ADAPT: Automated De-Coupled Adaptive Program Transformation

Michael J. Voss and Rudolf Eigenmann
Purdue University
School of Electrical and Computer Engineering
{mjvoss,eigenman}@ecn.purdue.edu

Abstract

Dynamic program optimization offers performance improvements far beyond those possible with traditional compile-time optimization [1, 2, 3, 4]. These gains are due to the ability to exploit both architectural and input data set characteristics that are unknown prior to execution time. In this paper, we propose a novel framework for dynamic program optimization, ADAPT (Automated De-coupled Adaptive Program Transformation), that builds on the strengths of existing approaches. The key to our framework is the de-coupling of the dynamic compilation of new code variants from the dynamic selection of these variants at their points of use. This allows code generation to occur concurrently with program execution, removing dynamic compilation overheads from the critical path. We present a compilation system, based on the Polaris optimizing compiler [5], that automatically applies this framework to general "plugged-in" optimization techniques. We evaluate our system on three programs from the SPEC floating point benchmark suite by dynamically applying loop distribution, loop unrolling, loop tiling and automatic parallelization. We show that our techniques can improve performance by as much as 70% over statically optimized code.

1 Introduction

Compile-time optimization is often limited by a lack of information about the program input and/or target architecture. It is not uncommon that a program is installed on a network file system server, and is run on machines of varying configurations. Some of these machines may be uniprocessors, some may be multiprocessors, and each machine may have different processor speeds and memory hierarchies of differing depths and sizes. Even if a program is compiled for a single machine, optimization decisions are still limited by the lack of information about the input data set. Whether a technique should be applied, and if so with what parameters, can often not be determined at compile-time due to these unknown quantities. Rather, a solution at runtime is necessary.

We propose a novel framework for dynamic program optimization that builds on the strengths of existing approaches. Automated de-coupled adaptive program transformation (ADAPT) separates the generation of new code variants from their use. A translator runs concurrently with the application and, using values obtained through compiler-inserted instrumentation, optimizes code sections in the context of the current input data and machine configuration. Because we overlap code generation with program execution, standard compilers can take the place of the highly specialized code generators of many dynamic compilation systems.

As new variants are created, they are added to those available to the *dynamic selection* mechanism, which is the runtime decision algorithm that chooses the best code variant to run. Our dynamic selection framework allows for *program sampling* similar to the dynamic feedback approach [2]. That is, at runtime, program variants are timed and the best one is chosen. In addition, our scheme can prune the search space of program variants through conditions defined by the user, i.e. the developer of the actual optimization techniques. Using the instrumentation that is added to the application, ADAPT monitors the environment to ensure that decisions based on sampling are, and remain, valid.

We have implemented the ADAPT scheme using the Polaris compiler infrastructure. It generates the framework for invoking one or several different compilers at runtime and for selecting from the resulting code variants. It also inserts the instrumentation that is necessary for monitoring critical program variables and the machine environment. Compiler developers can "plug in" new transformation techniques by describing, in the form of a C++ class, the compilation tool's command line, and (optional) conditions for guiding their runtime selection. Our ADAPT compiler translates programs fully automatically. No user interaction is necessary. The user specifies neither the code sections on which to operate nor the applicable transformations. The scheme works for both uniprocessors and multiprocessors.

We evaluate ADAPT by applying it to three SPEC floating-point benchmarks: Mgrid, Swim and Tomcatv. To each code, the implementation of our scheme in the Polaris/ADAPT compiler applies loop distribution, loop tiling, loop unrolling and automatic parallelization. We present results for ADAPT's performance when searching for the best combination of these techniques. For comparison, we also combine these techniques statically without using architectural and input data set knowledge. These results are collected on both a multiprocessor and uniprocessor system, and show that ADAPT can outperform the statically generated combinations by as much as 70%.

In Section 2, we present an overview of our framework. In Section 3, we describe our compiler-supported system for automatically applying this framework. In Section 4, we show experimental results. In Section 5, we discuss related work. In Section 6 we present our conclusions.

2 An Overview of ADAPT

Figure 1 shows the basic structure of the ADAPT framework, an extension of our previous work found in [6]. Optimization occurs at the granularity of Intervals, which are code sections with a single entry and single exit point, typically loop nests. Each interval is replaced in the original code by an `if-else` block that selects between a call to the Dynamic Selector and the default static version of the interval. The default static version is used during a *shelter period* if the execution time falls below a profitability threshold. The `if` test and the periodic re-evaluation are the only overheads incurred by a code section that cannot benefit from our scheme.

Environment Monitoring: In order to adapt to a runtime environment, it must be monitored. The Inspector contains a number of routines for collecting environmental and program characteristics. These characteristics can include timings, performance counter values, and even results from microbenchmarks. Calls to these routines are embedded in the Dynamic Selector and other support code. The information returned by the monitoring routines is stored into descriptors for the code variants, the intervals, or the machine configuration. The execution times collected by the Inspector are used to maintain and prioritize the Optimization Queue.

Code Triage: The Optimization Queue is a priority queue that orders the interval descriptors by execution time. Optimization is performed on hot-spots first, by choosing the most time-consuming interval from this queue. Hot-spot detection, along with the shelter period described above, make up the *triage services* provided by our system.

Dynamic Selection: If an interval has a sufficiently large execution time, it will call the Dynamic Selector. When the Dynamic Selector is called, it selects the best code variant based on the interval descriptor and the descriptors of the available code variants. By default, it selects the previous best variant, unless it has become stale. A code variant becomes stale after a user-set time interval. Developers may also specify additional criteria for determining staleness. For example, a developer may set a threshold for code variants that have loop tiling applied to them; they become stale if the number of array accesses fall below this threshold.

If the current code variant becomes stale, a new variant is selected using a combination of pruning and sampling mechanisms. The variants available to an interval for execution are first screened using conditions defined by the developer of the optimization. For example, a developer may choose to limit the selection of a parallelized variant such that it can only be selected if the application is executing on a multiprocessor. After the available variants are pruned, the remaining variants are sampled in order to select a best variant, that is each variant will be used once, and the one with the shortest execution time will be selected.

Our sampling phase monitors the data set. In our implementation, the loop bounds are recorded at the beginning of the sampling phase. Timings are only considered comparable if these bounds match. If the bounds change continually, ADAPT is forced to select variants somewhat arbitrarily. However, if the application has several phases, with constant loop bounds within but not across phases, ADAPT will restart sampling at the phase boundaries.

Dynamic Code Generation: ADAPT is different from many dynamic optimization schemes by its ability to generate new code variants at runtime. The Local Optimizer Thread runs concurrently with the application. It selects the most important interval from the Optimization Queue, and through a remote procedure call, activates the Remote Optimizer. It then

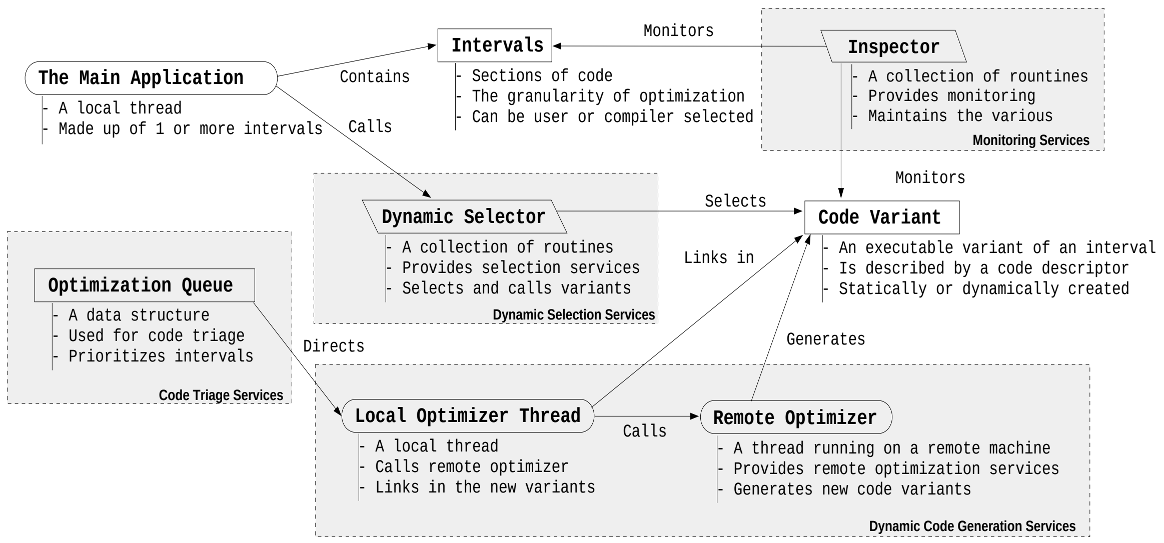

Intervals
Monitors
Inspector
The Main Application
Contains
- Sections of code
- The granularity of optimization
- Can be user or compiler selected
- A collection of rountines
- Provides monitoring
- Maintains the various
Monitoring Services
- A local thread
- Made up of 1 or more intervals
Calls
Monitors
Dynamic Selector
Selects
Code Variant
- A collection of routines
- Provides selection services
- Selects and calls variants
Links in
- An executable variant of an interval
- Is described by a code descriptor
- Statically or dynamically created
Dynamic Selection Services
Optimization Queue
- A data structure
- Used for code triage
- Prioritizes intervals
Code Triage Services
Directs
Generates
Local Optimizer Thread
Calls
Remote Optimizer
- A local thread
- Calls remote optimizer
- Links in the new variants
- A thread running on a remote machine
- Provides remote optimization services
- Generates new code variants
Dynamic Code Generation Services

Figure 1. Overview of the ADAPT optimization framework.

waits for the Remote Optimizer to return a new code descriptor, describing a newly generated code variant and its location. Upon receiving the code descriptor, it dynamically links in the new variant and adds it to those available to the Dynamic Selector. Finally it marks the interval's currently chosen variant as stale, and begins the process again with the current top of the Optimization Queue. If the Remote Optimizer returns an empty code descriptor, the interval is marked as fully optimized, removing it from the queue. The interval will be reconsidered if its current 'best' version becomes stale, allowing optimization to be performed in the context of the changed runtime environment.

The Remote Optimizer runs in the background on a remote machine, or a free processor of a multiprocessor, waiting for calls from the client application. It can generate a new code variant using any combination of restructurers and compilers, since it is not limited by the requirement for a short execution time. The Remote Optimizer is passed an interval descriptor, which includes information about the current runtime environment and the past behavior of the interval. It then generates a new version based upon this descriptor and the history of previous optimizations it has applied. After generating a new variant in a shared library, it creates a new code descriptor, updates the variant catalog file, and returns the code descriptor to the Local Optimizer.

3 Compiler-support for ADAPT

We have implemented comprehensive support for ADAPT in the Polaris optimizing compiler [5], which is a Fortran 77 source-to-source restructurer and parallelizer. Polaris/ADAPT automatically selects intervals, encapsulates them in newly created subroutines, and replaces them in the original code with an if-else block that conditionally calls the Dynamic Selector. It then generates the dynamic selector, inspector, and the local and remote optimizers. The framework applies developer-described stand-alone restructurers and back-end compiler flags.

A user description of a technique includes the following elements: (1) the number of different *sampling variants* and (2) the names of these variants. A sampling variant is simply an implementation of the technique. If multiple variants are supplied, Polaris/ADAPT will generate the framework such that the Dynamic Selector chooses the best code. This is the method by which a user turns on our sampling mechanism. If one of the sampling variants is specified as `OFF`, ADAPT will also sample the performance of the code without the technique applied. If no pruning criteria is provided by the developer, a technique is applied to all intervals, and variants using this technique can always be selected. All code variants become stale after a configurable time interval.

A user may optionally specify additional information to prune the sampling space, including: (1) a compile-time method for screening out intervals on which a technique should not be applied, (2) conditions under which the technique should be invoked, (3) conditions under which variants generated by this technique should be selected, and (4) conditions under which variants generated by this technique become stale. If multiple techniques are defined, all combinations that pass these screening criteria will be sampled.

```
class DistributionTechnique : public Technique
{
  public:
    void screen(ProgramUnit &pgm) {
      choose_imperfectly_nested_loops(pgm);
    }
    List<StringElem> *tools(String interval,
     ProgramUnit &pgm) {
      List<StringElem> *hold = new List<StringElem>();
      hold->ins_last(new StringElem(OFF));
      hold->ins_last(new StringElem("distributor "));
      return hold;
    }
    int num_tools(String interval, ProgramUnit &pgm)
      {return 2;}
};
```

Figure 2. User definition of a stand-alone loop distributor. This module shows the typical code a user must write to plug-in a transformation.

Figure 2 shows how loop distribution was described to our system. DistributionTechnique provides a compile-time method for screening intervals by defining the `screen` function to only select imperfectly nested loops. It then defines two sampling variants: `OFF` and `distributor`. Based on this definition, Polaris/ADAPT will generate the framework such that every imperfectly nested loop will have runtime variants generated with and without using loop distribution. At runtime, sampling determines the best of these variants.

4 Experimental Evaluation

4.1 Setup

Our evaluation is performed using three Sun SPARC machines: (1) an UltraSPARC Enterprise 4000 with six 250 MHz UltraSPARC-II processors, each with a 1 MB L2 cache, and a 1.5 GB shared main memory, (2) an UltraSPARC-II uniprocessor workstation with a single 300 MHz UltraSPARC-II processor, a 500 KB L2 cache, and a 1 GB main memory, and (3) a SPARCstation 20 with four 100 MHz hyperSPARC processors, each with a 256 KB external data cache, and a 128 MB shared main memory. Data is collected for both the Enterprise and the UltraSPARC workstation, with the SPARCstation 20 serving as a remote optimizer for the Enterprise.

We evaluate ADAPT using three programs from the SPEC floating point benchmark suite: Mgrid, Swim and Tomcatv. For both Mgrid and Swim, the SPEC'2000 versions are used, while for Tomcatv, the SPEC'95 version is used. In all cases, the ref data sets for the respective suites are used.

We evaluate the performance of our scheme in choosing the best combination of loop distribution, loop tiling, loop unrolling and automatic parallelization for these three programs. A loop distributor and a loop tiler were written using the Polaris infrastructure. Loop unrolling was performed by using the -unroll flag with the back-end compiler. Automatic parallelization was performed by using Polaris as a parallelizer. Table 1 contains a description of each technique and the conditions provided to our system.

For comparison, we generated a static version by combining each of these techniques, at compile-time, without input data set and architectural information provided. In addition, a static combination was generated using only the techniques that yielded a positive improvement when applied individually. The measurements of these two program variants are important reference points. They represent the performance of a static compilation and that of a feedback-directed compilation, respectively.

Figure 3 shows the results for both the Enterprise and UltraSPARC workstation. On the Enterprise, we collected timings for two configurations of ADAPT. The first configuration, RemoteDyn, performed the dynamic compilation remotely on the SPARCstation 20, and the second configuration, LocalDyn performed compilation locally on the Enterprise. On the UltraSPARC workstation, only a RemoteDyn scheme is present, with optimization performed on the Enterprise. All dynamically optimized programs start with the intervals unoptimized, that is, without loop distribution, loop unrolling, loop tiling and automatic parallelization applied.

4.2 Results

Mgrid: Using the SPEC'2000 ref data set, the sequential execution time of Mgrid on the Enterprise exceeds 1 hour. Many of the major loop nests in Mgrid have frequently changing loop bounds and so sampling is ineffective in this program. However as shown in Figure 3.a, ADAPT is still able to outperform both the static and profile-based compilations by over 70% on the Enterprise. This gain is due to the pruning mechanisms of ADAPT and the negative impact of inter-optimization effects in the statically compiled version. The LocalDyn version, in which the compilation is performed on a free processor of the much faster Enterprise machine, is able to slightly outperform the RemoteDyn version of Mgrid.

Both StaticCombo and ProfileCombo apply all of the techniques in Mgrid and show a significant degradation over automatic parallelization applied alone. The combination of loop distribution and automatic paral-

	Technique	Purpose	Static Screening	Runtime Screening	Stale Condition	Sampling Variants
LpDist	loop distribution	facilitates optimizations	imperfect nests only	none	none	OFF distributor
LpUnr	loop unrolling	increase ILP & decrease loop overheads	none	none	none	-unroll=0 -unroll=4 -unroll=8
LpTile	loop tiling	exploit temporal reuse	is tilable and has temp reuse	refs > cache	refs < cache	tiler
LpPar	loop-level parallelization	exploit parallelism	none	procs > 1	procs == 1	polaris

Table 1. Description of the various techniques added to our system. The symbols *refs* refers to the number of array references within the interval, *cache* refers to the machine's L2 cachesize, and *procs* refers to the number of processors available in the system.

lelization degrades performance by splitting some parallel nests into multiple parallel nests, thereby increasing fork/join overheads. In Mgrid, loop distribution uncovers no new opportunities for optimization and so only contributes this overhead. Unfortunately since sampling fails in Mgrid, and loop distribution is selected solely based upon sampling, ADAPT arbitrarily chooses the distributed version. A further decrease in execution time of over 1 minute could have been obtained had the non-distributed version been selected.

The combination of tiling and parallelization likewise degrades the performance of Mgrid on the Enterprise. Since the loop bounds are not known, tiling is applied to several nests in cases where the tile size exceeds the number of iterations of the loops being tiled. The compiler attempting to reduce the number of synchronization points, selects the outermost loop of these tiled nests as the parallel loop. In Mgrid, these tile control loops have only a single iteration and therefore the nest is essentially serialized. ADAPT, having both the iteration counts and the cache size available to it at runtime, does not tile these nests and therefore this degradation is not seen.

On the Enterprise, our framework begins with a sequential, unoptimized program, and is able to outperform both StaticCombo and ProfileCombo by over 70%. On the UltraSPARC uniprocessor workstation, the local optimizer thread is running on the same processor. However as shown in Figure 3.b, the ADAPT version of Mgrid is still able to perform to within 1% of the best static combination.

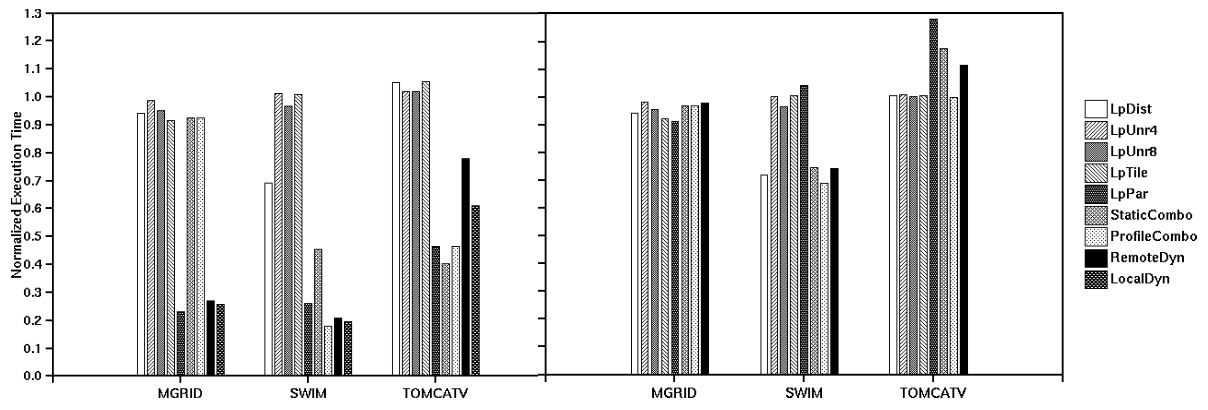
Swim: Using the SPEC'2000 ref data set, Swim likewise executes in over 1 hour on the Enterprise. Unlike Mgrid, it has runtime constant loop bounds and therefore allows sampling to converge. In Swim, again StaticCombo degrades over LpPar on the Enterprise due to conflicts between tiling and automatic parallelization. ProfileCombo applies only loop distribution and automatic parallelization. Unlike Mgrid, Swim benefits from this combination of distribution and parallelization; it enables loop interchange of a major loop nest. Therefore, a simple-minded heuristic of never applying loop distribution with parallelization is not sufficient. ADAPT shows performance near this profile-based static compilation, outperforming StaticCombo by 26%. In this case, ProfileCombo is exactly the correct combination of techniques and therefore represents a lower bound on the execution time.

ADAPT is able to begin with an unoptimized version of Swim and to perform to within 2% of this lower bound. On the uniprocessor workstation, ADAPT is able to slightly outperform StaticCombo and to perform within 5% of ProfileCombo.

Tomcatv: Our measurements of Tomcatv use the SPEC'95 ref data set. The sequential execution time of this code is significantly shorter than the other two programs, being less than 5 minutes on the Enterprise. While waiting for optimized code to be available, suboptimal variants are used instead. Because of its short execution time, the time spent in these suboptimal variants is a significant factor.

On the Enterprise, only parallelization improves performance when applied alone. Interestingly the blind combination of all techniques in StaticCombo outperforms ProfileCombo by 6%, highlighting the difficulty in selecting appropriate combinations. Due to its short execution time, both dynamic versions perform significantly worse than the static combinations. However, LocalDyn does perform much better than RemoteDyn.

Figure 4 shows the breakdown of execution time of each of the intervals in Swim and Tomcatv into their components. The intervals are ordered by the time that they are first dynamically compiled in the RemoteDyn version. In Swim, the T_{lag} overhead clearly dominates T_{fdbk}, showing that the unavailability of best variants due to de-coupling has a more significant impact on



(a) 4 Processors of an UltraSPARC Enterprise (b) A Uniprocessor UltraSPARC Workstation

Figure 3. The performance of optimizations on (a) the multiprocessor UltraSPARC Enterprise and (b) the uniprocessor UltraSPARC workstation. The individual techniques were applied at compile-time without architectural and data set information. The StaticCombo is the compile-time combination of all techniques. ProfileCombo is the combination of techniques that improved performance when applied individually at compile-time. RemoteDyn is the performance of ADAPT when dynamic compilation is performed remotely. On the Enterprise, RemoteDyn performs compilations on the SPARCstation 20. On the UltraSPARC workstation, RemoteDyn performs compilations on the Enterprise. LocalDyn is the performance of ADAPT when dynamic compilation is performed locally. The LocalDyn approach is only shown for the multiprocessor Enterprise. All times are normalized to the execution time of the unoptimized code.

performance than does sampling. The breakdown of the loops in Mgrid showed similar trends. In Tomcatv, which has a much shorter total execution time, the effect of T_{lag} is dramatic.

When compilation is done remotely on the slower 100 MHz hyperSPARC workstation, only 2 intervals are optimized before the program completes. The contribution of T_{lag} from just these two intervals accounts for 16% of the total program execution time. If the best variant of these two intervals had been used from the start of the execution, the program would be 16% faster. Due to the lag time, the majority of the code remains unoptimized. 70% of the execution time is spent in this part of the code.

When compilation is done locally, all of the significant loop nests in Tomcatv are able to be optimized. The short execution of the program still causes T_{lag} to be significant, being directly responsible for over 30% of the execution time. However, time spent in completely unoptimized code sections becomes negligible, and the total program execution time improves by over 15%.

On the uniprocessor, none of the techniques improve performance in Tomcatv. Therefore, only the overhead of the ADAPT framework is seen. This leads to a performance degradation with respect to the original code; however, this degradation is still an improvement over the loss seen by StaticCombo.

5 Related Work

One of the earliest methods proposed for performing runtime optimization was multiple version loops [7]. In this technique, several variants of a loop are generated at compile-time and the best version is selected based upon runtime information. Since multiversioning cannot make use of runtime information to prune useless variants, it may cause significant code explosion. Typically, to avoid such an explosion, only a few variants are generated for each code section.

Gupta and Bodik [8] proposed *adaptive loop transformations* to allow the application of many standard loop transformations at runtime using parameterization. They provide a framework for applying loop fusion, loop fission, loop interchange, loop alignment and loop reversal efficiently at runtime. We are currently adding support for parameterization to the ADAPT framework to increase its flexibility.

Diniz and Rindard [2] proposed *dynamic feedback*, a technique for dynamically selecting code variants based upon measured execution times. In their scheme, a program has alternating sampling and production phases. In the sampling phase, code variants, generated at compile-time using different optimization strategies, are executed and timed. This phase continues for a user-defined interval. After the interval expires, the code variant that exhibited the best execution time

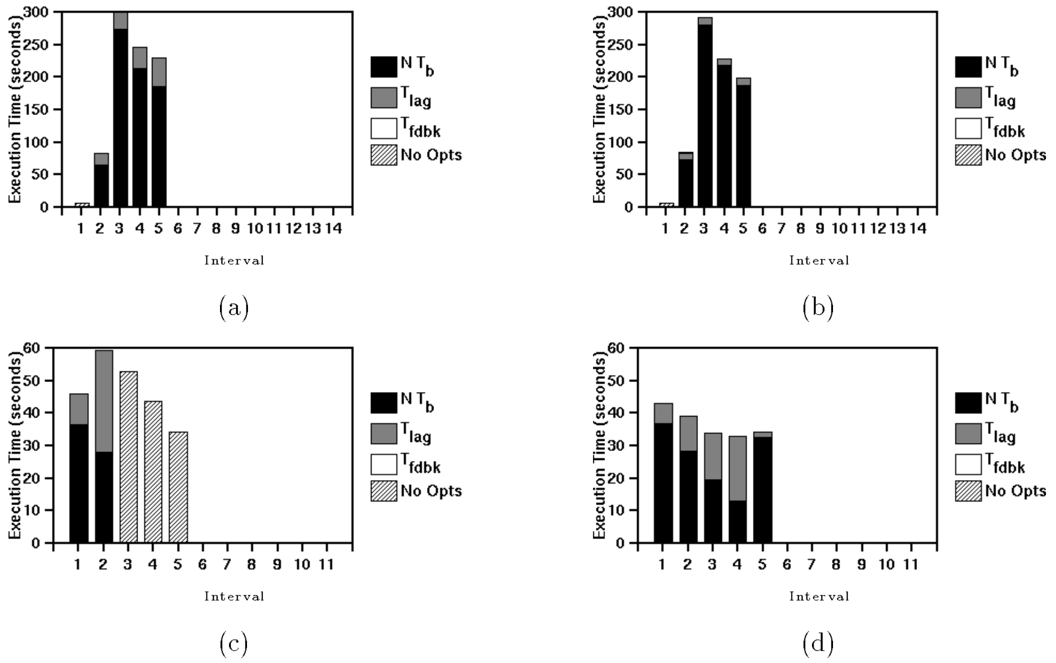


Figure 4. Interval execution times on 4 processors of the UltraSPARC Enterprise when performing optimization on the SPARCstation 20: (a) Swim RemoteDyn, (b) Swim LocalDyn, (c) Tomcatv RemoteDyn and (d) Tomcatv LocalDyn. NT_b is the time that the loop would take if the best generated variant had been used during each execution of the loop. T_{lag} is the overhead due to executing suboptimal variants because the best generated variant was not yet available. T_{fdbk} is the overhead due to executing suboptimal variants during sampling phases while the best variant was available. *NoOpts* refers to time spent in completely unoptimized code.

during the sampling phase is used during the production phase. ADAPT, unlike dynamic feedback, tracks loop bounds ensuring that all comparisons are, and remain, valid.

A dynamic technique often discussed in relation to parallel processing is runtime data dependence testing. In [9, 10, 3], runtime tests are performed to uncover parallelism undetectable at compile-time. One open issue is to decide when and where to apply such tests since the overheads are often large. Runtime dependence testing can easily be incorporated as a technique into the Polaris/ADAPT infrastructure. ADAPT could then dynamically determine where to apply the technique profitably.

Much work has also been done on dynamic compilation and code generation [11, 12, 13, 1, 14, 15, 16]. This work has primarily focused on efficient runtime generation and specialization of code sections that are identified through user-inserted code or directives. To reduce the time spent in code generation, optimizations are usually staged by using compilers that are specialized to the part of the program being optimized [11]. We attempt to minimize the need for specialized dynamic compilers by removing code generation from the critical execution path of a program.

Plezbert and Cytron [17] have proposed *continuous compilation* to overlap the "just-in-time" compilation of Java applications with their interpretation. Compilation occurs in the background as the program continues to be executed through interpretation. They order the code section to be compiled by targeting hot-spots first. This is also the approach taken by the Java HotSpot Performance Engine [18]. Unlike our approach, there is no specialization performed using machine or input data set information. We have shown that it is possible to specialize using relatively constant runtime characteristics by compiling in the background.

6 Conclusions

Automated de-coupled adaptive program transformation (ADAPT) is a novel approach for dynamic program optimization that builds on the strengths of existing schemes. It de-couples runtime code generation from the dynamic selection of generated variants at their points of use. The dynamic selection mechanism in our scheme allows for sampling to be done, as in the dynamic feedback approach, as well as for a pruning of the sampling space. We have given an overview of our framework and have presented and evaluated a compiler-supported framework for applying it.

We have shown that ADAPT can improve performance by up to 70% over statically optimized code. Our framework performs best with applications that are regular, and that have long execution times. Applications that have frequently changing loop bounds may profit from techniques that do not use sampling as a selection mechanism. Programs that have short execution times may improve in performance if the applied optimizations have large impact, but the lag in the availability of optimized variants, due to de-coupling, can have a significant impact on the execution time.

Our framework substantially simplifies one of the most difficult tasks in traditional compiler design: the decision making of when and where to apply a transformation. ADAPT makes these decisions based upon runtime measurements (i.e., sampling) and optional user-supplied static and dynamic pruning criteria. ADAPT uses "plugged-in" optimization techniques, allowing developers to easily add new optimizations. We believe that our framework and comprehensive compiler support provides, for the first time, a practical method for using and experimenting with diverse dynamic program optimizations. It facilitates the combination of a wide variety of new and existing optimization techniques and the development of strategies for orchestrating them in an optimal way.

References

[1] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *Proc. of the SIGPLAN '96 Conf. on Program Language Design and Implementation*, May 1996.

[2] Pedro Diniz and Matrin Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proc. of the ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation*, May 1997.

[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] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algera software. In *SC'98: High Performance Networking and Computing Conference*, November 1998.

[5] 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. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[6] Michael J. Voss and Rudolf Eigenmann. A framework for remote dynamic program optimization. In *Proceedings of Dynamo'00: ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, January 2000.

[7] 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.

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

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

[10] 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*, pages 33–43, July 1994.

[11] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proc. of the SIGPLAN '99 Conf. on Program Language Design and Implementation*, May 1999.

[12] Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental run-time specialization for free. In *Proc. of the SIGPLAN '99 Conf. on Program Language Design and Implementation*, May 1999.

[13] Massimiliano Polettto, Wilson C Hsieh, Dawson R Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.

[14] Charles Consel and Francois Noel. A general approach for run-time specialization and its application to C. In *Proc. of the SIGPLAN '96 Conf. on Principles of Programming Languages*, January 1996.

[15] D. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proc. of the SIGPLAN '96 Conf. on Program Language Design and Implementation*, May 1996.

[16] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proc. of the SIGPLAN '96 Conf. on Program Language Design and Implementation*, May 1996.

[17] Michael P. Plezbert and Ron K. Cytron. Does "just in time" = "better late than never"? In *Proc. of the ACM SIGPLAN-SIGACT '97 Symposium on Principles of Programming Languages*, January 1997.

[18] Sun Microsystems. The Java HotSpot Performance Engine Architecture. Technical White Paper, http://java.sun.com/products/hotspot/whitepaper.html, April 1999.