Dynamically Adaptive Parallel Programs ⋆

Michael Voss and Rudolf Eigenmann

Purdue University, School of Electrical and Computer Engineering

Abstract. Dynamic program optimization is the only recourse for optimizing compilers when machine and program parameters necessary for applying an optimization technique are unknown until runtime. With the movement toward portable parallel programs, facilitated by language standards such as OpenMP, many of the optimizations developed for high-performance machines can no longer be applied prior to runtime without potential performance degradation. As an alternative, we propose dynamically adaptive programs, programs that adapt themselves to their runtime environment. We discuss the key issues in successfully applying this approach and show examples of its application. Experimental results are given for dynamically adaptive programs that seek to eliminate redundant runtime data dependence tests, to select the optimal tile size for tiled loops and to serialize loops that do not profit from parallelism.

1 Introduction

Many of today's well-known program optimization techniques are applied prior to runtime, and assume that detailed knowledge of the target machine and the program is available. Unfortunately, this information may be unknown until program execution. If the information a technique requires is not yet available, then the optimization is either not applied, or is applied using possibly incorrect assumptions about the architecture and application. Such assumptions can significantly limit the performance improvement, or even cause degradation.

In the context of parallel programs, the availability of portable parallel languages, such as the OpenMP API, exacerbates this problem. Portability is attractive for many reasons, such as ease of distribution, and use with new computing paradigms such as network computing [KF98,LLM88] and metacomputing [FK97]. However, portability requires that we do not use advanced knowledge of the target configuration. This loss of information means that much of the hand-tuning done on high-performance codes cannot be done in portable programs.

Since performance is a key issue in these programs, we cannot simply discard optimizations if they cannot be statically performed. We propose, as an alternative, *dynamically adaptive programs*. These programs perform optimizations dynamically as their usefulness and correctness can be evaluated. In this

⋆ 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.

paper we discuss such a scheme and highlight key performance issues. We also demonstrate its scope by applying it to three diverse example problems.

In our case studies, we first apply dynamic optimization to reduce redundant runtime data dependence testing and see that the adaptive code can outperform the original sequential code by 60%. Our scheme, which re-evaluates a loop's classification as serial or parallel only when necessary, executes 38% faster than a program which always performs a dependence test and is only 5% slower than a program which executes the loop with advanced knowledge of its data independence. Next, we apply the dynamically adaptive programming approach to tiling, and show that a matrix multiplication kernel, that automatically selects the best tile size, improves by as much as 75% over an untiled version. This dynamic scheme is within 5% of the best possible statically optimized program using advanced knowledge of the best tile size. Finally, we show that dynamically adaptive programs, automatically generated by the Polaris parallelizing compiler [BDE+96,BEF+96], can detect loops that cannot amortize the overheads associated with their parallel execution, and can serialize these, leading to reductions in parallel execution time as large as 85% on 16 processors of an Origin 2000. On average, programs with loops serialized based on profiling showed no larger improvement.

2 An Overview of Dynamically Adaptive Programs

Dynamically adaptive programs seek to apply optimizations that cannot have their usefulness or correctness evaluated prior to program execution. At runtime, these programs modify their behavior as these techniques can be applied and evaluated. The optimizations are then re-evaluated when the parameters effecting their usefulness or correctness change.

We have found that dynamically adaptive programs can often be written using one of three schemes. Figure 1 shows the state transition diagrams for the *binary selection*, *k-ary selection* and *iterative modification* approaches. In binary selection, an optimization is turned on or off for a given code section. This is done by simply verifying the correctness or usefulness of the optimization at runtime. The optimized code will be used if it is shown to be correct or have a better performance than the unoptimized code. This choice is then re-evaluated when the environmental parameters effecting its correctness/usefulness change. In this paper, our approaches to dynamic data dependence testing and dynamic serialization use the binary selection scheme. In both cases, it is determined whether a loop should be executed in parallel or sequentially, turning parallelism on or off. Our approach to tiling uses k-ary selection. We try several different tile sizes and select the one that yields the shortest execution time. Tiling could as well use iterative modification, and we discuss the reason for choosing k-ary selection in Section 4.

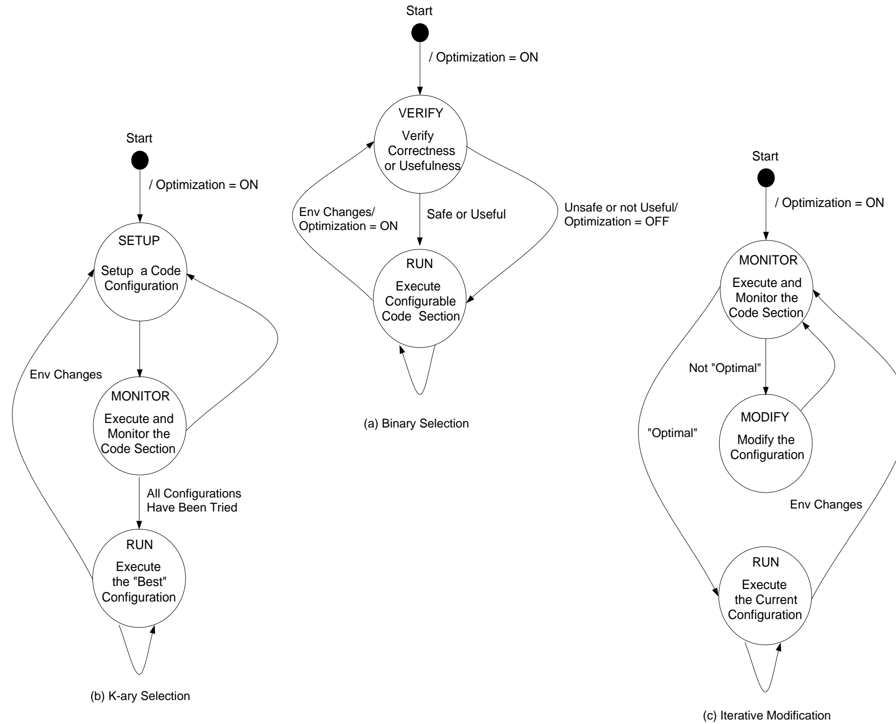

Fig. 1. Three schemes for dynamically adaptive programs.

3 Case Study 1: Runtime Dependence Tests

Runtime data dependence tests can be used when traditional compile-time dependence tests cannot determine if it is safe to execute a loop in parallel [SMC91,RP94,RP95]. A common case in which traditional tests fail is when an array is accessed with a subscripted subscript. This would be the case when an access is of the form `A(B(i))`, where i is the loop index and B is the subscript array. Since the compiler cannot determine the values held in the B array, a cross-iteration data dependence is conservatively assumed. In a runtime test, such a loop is speculatively run in parallel, and all accesses to the A array are tracked. After the loop completes, it is determined from the gathered information whether any cross-iteration dependencies exist. If there are dependencies, the loop is re-executed sequentially. To ensure correctness, all variables that may be modified must be copied prior to the speculative execution of the parallel loop and restored if the loop is found not to be parallel.

The obvious downside of such a runtime test is the overhead it introduces into the program. All of the variables that may be modified must be copied prior to loop execution, variables which may have dependencies must be tracked as the loop executes, and the gathered information must be evaluated when the

loop completes. If the loop is found to have cross-iteration dependencies, the overheads are compounded by the need to then restore all modified variables and to re-execute the loop sequentially. Fortunately, the subscript array may remain constant throughout the program or may only change infrequently. This phenomenon is called *schedule reuse* [Law96] and may be exploited to minimize the incurred overheads. The dependence test need only be applied when the values in the subscript array are modified, otherwise the previously determined classification, parallel or not parallel, is still valid.

An example of a loop that cannot be statically parallelized by Polaris is the MXMULT_do10 loop in the Perfect Benchmark DYFESM. The loop is not parallelized due the array MX being accessed with a subscripted subscript. The loop, however, can be executed correctly in parallel. Thus, if a runtime test is applied to this loop, it would show the loop to be parallel. It can also be noted that the subscript array, responsible for the access pattern, remains constant throughout the program execution. Thus, it would suffice to perform a runtime test on this loop once, verifying that it could be run in parallel, and then to simply execute it in parallel thereafter.

Runtime data dependence testing fits well into our dynamically adaptive scheme. The optimization to be performed is loop parallelization. The executing program must decide if this optimization is beneficial for a given loop (i.e. the loop is in fact parallel). This is done using binary selection, classifying the loop as parallel or not parallel with a runtime data dependence test. The classification is then re-evaluated when the conditions determining its applicability change (whenever the subscript array is modified.)

In our experiment, we hand modified DYFESM to incorporate this scheme. After the test loop is executed, a TEST flag is set to FALSE. At each point in the program where the subscript array is modified, this TEST flag is set to TRUE. The next time the MXMULT_do10 loop is executed, the classification of the loop as parallel or serial is still valid if TEST is still FALSE. If TEST is TRUE the dependence test must again be performed.

We executed the hand modified version of DYFESM on 4 processors of an UltraSPARC Enterprise. We also ran a version that always performed the runtime test, and one that ran MXMULT_do10 in parallel without testing it. For all versions we expressed the parallelism using the OpenMP API. Figure 2 shows that none of the versions of the loop approach a linear speedup of 4. The overhead is due to a critical section that must enclose the reduction statements in the parallel versions (reductions are statements of the form $A = A + \dots$). Performing the runtime test improves over the sequential version by 16%. When a dynamic approach is used, and retesting is only done when necessary, an improvement of 60% is seen. The fixed parallel version, having advanced knowledge of the loop's data independence, shows only a slight improvement over the dynamically adaptive version. Therefore, the dynamically adaptive program is able to perform within 5% of the optimal statically optimized program.

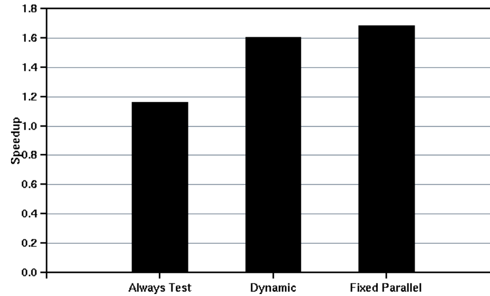

Fig. 2. The speedup, on 4 processors of an UltraSPARC Enterprise, of several versions of the MXMULT_do10 loop from DYFESM. The *Always Test* version refers to the program with the runtime data dependence test applied in each execution of the loop. The *Dynamic* version retests the loop only when the subscript array is modified. The *Fixed Parallel* version refers to a parallel version with no testing performed. It assumes that the loop is always parallel.

4 Case Study 2: Tiling

An optimization commonly used to enhance temporal locality in parallel loop nests is tiling. A loop is tiled by partitioning the iteration space into regions of a chosen size and shape known as *tiles*. All iterations within a tile are executed before a processor begins executing a new tile. If there is temporal reuse within a tile, the reused locations can remain resident in the processor cache between accesses. The original iteration order might have evicted the data item from the cache because of a large number of other accesses before the reuse.

A common example used to demonstrate the applicability of tiling is matrix multiplication. Figure 3 shows a matrix multiplication before and after tiling. In Figure 3.a, one can note that each element of B is reused in each iteration of the I loop. If the cache is large enough to hold the entire B array, then after the first iteration of the I loop, there will be no more cache misses on accesses to B. However, if the cache is not large enough, then the entire B array will need to be reloaded into the cache in each iteration of the I loop. In Figure 3.b, a tiled version of the matrix multiplication is shown. Now smaller tiles of the B array are operated on within each iteration of the I loop. By choosing a small enough tile size, one can ensure the portion of B used will remain in the cache between iterations of the I loop. If the tile size is too small, unnecessary overheads are introduced by the added iterations of the outer loops.

The difficulty arises in choosing the tile size. This decision requires that the size of the A, B and C matrices be known, and that the size of the cache is known as well. If one writes a program that is portable among different machines, the size of the cache cannot be assumed. The size of the arrays may also not be statically determinable. Thus a dynamic scheme must be used to determine the proper tile size.

```
DO I =  1,N,1
 DO K = 1,N,1
  DO J = 1,N,1
   C(J,I) = A(K,I)*B(J,K) + C(J,I)
  ENDDO
 ENDDO
ENDDO
```

(a) Untiled

```
DO KK = 1,N,BLK
 DO JJ = 1,N,BLK
  DO I =  1,N,1
   DO K = KK,min(kk+BLK-1,N),1
    DO J = JJ,min(jj+BLK-1,N),1
     C(J,I) = A(K,I)*B(J,K) + C(J,I)
    ENDDO
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

(b) Tiled

Fig. 3. Matrix multiplication: (a) original and (b) tiled into BLKxBLK sized tiles.

To demonstrate the applicability of adaptive programs to this problem, we hand modified a matrix multiplication kernel to automatically select a tile size using k-ary selection. The kernel performs 100 matrix multiplications on two 512×512 matrices. Each element in the matrices is a 4-byte real number. Our approach is to try several possible tile sizes and to choose the one that yields the minimum execution time. The first time that the multiplication is run, it is executed without tiling, and this time is recorded. After this, each execution of the multiplication is performed using a tile size of

$$\frac{N}{2^i} \times \frac{N}{2^i} \tag{1}$$

where i is the number of times the nest has already been executed.

Therefore, tile sizes of 256×256, 128×128, 64×64, ... are tried. This continues until a tile size of 16×16 is reached (it is assumed that smaller tile sizes would be inefficient on most machines). At each step, the tile size yielding the minimum time is recorded. After all tile sizes have been tried, the one producing the minimum time is used for each subsequent multiplication.

Iterative modification could also be used to select the tile size. This approach would begin with the untiled loop and, as with our k-ary selection approach, run with smaller tile sizes on each subsequent execution of the loop. The iterative modification scheme would, however, stop as soon as a configuration yielded a larger execution time than the previously used configuration. At this point, it could be assumed that the overhead associated with the smaller tile size had begun to offset its benefit. Unfortunately, this approach could lead to the selection of a local minimum and so we chose to use the less efficient k-ary selection technique.

We ran our hand modified kernel on both an UltraSPARC Enterprise with direct mapped 1 MByte external data caches and direct mapped 16 Kbyte internal data caches, and a SPARCstation 20 with direct mapped 256 Kbyte external caches and no internal data caches. On both machines we ran on 4 processors. We then ran the same kernel with the tile size fixed for various sizes, and also with the multiplication in its original untiled form. Figure 4 shows that the dynamic scheme shows performance close to the best fixed tile size on both machines. On

both machines, the dynamic scheme chose the correct tile size and was able to perform within 5% of the best statically optimized program.

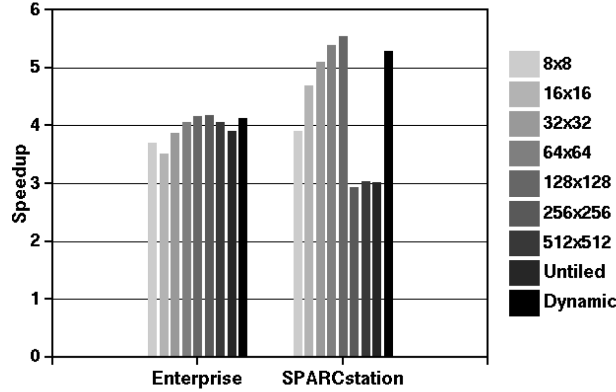

Fig. 4. The speedup of a 100 Matrix Multiplications.

5 Case Study 3: Dynamic Serialization

Even well-structured parallel applications, executed on current shared-memory machines, may run slower than their serial counterparts. These programs *slow down* if they contain parallel regions that cannot amortize the overheads associated with their parallel execution. It is therefore important to recognize situations in which the parallel execution of a code section would perform less than its original serial version, and to "undo" the parallelization. The overheads responsible for this poor performance include factors such as fork/join costs, communication overheads, and added memory latency for accessing remote data. All of these quantities are target-specific. In addition, the work performed by a program region depends on the input and may vary as the program runs. Thus, the impact of parallel overheads are a function of the program, the program input and the machine configuration. In a portable program, these parameters are only known at runtime. Hence, deciding whether a parallel code region will be profitable can benefit from a dynamically adaptive runtime scheme.

We added a pass to the Polaris parallelizing compiler to automatically generate programs that include a dynamic scheme for serialization [VE99]. Each loop is classified dynamically as serial or parallel using the state transition diagram shown in Figure 5. This scheme is a binary selection like our data dependence test. The Warmup and Test states in Figure 5 can be collapsed into the VERIFY state in Figure 1, and the Serial and Parallel states refer to the RUN state. 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.

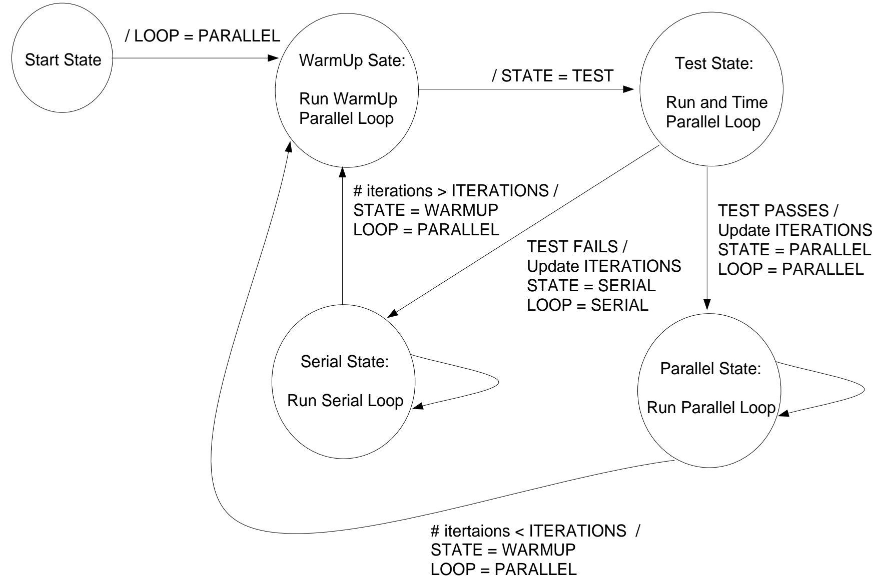

Fig. 5. The state transition diagram for dynamic serialization.

The test used to classify a loop as serial or parallel is based upon a sequential profile of the program on a *base machine*. The average per-iteration time of each loop is recorded on the base machine and is fed into the Polaris compiler when the adaptive program is generated. At runtime, a small kernel, embedded in the program, is run and timed to determine a scaling factor that is used to scale the profiled timings to the target machine. The test is to compare the per-iteration scaled sequential time to the measured parallel time. If the parallel time is longer, the loop is classified as Serial.

The model in Figure 5 is used 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.

To evaluate this technique for dynamic serialization, we first generated parallel versions of six benchmark programs from the Perfect and SPEC95 benchmarks suites: ARC2D, FLO52, MDG, HYDRO2D, SWIM and TOMCATV. These programs were generated by Polaris with the parallelism expressed us-

ing OpenMP. We then timed these original versions on 1 and 16 processors of an Origin 2000. We used the loop timings collected during these runs to generate a version of each program in which loops that executed more quickly on 1 processor were serialized. We view this profile-based program as the best possible statically optimized program using serialization to avoid overheads.

Next, we generated versions of each program using the dynamically adaptive scheme outlined in the previous section. The base timings used in the dynamic scheme were collected on a SPARCstation 20. This dynamic scheme, like the profile-based approach, uses a profile, but the profile is scaled and thus allows the program to be run on any machine, not just the machine on which the profiling was done.

We ran the various versions of each program on 16 processors of an Origin 2000. Figure 6 shows that the dynamic scheme was able to improve the performance in 4 of the 6 programs. The normalized execution times of the adaptive programs improved by as much as 85%, and on average 15%. The profile-based scheme likewise improved performance in 4 programs, and was able to improve performance by as much as 89%, and on average 15%.

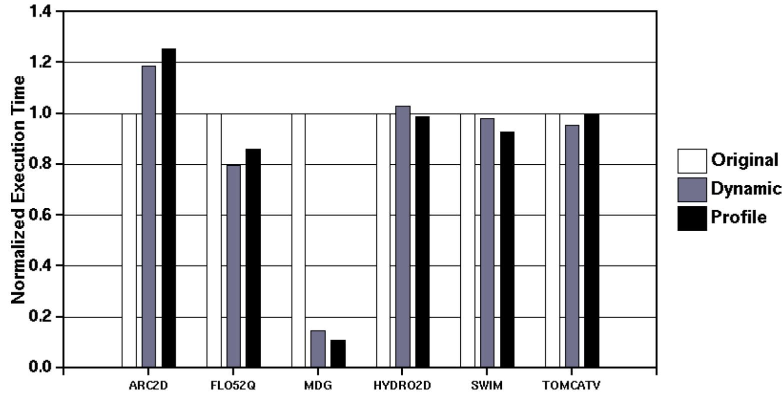

Fig. 6. The normalized execution time measured for several benchmark programs.

The noticeably poor performance of ARC2D must be explained. The ARC2D benchmark shows a large degradation in both the adaptive program and the profile-based program. This is due to the fact that in the profiles, there are loops which execute more quickly in the sequential version, however, serializing them in the parallel version causes degradation elsewhere in the application. This can be attributed to cache effects induced by the serialization. For example, if there are two consecutive parallel loops that access the same regions of an array, the first loop loads the data into the cache and thus the second loop sees no misses to this array. If the first loop were serialized, both loops would incur misses.

Again the dynamically adaptive programs outperform the original unoptimized programs in most cases. In addition, on average the optimal profile-based programs show no better performance than the dynamic scheme.

6 Related Work

Several techniques have been proposed to adapt a program dynamically to its input and/or runtime environment. In [BDH+87], Byler et al. develop multiversion loops for loop parallelization, in which a parallel and serial version are selected from at runtime. We use multiversion loops in our dynamic serialization scheme. In [GB95], Gupta and Bodik develop a framework for the runtime application of loop fission, loop fusion, loop alignment and loop reversal. Gupta and Bodik focus on modifying parameters to alter the execution of code sections and do not generate multiversions of code. We use a similar technique when we perform tiling dynamically. Such an approach is useful when generating multiversions is infeasible or impractical.

Several techniques have been developed which apply dynamic optimization to specific optimization problems. Saavedra and Park dynamically determine the amount of software prefetching and distance of the prefetch instructions to improve the performance of parallel programs on networks of workstations [SP96]. Hall and Martonosi propose a system that dynamically adjusts the number of processors used by compiler-parallelized programs to improve throughput [HM97]. In [DR97], Diniz and Rinard use *dynamic feedback* to automatically select the best synchronization policy for parallel object-based programs. Diniz and Rinard briefly outline the idea of dynamic feedback as a generic approach to optimization. They use alternating *sampling* and *production* phases of user defined lengths. In contrast, our approach re-evaluates configurations automatically as the environmental factors effecting previous decisions change.

In parallel programming, the technique most often thought of when discussing runtime techniques, however, is runtime data dependence testing. In [SMC91,RP94,RP95], runtime tests are performed to uncover parallelism undetectable at compile-time. The authors discuss *schedule reuse*, a phenomenon which can be exploited to reduce the number of times a test need be applied. We apply these techniques in our runtime data dependence example, and show that they fit well into our dynamic optimization framework.

The schemes discussed above are all forms of dynamic optimization. Our goal, however, is not to only solve a particular problem, as these schemes did, but to develop a generic scheme for dynamic optimization and to show its applicability to a wide range of problems. Thus, we aim at combining and integrating the many techniques discussed in the literature cited above, as well as new techniques, into one generic framework.

7 Conclusion

We have shown dynamically adaptive programs to be an effective means of applying optimizations when the information required to determine their usefulness, or correctness, is unavailable until runtime. We have discussed the key features of such programs and issues that must be addressed for efficient implementations. We described the *binary selection*, *k-ary selection* and *iterative modification* approaches.

We demonstrated that using a dynamic approach to remove redundant runtime data dependence tests can yield a decrease in execution time of 38% compared to always performing the tests. This was within 5% of the performance of a program with the tested loop explicitly parallelized. In a second experiment, the tile size was automatically selected in a matrix multiplication kernel. The resulting code showed improvements as large as 75% over the untiled code, and was within 5% of the best statically tiled version. Third, we presented a dynamic serialization scheme as automatically applied by the Polaris compiler. It showed decreases in parallel execution time as large as 85%. On average, a profile-based approach showed no larger gain.

Dynamically adaptive programs is a new paradigm that promises to overcome one of the most severe limitations of optimizing compilers: their dependence on the availability of compile-time information. Our general framework will make it possible to defer optimization decisions until runtime, inspect the state of the program and the machine environment, use timers and hardware monitors to evaluate optimizations, and use multi-version code, parameterized code, or dynamic recompilation for runtime tuning. This may not only yield truly performance-portable programs but also lead to a new generation of optimizing compilers.

References

[BDE+96] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, pages 78–87, December 1996.

[BDH+87] M. Byler, J.R.B. Davies, C. Huson, B. Leasure, and M. Wolfe. Multiple version loops. In *International Conf. on Parallel Processing*, pages 312–318, August 1987.

[BEF+96] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Lee, T. Lawrence, J. Hoeflinger, D. Padua, Y. Paek, P. Petersen, B. Pottenger L. Rauchwerger, P. Tu, and S. Weatherford. Restructuring programs for high-speed computers with Polaris. In *1996 ICPP Workshop on Challenges for Parallel Processing*, pages 149–162, August 1996.

[DR97] Pedro Diniz and Matrin Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proc. of the ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation*, Las Vegas, NV, May 1997.

[FK97] Ian Foster and Carl Kesselmann. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, January 1997.

[GB95] Rajiv Gupta and Rastislav Bodik. Adaptive loop transformations for scientific programs. In *IEEE Symposium on Parallel and Distributed Processing*, pages 368–375, San Antonio, Texas, October 1995.

[HM97] Mary W. Hall and Margaret Martonosi. Adaptive parallelism in compiler-parallelized code. In *Proc. of the 2nd SUIF Compiler Workshop*, August 1997.

[KF98] Nirav H. Kapadia and José A.B. Fortes. On the Design of a Demand-Based Network-Computing System: The Purdue University Network Computing Hubs. In *Proc. of IEEE Symposium on High Performance Distributed Computing*, pages 71–80, Chicago, IL, 1998.

[Law96] Thomas Lawrence. Implementation of Run Time Techniques in the Polaris Fortran Restructurer. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1996.

[LLM88] M. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proc. of the 8th Int'l Conf. of Distributed Computing Systems*, pages 104–111, June 1988.

[RP94] Lawrence Rauchwerger and David Padua. The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization . *Proceedings of the 8th ACM International Conference on Supercomputing, Manchester, England*, pages 33–43, July 1994.

[RP95] 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 1995.

[SMC91] J. Saltz, R. Mirchandaney, and K. Crowley. Run time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5), May 1991.

[SP96] R. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *Proc. of the 1996 Conf. on Parallel Algorithms and Compilation Techniques*, Boston, MA, October 1996.

[VE99] Michael J. Voss and Rudolf Eigenmann. Reducing parallel overheads through dynamic serialization. In *Proc. of the 2nd Merged Symposium IPPS/SPDP: 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (to appear)*, San Juan, Puerto Rico, April 1999.