# Reducing Parallel Overheads Through Dynamic Serialization*

Michael 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 as much as 85% on 16 processors of an Origin 2000.*

## 1 Introduction

Identifying parallelism in a program is only a first step in generating efficient parallel code. In a previous study, 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 [5]. Detecting such regions in a portable program is difficult. 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 machines. In writing portable programs, however, a user/compiler cannot easily take advantage of advanced knowledge of the machine configurations on which they will run. This makes predicting parallel performance difficult if not impossible.

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. The impact of parallel overheads are a function of the program, the program input and the machine configuration and hence can only be determined at runtime. We propose a method similar to an inspector-executor scheme to identify unprofitable parallelism. Each parallel loop 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. In Section 3, our framework 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 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.

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.

*This work was supported in part by DARPA contract #DABT63-95-C-0097, NSF grant #9872516-EIA and an NSF CAREER award. This work is not necessarily representative of the positions or policies of the U. S. Government.

This approach estimates the 1 processor performance of the target machine, which is much simpler than predicting parallel performance.

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. 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, $T_{serial} = t_p \times p$, where p is the number of processors in the system. If $t_p \times p < T_p$, no gain is possible from running the loop in parallel.

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 $t_p = T_p - t_{ov}$, we can then find the following inequality that must hold if a parallel loop does not perform enough work to speed up:

$$T_p < t_{ov} \times \frac{p}{p-1} \quad (1)$$

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.

### 1.2 The Polaris parallelizing compiler

This work is done as an extension of the Polaris parallelizing compiler [1]. Polaris is a source-level restructurer and compiler infrastructure, developed by Purdue University and the University of Illinois at Urbana-Champaign. 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.

### 1.3 Related work

Several projects have developed techniques to avoid excessive overheads by serializing parallel loops. In the SUIF compiler, a simple heuristic based on the number of lines in the loop body and the iteration count is used [7]. This approach requires that a single threshold be chosen. A single threshold cannot capture the speed of more than 1 machine and is therefore not portable. Hall and Matronosi [2] have developed support in the SUIF runtime libraries for dynamically selecting the number of processors to execute a given parallel section. There evaluation metric, however, is throughput in a multiprogrammed environment, and thus a different problem is being addressed. In addition, they have embedded the decision making into their machine-specific SGI library, while our approach is embedded into the machine-independent program itself.

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 runtime data dependence analysis [4, 3].

## 2 Analytical evaluation

Figure 1.a 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 original parallel program runs all parallel loops in parallel. The *scaled-test* time is calculated by using the 16 processor loop time unless the 1 processor loop time is smaller. The *overhead-test* time is calculated by using a value of 50 $\mu s$ as $t_{ov}$, and evaluating Equation 1. The value of $t_{ov}$ was determined experimentally by timing an empty parallel loop. If Equation 1 shows that the loop cannot speedup, the 1 processor time is used, otherwise the 16 processor time is used.

These approximations show that improvements are possible on five of the six codes. The *scaled-test* is always able to outperform the *overhead-test* since the former is assumed to be 100% accurate in identifying loops to serialize. Significant 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 1.b shows the percentage of loops that the *overhead-test* scheme can correctly identify. 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 1.b, that this is an over simplification. The *overhead-test* is seen to correctly classify only 63% of the parallel loops.

The flo52 application, which shows poor performance when the *overhead-test* is used, is shown to

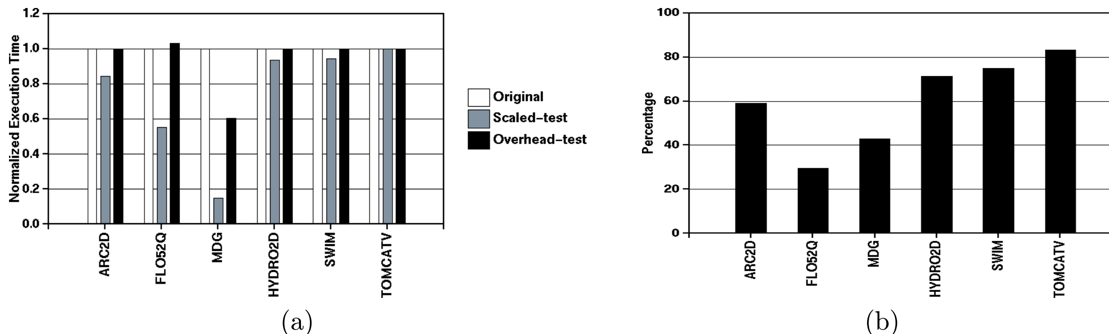


**Figure 1. Analytical evaluation of dynamic serialization: (a) the normalized execution time of benchmarks executed on 16 processors of an Origin 2000 (the bars labeled Original are the actual measured time) and (b) the percentage of loops correctly classified by the overhead-test scheme.**

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 1.a is 26% for the *scaled-test* approach, and 6% for the *overhead-test* approach.

## 3 Implementation

Figure 2 shows the state transition diagram of the framework we propose. This inspector-executor framework provides a means to correctly classify parallel loops as profitable or unprofitable, through the application of one of the previously described tests. It also provides the capability to dynamically adapt to changes in a loop's context. 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 in a way that can affect its classification. And finally, it avoids mis-classifications due to program startup effects through its WarmUp State.

This paper uses the model in Figure 2 as a basis for a proof of concept. There are many optimizations that can 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.

To use this framework with the *scaled-test* technique, requires several compilation steps. Since we are interested in automatic parallelization, we assume that

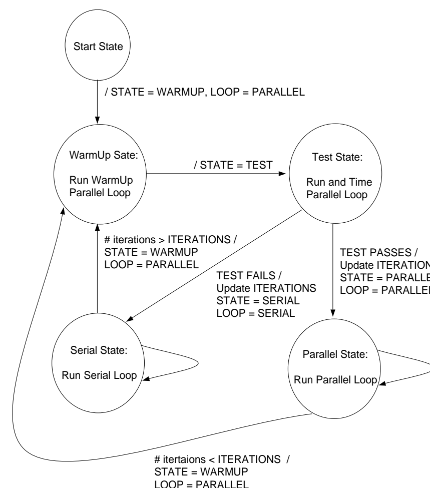


**Figure 2. Classifying loops.**

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*.

This instrumented program will generate per-iteration loop times that will be fed back into the Polaris compiler. Polaris generates another parallel OpenMP program, embedding the state machine, shown in Figure 2, 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. At runtime, 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. We are currently developing schemes for making the scaling more data-set independent. One possible approach to compensate for incorrectly scaled values, would be to time a loop if it is serialized, and compare this actual serial time to the last measured parallel time, re-parallelizing the loop if the actual serial time is larger. We will implement this in future work.

To generate programs using the *overhead-test*, we again 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 $t_{ov}$. 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.

## 4 Experimental results

Figure 3.a 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 with all parallellizable loops executed in parallel. The *scaled-test* approach yields an average decrease of 15% and the *overhead-test* yields an average decrease of 6%.

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. The performance degradation is an indirect effect of the program transformation due to inter-loop cache effects. For example, if 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. 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 3.a, 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. Both swim and tomcatv, which the analytical model predicted would not benefit from this scheme, show small decreases in execution time.

The classification accuracies of each scheme are shown in Figure 3.b. These accuracies are based upon the loop performance in the original parallel program. If the 16 processor time measured for the original program is smaller than the 1 processor time, loop parallelization is considered to be profitable, otherwise it is unprofitable. In Figure 3.b, the percentage of loops that are correctly classified by each scheme is shown. The *scaled-test* accuracy is equal to, or better than, the *overhead-test* in all cases.

Figure 3.c shows the percentage of loops dominated by overheads that are correctly classified by each scheme. The percentages shown in Figure 3.c, are based only on loops that execute more than once. 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.

In Figure 3.d, the percentage of loops which speed up and are accurately classified as Parallel by each approach is given. It is evident that the incorrect classifications in flo52, hydro2d and tomcatv are of insignificant loops, because their performance impact in Figure 3.a is small.

## 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 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 address the issue of maintaining program portability while factoring in parameters of machines and execution environments. 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

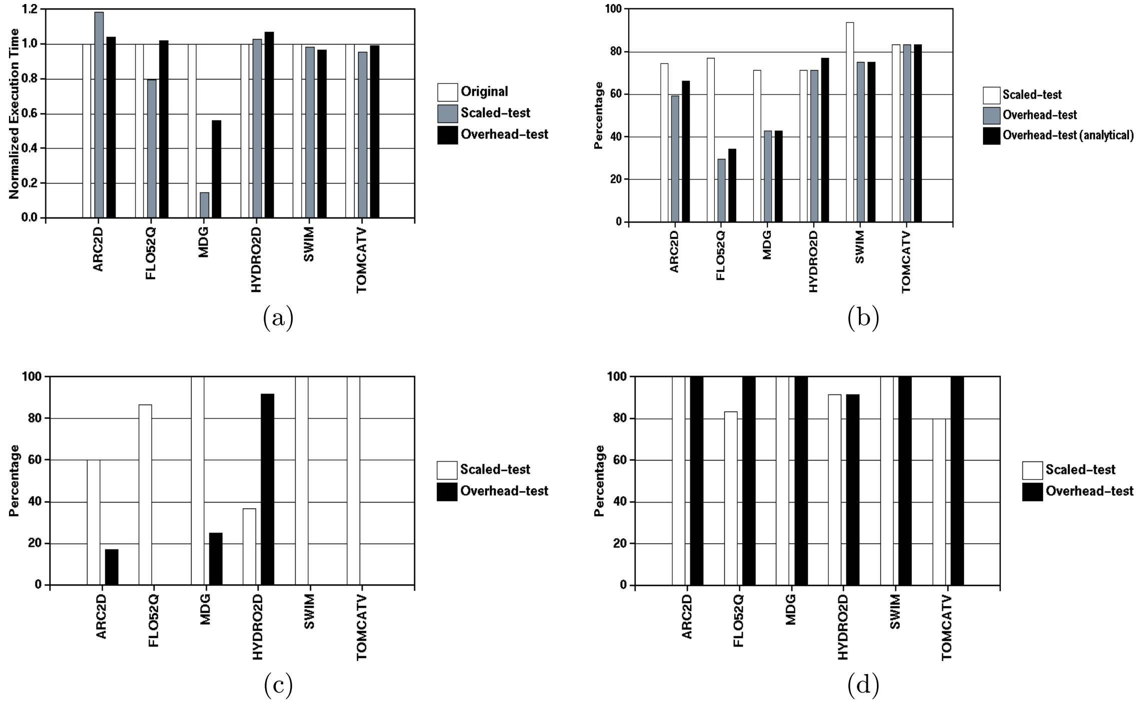


**Figure 3. Experimental results: (a) the normalized execution time on 16 processors of an Origin 2000, (b) the percentage of loops classified correctly by each test, (c) the percentage of nonprofitable loops correctly classified and (d) the percentage of loops that speed up that are correctly classified.**

scheme, *overhead-test*, makes the decision based upon the fork/join overhead of a parallel loop. 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 equal to or better than the *overhead-test* in all cases.

In future work, we will be combining dynamic serialization with other adaptive techniques in a general framework for generating *dynamically adaptive* parallel programs [6]. This new paradigm of dynamic adaptation may not only yield truly performance-portable programs but also lead to a new generation of optimizing compilers.

## References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced program restructuring for high-performance computers with polaris. *IEEE Computer*, Dec. 1996.

[2] M. W. Hall and M. Martonosi. Adaptive parallelism in compiler-parallelized code. In *Proc. of the 2nd SUIF Compiler Workshop*, Aug. 97.

[3] 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.

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

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

[6] M. Voss and R. Eigenmann. Dynamically adaptive parallel programs. In *Proceedings of the International Symposium on High Performance Computing*, Kyoto, Japan, May 99.

[7] M. E. Wolf and J. Anderson. skweel man page. basesuif-1.1.2, www-suif.stanford.edu, Apr. 1994.