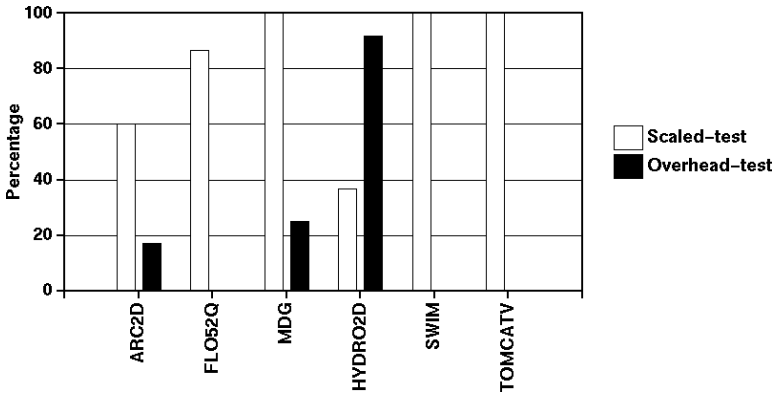

Figure 9: The percentage of nonprofitable loops that are correctly classified by each scheme.

correctly identify any of the unprofitable loops in flo52, swim and tomcatv, while the *scaled-test* has a near perfect rate in each of these codes. This again shows the need for improved models that consider other overhead factors, in addition to the fork/join cost of parallel loops.

In Figure 10, the percentage of loops which speed up and are accurately classified as PARALLEL by each approach is given. It is evident from Figure 7 that the incorrect classifications in flo52, hydro2d and tomcatv must be of insignificant loops, because their performance impact in Figure 7 is small.

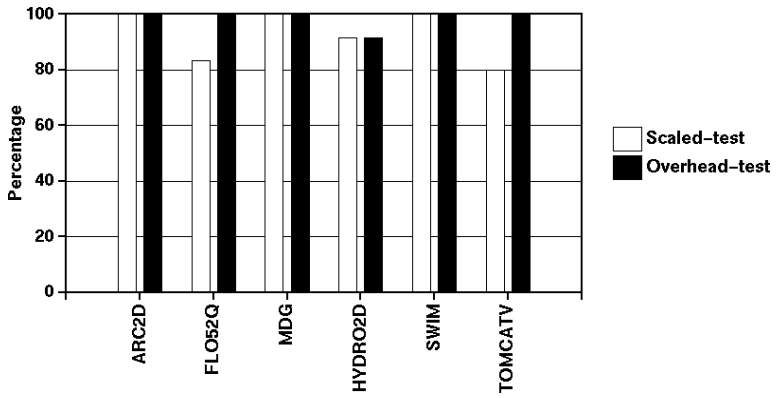

Figure 10: The percentage of loops that speed up that are successfully classified by each scheme.

5 Conclusion

Parallelism, if efficiently exploited, can lead to significantly reduced execution times. One issue, which has increased in importance with the newest machine generation, is that

the parallel execution may incur overheads that degrade performance. Such overheads may be reduced by carefully considering the execution and machine parameters of a parallel program. However, this may lead to non-portable programs. In this paper we are addressing the issue of maintaining program portability while factoring in parameters of machines and execution environments. For our work we use the portable OpenMP parallel API.

We have proposed a framework similar to an inspector-executor model for identifying loops that are dominated by parallel overheads. We have shown that two tests can be used to identify such loops in our framework. The first scheme, *scaled-test*, compares the measured parallel loop time to a predicted serial time, classifying the parallelism as profitable if the parallel time is smaller. The second scheme, *overhead-test*, makes the decision based upon the fork/join overhead of a parallel loop. If the parallel time is too small, the loop will be dominated by this overhead and the loop is classified as unprofitable. This decision is made at runtime and adapts to changing environments.

In our analytical evaluation of these two approaches on six benchmarks from the Perfect Club and SPEC 95 benchmark suites, we found that on 16 processors of an Origin 2000, the *scaled-test* should decrease the program execution by an average of 26%. The *overhead-test* is expected to decrease the execution times by an average of 6%.

We implemented both techniques using the Polaris parallelizing compiler and experimentally evaluated each on 16 processors of an Origin 2000. We found that the *scaled-test* improved the execution time of the six codes by an average of 15%, and the *overhead-test* decreased the execution times by 6%. A detailed analysis of the loop classification accuracies was presented for each scheme, showing that the *scaled-test* was always equal to or better than the *overhead-test* in all cases.

Although both schemes showed an improvement of the benchmarks on average, each caused performance degradations on some of the applications. Future work will be done on improving the classification accuracies of these schemes and on reducing the overheads associated with the tests.

Perhaps most importantly, this paper has shows the proof of concept of a novel compilation scheme generating a program that adapts to dynamically changing environments. The overhead reduction technique is one application of this scheme. Within the same framework we can measure many other parameters, such as cache statistics, load factors, or properties of a heterogeneous machine suite, and dynamically adapt the program to the situation at hand.

References

[1] Seon Wook Kim, Michael J. Voss, and Rudolf Eigenmann. Moerae: Portable, thread-based interface between parallelizing compilers and shared-memory multiprocessors. Technical report, Purdue University School of ECE, High-Performance Computing Lab, 98.

[2] Jee Ku. The design of an efficient and portable interface between a parallelizing compiler and its target machine. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1993. CSRD report no. 1500.

[3] Richard L. Kennell and Rudolf Eigenmann. Automatic parallelization of C by means of language transcription. Technical Report ECE-HPCLab-98205, Purdue University School of ECE, High-Performance Computing Lab, 98.

[4] Insung Park and Rudolf Eigenmann. URSA MAJOR: Exploring Web technology for design and evaluation of high-performance systems. In *Int'l Conf. on High-Performance Computing and Networking, HPCN Europe'98*, pages 535–544, April 98.

[5] Insung Park, Michael J. Voss, Brian Armstrong, and Rudolf Eigenmann. Parallel programming and performance evaluation with the URSA tool family. *Int'l Journal of Parallel Programming*, 1998. to appear.

[6] Michael E. Wolf and Jennifer Anderson. skweel man page. basesuif-1.1.2, www-suif.stanford.edu.

[7] Brian Armstrong and Rudolf Eigenmann. Performance forecasting: Towards a methodology for characterizing large computational applications. In *Proc. of the Int'l Conf. on Parallel Processing*, pages 518–525, August 1998.

[8] R. H. Saavedra-Barrera and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical Report USC-CS-92-524, University of Southern California, 92.

[9] K. Gallivan, W. Jalby, A. Malony, and H. Wijshoff. Performance Prediction for Parallel Numerical Algorithms. *Int'l. Journal of High Speed Computing*, 3(1):31–62, February 1991.

[10] J. Saltz, R. Mirchandaney, and K. Crowley. Run time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5), May 1991.

[11] L. Rauchwerger and D. Padua. The LRPD Test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Languages Design and Implementation*, June 95.

[12] Michael Voss. Portable loop-level parallelism for shared-memory multiprocessor architectures. Master's thesis, Purdue University, School of Electrical and Computer Engineering, Dec 1997.