A Framework for Remote Dynamic Program Optimization

Michael J. Voss and Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University

ABSTRACT

Dynamic program optimization allows programs to be generated that are highly tuned for a given environment and input data set. Optimization techniques can be applied and re-applied as program and machine characteristics are discovered and change. In most dynamic optimization and compilation frameworks, the time spent in code generation and optimization must be minimized since it is directly reflected in the total program execution time. We propose a generic framework for remote dynamic program optimization that mitigates this need. A local optimizer thread monitors the program as it executes and selects program sections that should be optimized. An optimizer, running on a remote machine or a free processor of a multiprocessor, is then called to actually perform the optimization and generate a new code variant for the section. A dynamic selector is used to select the most appropriate code variant for each code interval based upon the current runtime environment. We describe this framework in detail and present an example of its use on a simple application. We show that our framework, when used with changing input, can outperform the best statically optimized version of the application.

1. INTRODUCTION

Dynamic program optimization allows information that is unavailable at compile-time to be used to improve program performance. This information can be dependent on the input data set, the current machine environment or a combination of both. Since optimization is done at runtime, re-optimization and re-compilation can likewise be done as critical values change during the course of execution. If done effectively, dynamic schemes offer improvements far beyond those possible by traditional static compilation methods.

Because most dynamic optimization methods pause program execution as a new version of the code is generated, the time spent in optimization and compilation must be minimized. Therefore, optimization is usually restricted to (1) selecting among several statically generated versions, (2) to using code that is parameterized, or (3) to generating code that has been staged before execution. Staging refers to the creation of code generators that have been specialized for the code section to be optimized and optimization to be applied. There are strengths and weakness to each of these techniques.

(a)
```
IF (N<M) THEN
 DO I = 1,N
  DO J = 1,M
   ...
  ENDDO
 ENDDO
ELSE
 DO J = 1,M
  DO I = 1,N
   ...
  ENDDO
 ENDDO
ENDIF
```

(b)
```
IF (N<M) THEN
 U1 = N
 U2 = M
 I -> I1
 J -> I2
ELSE
 U1 = M
 U2 = N
 I -> I2
 J -> I1
ENDIF
DO I1 = 1,U1
 DO I2 = 1,U2
  ...
 ENDDO
ENDDO
```

(c)
```
gen_nest1(fp,N,M)
(*fp)();
```

Figure 1: Loop interchange performed by: (a) selection of statically generated versions, (b) parameterization and (c) dynamic compilation. In (c), gen_nest1 generates new code that is interchanged if $N < M$.

Selecting from statically generated versions typically incurs very little overhead [2; 4]. However, these versions are created without any runtime information, and so cannot benefit from exact runtime values. Without any knowledge of which version might be needed, generating enough static versions to cover all possible needs may lead to significant code explosion. Parameterization is an alternative that reduces code explosion, but often at the cost of higher overhead [8; 16]. Although many techniques can be parameterized, not all program optimizations can be performed in this manner. Finally, dynamic compilation and code generation allows runtime value-specific specialization [3; 6; 7; 10; 12; 14]. This can create code highly tuned for the given input data set. However, overheads may be large, and so most practical systems restrict the optimizations to those that can be staged at compile-time [7]. Figure 1 shows an example of how *loop interchanging* can be applied using each of these three schemes.

We propose a framework that exploits the strengths of each of these approaches, as well as provides for truly flexible dynamic program optimization. This is done by performing the optimization on a remote machine or a free processor of

a multiprocessor. The code optimization, generation, and compilation can then occur concurrently with the execution of the application. This mitigates the need for minimizing the time spent in optimization since it is no longer directly seen in the application execution time. While a code section is being optimized, previously generated versions of it are used. Of course, the faster that a code section is optimized, the sooner that its optimized version can be used. Therefore, the previously discussed approaches for minimizing optimization time are still useful, however techniques need not be forced to use any one of these paradigms. In fact, new code can even be generated using a standard compiler, as our example application will show.

Unlike most dynamic compilation systems, we use information that may be stale by the time code is generated. We therefore target our system at regular applications and at optimization techniques that can use relatively constant environmental characteristics. For example, there are many loop-level optimizations that only require knowledge of iteration counts to accurately apply them. Often, compilers will make assumptions about iteration counts if they cannot be determined statically (i.e. the number of iterations is large). However, an inspection of a set of regular applications, the SPEC floating point benchmark suite, shows that loop bounds, although not compile-time constant, are relatively constant at runtime (see Figure 2). Therefore, code generated using the current loop bounds will likely be valid for at least a portion of the program execution that follows. It is this type of program characteristic that our framework can exploit. There have been others that have proposed overlapping runtime code generation with execution [13; 18], but they make no use of these relatively constant characteristics to specialize the code.

In Section 2 we give an overview of our framework and the services that it provides. In Section 3 we present an example application and show how three techniques can be applied to it using our approach. In Section 4 we discuss related work. In Section 5 we address open issues. Section 6 concludes the paper.

2. OVERVIEW OF OUR APPROACH

Our scheme optimizes code sections called intervals, which are code sections that have a single entry point and a single exit point, typically loop nests. Intervals are the static code sections that can be optimized in various ways at runtime, re-compiled, and dynamically linked into the executing application. Multiple versions of an interval may exist, and the variant most appropriate to the current runtime environment will be selected. In our approach, optimization and re-compilation is done on a remote machine or a free processor of a multiprocessor. Since this does not interrupt the execution of the application, we are able to use standard tools for the restructuring and compilation. In this section, we give an overview of the services that our framework provides as well as its implementation.

2.1 The Framework

Our framework provides services to (1) monitor the characteristics of the program and environment, (2) select code sections that will most benefit from the framework, (3) perform optimization and compilation on a remote machine, (4) dynamically select code variants most appropriate to the current runtime environment and (5) store and manage the generated code variants. We will briefly expand on each of these services.

2.1.1 Program and Environment Monitoring

In order to adapt to the runtime environment, it must be monitored. Our framework allows information to be collected for each interval as well as for each optimization variant of the interval. Basic machine characteristics may be collected as well. This information can include timings, performance counter values and even results from microbenchmarks.

2.1.2 Code Triage

For dynamic optimization to have the greatest impact, techniques must be applied to the right code sections. Based upon the average execution times collected for each interval, we separate intervals into those that may profit from dynamic optimization and those that will most likely be dominated by overheads. Those that have small execution times will be only intermittently monitored and will execute using their default versions. If their execution times later increase, they will again be considered for dynamic optimization. The selection of intervals to optimize is ordered by the total time spent in each interval, thereby targeting hotspots first.

2.1.3 Remote Optimization and Compilation

To minimize the overhead associated with the dynamic generation of code, we perform all re-compilation on a remote machine or a free processor of a multiprocessor. When an interval is selected for optimization, a remote procedure is called to perform the optimization. The remote optimizer will use information provided about the current runtime environment as well as the optimization history of that interval. A local optimizer thread will run concurrently with the application, calling the remote optimizer and dynamically linking in the new versions that it generates.

2.1.4 Dynamic Selection

A variant of each interval will be selected based upon the current runtime environment. Once a 'best' variant has been selected, it will be used until the environment changes in such a way that it becomes *stale*. The environmental changes that cause a variant to become stale depend on the optimization techniques that were applied to it. For example, a tiled loop may need to be re-evaluated if the number of iterations change, or if the program has migrated and the cache size of the new machine is different. In addition to environmental changes, the generation of a new code variant will force the application to consider the new variant as a possible alternative.

2.1.5 Code Management

Since we propose our method as a general scheme for dynamic optimization, we make no restrictions on the number or type of optimizations that can be applied. This means that a large number of code variants may need to be stored and managed. All code variants are stored in shared libraries on a shared file system. A catalog file is maintained that describes all of these variants. When a new variant is compiled by the remote optimizer, it is written to a new shared library and the catalog is updated. If the application is run again at a later time, the catalog can be read and the already existing variants can be linked into the application.

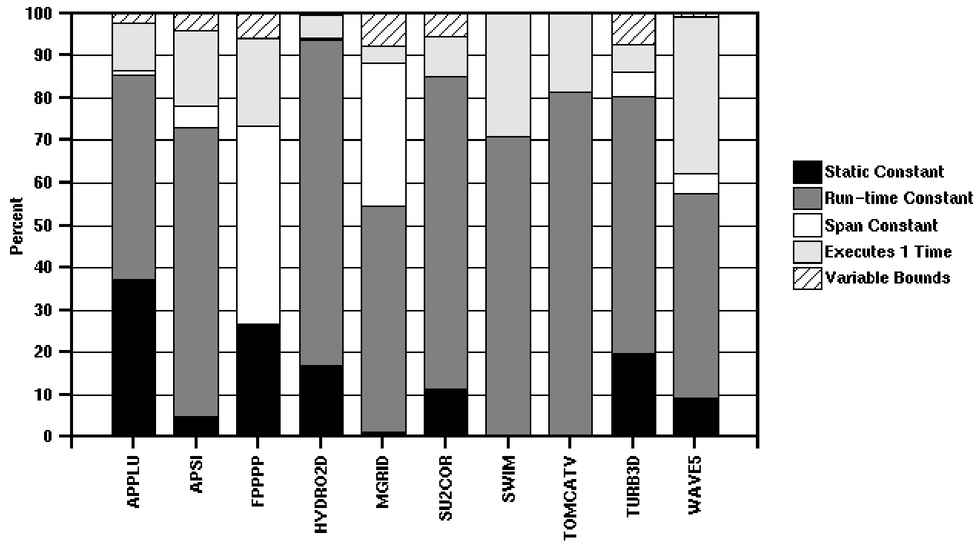

Figure 2: The breakdown of loop bounds in the SPEC CFP95 benchmarks. Each program is broken by percentage into loops that have statically known loop bounds, run-time constant loop bounds, bounds that are constant for more than 1 execution of the loop, bounds that are trivially constant since the loop executes only once, and bounds that are variable. On average over 70% of loops are not compile-time constant, but show some form of runtime constantness.

2.2 Implementation Mechanisms

Figure 3 shows the mechanisms employed by our framework. Optimization occurs at the granularity of intervals, which can be either user- or compiler-selected. 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. If the execution time of the interval is below the profitability threshold, a flag is set so that the default version will be selected, minimizing overheads. This flag will be reset after a configurable time interval. Currently the profitability threshold is also user-set, however we plan to develop a mechanism to automatically determine an appropriate value at runtime.

The behavior of each interval is monitored by the Inspector. The Inspector is a collection of routines used for collecting environmental and program characteristics. Calls to these routines are embedded in the dynamic selector and other support code. The information collected by these routines is stored into the code descriptors that describe each code variant, the interval descriptors that describe each interval, or the machine descriptor that describes the current machine configuration. Where the information is stored depends upon which routine is called.

Intervals that have sufficiently large execution times will call the dynamic selector to select and execute an appropriate code variant. The dynamic selector will select a code variant based upon the interval descriptor and the code descriptors for the available code variants. It will, by default, select the previously used variant unless this variant has become stale. Each code descriptor will provide the conditions under which its variant becomes stale, and the dynamic selector will check these conditions each time before it executes the code. If the code has become stale a new code variant will be selected.

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 waits for the Remote Optimizer to return a new code descriptor, describing the 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 Optimization Queue. The interval will be reactivated 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. These tools are 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.

The current implementation of the framework is written in C, but can be used with both C and Fortran applications. The Local Optimizer Thread is implemented using Solaris threads and the remote procedure calls are implemented using SunSoft ONC+ distributed services. All dynamic compilation is done using the standard Solaris C and Fortran compilers and dynamic linking is done with the standard Solaris link-editor and runtime linker. All of these libraries and utilities are standard on Solaris 2.6 machines. Unlike many other dynamic compilation frameworks, no specialized compilers or code generators are required.

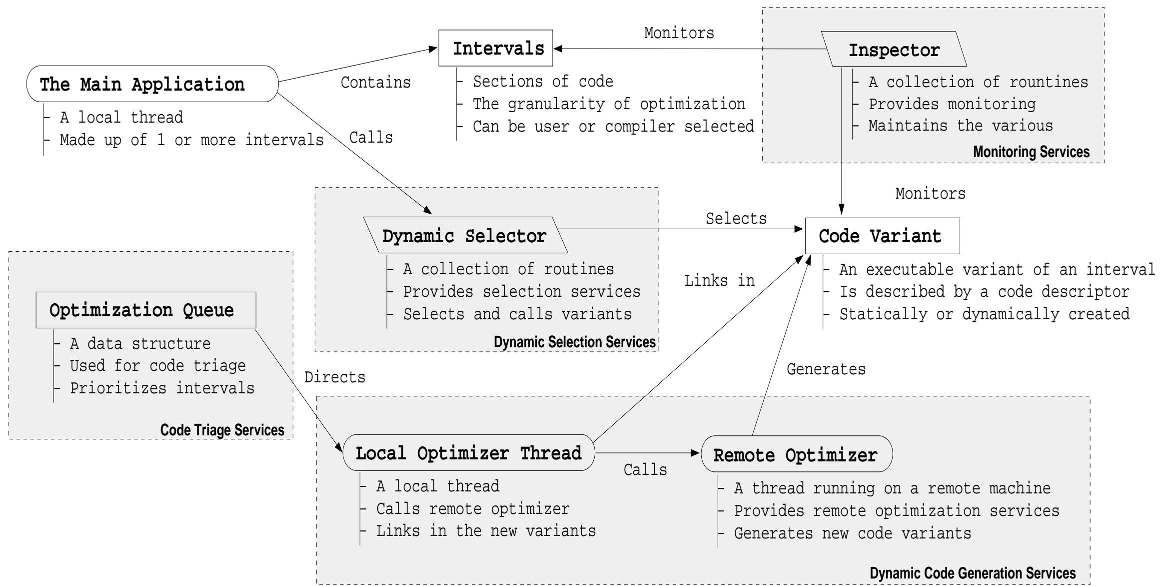

Figure 3: Overview of the optimization framework.

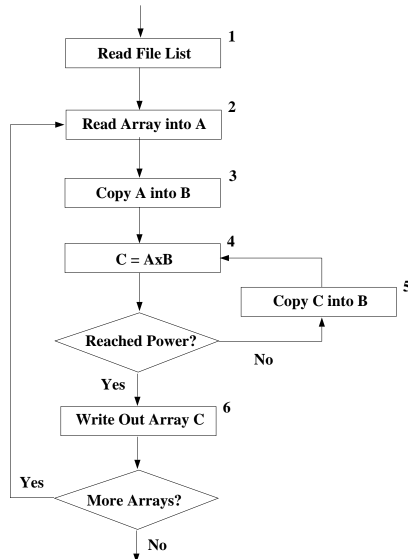

Figure 4: An overview of the example application. The numbered boxes refer to code blocks in the application. Blocks 2, 3, 4, 5 and 6 are selected as intervals for use with our dynamic scheme.

3. AN EXAMPLE APPLICATION

In order to better show the operation and usefulness of our framework, we apply it to the example application shown in Figure 4. The code begins by reading in a list of files that contain NxN matrices of various sizes. Each file also specifies the power, P, to which its matrix should be raised. The code then loops, reading in each array and multiplying it by itself P-1 times, writing the result to an output file. Given the scope of this paper, we felt that the use of an artificial benchmark would better expose the basic services we have introduced. In other work [19], we are performing a study of our framework on several of the SPEC'95 codes shown in Figure 2.

In this paper, we target our framework at finding the best combination of three optimization techniques: (1) loop tiling, (2) conditional reduction assignment and (3) loop unrolling. Loop tiling breaks the iterations of a loop into tiles of a given size and shape. All of the iterations of a tile are executed before another tile is begun. This technique is often used to move memory location reuses closer together in time, to reduce the likelihood of a data cache eviction before the reuse. Conditional reduction assignment (CRA) would replace the code fragment in Figure 5.a with the fragment shown in Figure 5.b. This can be used to greatly reduce the number of memory references and floating point computations when using sparse data. The final technique, loop unrolling, replicates a loop's body n times and adjusts the loop bounds accordingly. It is typically used to increase ILP and reduce looping overheads.

3.1 Applying the Framework

We manually applied the framework to the code in Figure 4. As discussed in Section 2, we encapsulated each interval in a subroutine and placed an `if-else` block at its original location. All code blocks were transformed in this way, except for Block 1, since it only executes once and therefore cannot profit from our scheme. A call to an initialization routine was placed at the beginning of the application and a call to a cleanup routine was placed at the end. The initialization routine creates and initializes the interval descriptors for each code section. It then reads the catalog file that is found in the current directory, creating code descriptors for each variant. Next, the machine's external cache size is determined by calling the Solaris `prtdiag` utility. This value is recorded in the machine descriptor. Finally the Local Optimizer Thread is created and started by a call to the Solaris threads library. The cleanup routine is used to kill this thread at the end of the program execution.

We developed and implemented heuristics for dynamically selecting the best code variant for each of the inter-

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

(a)

```
DO K = 1,N
 DO L = 1,N
  IF (A(L,K) .NE. 0) THEN
   DO J = 1,N
    IF(B(J,L) .NE. 0) C(J,K) = C(J,K) + A(L,K)*B(J,L)
   ENDDO
  ENDIF
 ENDDO
ENDDO
```

(b)

Figure 5: An example of conditional reduction asssignment: (a) the original code and (b) the transformed code. When the A or B matrices are sparse, the technique can lead to significant reductions in memory accesses and floating point operations.

vals. These heuristics are embedded in the corresponding Dynamic Selector subroutines. Blocks 2, 3, 5 and 6 need to determine whether to use the default statically optimized version or to use an unrolled variant. The Remote Optimizer was written to generate variants that were unrolled by factors of 2, 4 and 8. As each new variant is created, it is forced to run once. This creates a valid average execution time in its code descriptor. The heuristic then simply selects the variant with the smallest average execution time. The 'best' variant is set to become stale after a configurable time interval. If the current version goes stale, all of the variants have their average execution times invalidated and are again forced to run once. The new 'best' variant is selected from the updated average execution times. This method of choosing the most appropriate variant is similar in approach to Dynamic Feedback [4]. The Dynamic Selector subroutines are plug-in modules that allow experimentation with selection methods. Our framework does not force the use of any particular selection schemes and allows the easy addition of user-created methods.

Block 4 needs to select the best combination of loop tiling, conditional reduction assignment and loop unrolling. We place special restrictions on the versions that use CRA and/or loop tiling. Variants that use CRA are generated such that they return a denseness factor (DF). After the first CRA variant runs, it will write the DF to the interval's descriptor. If the DF is above 50% no other CRA variant will be generated or executed for this interval. Likewise, if the DF is below 50% no other non-CRA variants will be generated or run. If a CRA variant is selected as 'best', it will become stale after the configurable time interval or if the denseness factor rises above 50%. When any variant becomes stale it will invalidate all of that interval's variants' average execution times.

Whether a variant that uses tiling is generated and/or executed is based upon loop bounds. We want to exploit the reuse of the B matrix in Figure 5.a and so tile the J and L loops. Within the K loop there are approximately N^2 memory references. Therefore, the loops need to be tiled if and only if $N \geq \sqrt{C}$, where C is the machine's effective cache size. A tiled variant will be selected if this relation holds, and will become stale when the relation is no longer true. Again, any variant becomes stale after the user-specified time interval. Table 1 summarizes the heuristics used for each scheme.

The Remote Optimizer is responsible for deciding what optimizations should be applied to each interval that it is passed, and for generating the corresponding code variant. We staged the tiling and CRA optimizations by writing a source code generator that can generate the Fortran source for any combination of loop tiling and CRA applied to Block 4. Unrolling is applied by simply using the `-unroll` flag when calling the Sun f77 compiler. Each variant is compiled into a shared library by using the standard Sun f77 compiler. For each interval we statically compiled a default version that did not use any of the optimization techniques. These default versions were likewise stored in shared libraries and logged in the catalog file. Calls to the Remote Optimizer generate new versions based upon knowledge of the already existing variants and the current interval and machine characteristics.

3.2 Evaluation

We ran our example application on a six processor UltraSPARC Enterprise, a four processor SPARCstation 20 and a uniprocessor UltraSPARC workstation. On the multiprocessors, versions were run with compilation performed locally on a free processor, as well as remotely. Remote compilation was performed on the UltraSPARC Enterprise, except when the application was run on the Enterprise, in which case remote compilation was performed on the SPARCstation 20. The Enterprise has six 250 MHz UltraSPARC-II processors with 64 Kbyte internal data caches and 1 Mbyte unified external caches. The SPARCstation 20 has four 100 MHz HyperSPARC processors, each with a 256 Kbyte external cache. The UltraSPARC workstation has a single 300 MHz UltraSPARC-II processor with a 64 Kbyte internal data cache and a 500 KByte unified external cache. All of the caches are direct-mapped.

All libraries were shared through the network file system. We set both the stale and the reset time intervals to 600 seconds. These user-set time intervals are described in Section 2.2. We set the minimum execution time for profitability to 100 μs. Conservatively, we set the system to use tiling if the data set exceeded 25% of the external cache size (the cache size is determined at runtime through a system call).

We experimented with four data sets: (1) a 100x100 dense matrix raised to the 100th power, (2) a 512x512 sparse matrix raised to the 100th power, (3) a 512x512 dense matrix raised to the 100th power and (4) a data set which included the previous three data sets in succession. We found that loop unrolling had a negligible effect on the performance. The CRA optimization showed a large improvement on the sparse data set, but incurred large overheads on the dense matrices. Tiling showed improvements on all but the 100x100 data set.

Figures 6, 7 and 8 show the execution time of the four data sets on the Enterprise, the SPARCstation and the UltraSPARC workstation respectively. The execution time is given for our framework, as well as for four statically optimized variants: (1) the code with no optimizations applied,

	Tiling	Cond. Assign	Unrolling
SelectType	Iterations > Theshold	DF < Threshold	Minimum Time
StaleType	Iterations < Theshold	DF > Threshold	-
Theshold	$\sqrt{C}$	50	-

Table 1: Optimization Technique Parameters

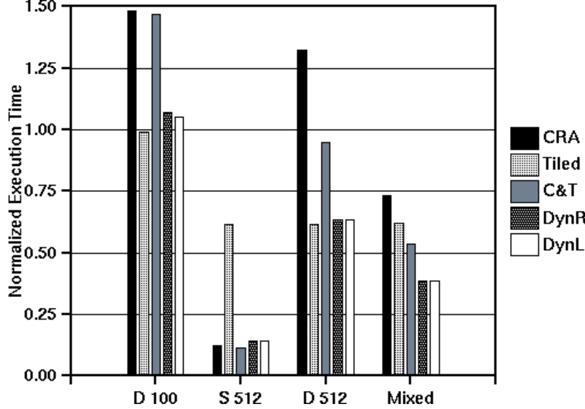

Figure 6: The performance of the application on the UltraSPARC Enterprise. *D 100* refers to the 100X100 dense data set, *D 512* refers to the 512x512 dense data set, *S 512* refers to the 512x512 sparse data set, and *Mixed* refers to the mixed data set. DynR is the dynamic scheme with compilation performed remotely on the SPARCstation 20. DynL is the dynamic scheme with compilation performed locally on the Enterprise. The other data points refer to statically applied optimizations.

(2) the code with CRA applied, (3) the tiled code and (4) the tiled code with CRA applied. Loop unrolling is not shown in the figures since its effect was negligible.

In the first three data sets, the dynamic framework was able to closely match the fastest of the statically optimized versions on each architecture, being always within 15% of its execution time. On each machine, the dynamic framework was able to outperform all of the statically optimized variants when the mixed data set was used.

The largest variations occurred in the Dense 512x512 data set. The CRA optimized variants showed large overheads when run on this data set on both the SPARCstation 20 and the UltraSPARC workstation. In the course of decision making, our framework must execute sub-optimal variants in order to determine the denseness factor as well as to compare execution times. In this case, the sub-optimal variants had much larger execution times than the best variant. Therefore, our framework was 11% and 15% slower than the best statically optimized variant on the SPARCstation and UltraSPARC workstation, respectively. The CRA optimization showed less overhead on the UltraSPARC Enterprise, and there our framework was within 1.5% of the best variant on this data set.

When the data sets are mixed, we see that our framework executes 8% faster on the SPARCstation and UltraSPARC workstation, and 15% faster on the UltraSPARC Enterprise than the fastest statically optimized version on each machine. It should be noted that the fastest statically optimized variant was not the same across the architectures.

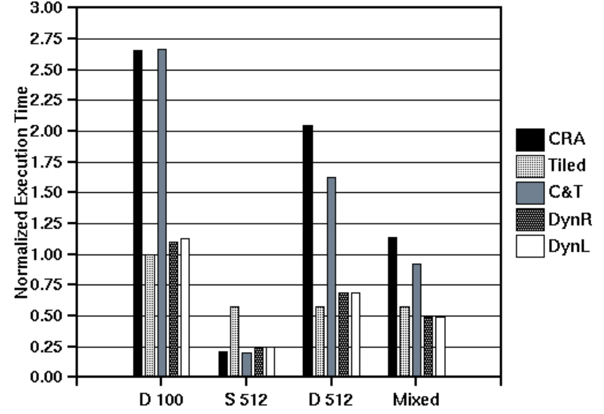

Figure 7: The performance of the application on the SPARCstation 20. *D 100* refers to the 100X100 dense data set, *D 512* refers to the 512x512 dense data set, *S 512* refers to the 512x512 sparse data set, and *Mixed* refers to the mixed data set. DynR is the dynamic scheme with compilation performed remotely on the UltraSPARC Enterprise. DynL is the dynamic scheme with compilation performed locally on the SPARCstation. The other data points refer to statically applied optimizations.

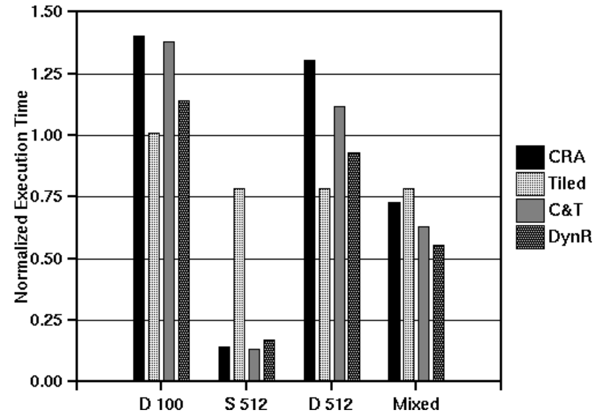

Figure 8: The performance of the application on the uniprocessor UltraSPARC workstation. *D 100* refers to the 100X100 dense data set, *D 512* refers to the 512x512 dense data set, *S 512* refers to the 512x512 sparse data set, and *Mixed* refers to the mixed data set. DynR is the dynamic scheme with compilation performed remotely on the UltraSPARC Enterprise. The other data points refer to statically applied optimizations.

Our framework was able to create a better variant for each machine by using the best set of optimizations during each phase of the execution.

Figure 9 shows the average improvement for each optimized variant across all data sets and machines. Although CRA showed huge improvements in the sparse data cases, on average it degraded performance by 11%. Tiling improved performance in all but the 100X100 data set, on average improving performance by 26%. When CRA was combined with Tiling it yielding on average only a 2% improvement. Both dynamic schemes were able to exceed all other optimized variants. With compilation performed remotely, the average improvement was 37% and when compilation was performed locally, on a free processor, the average improvement was 40%.

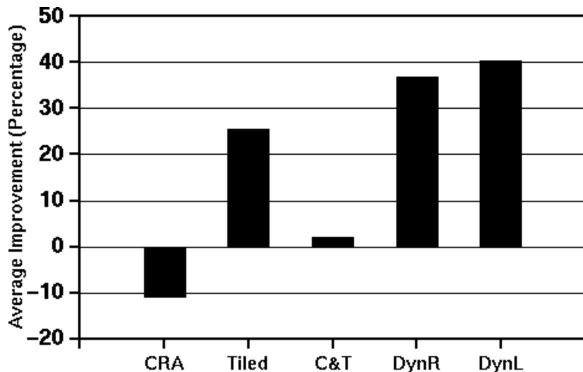

Figure 9: The average improvement across all machines and data sets. The values are the percentage of improvement over the original unoptimized code. DynR is the dynamic scheme with compilation performed remotely and DynL is the dynamic scheme with optimization performed locally.

The overheads incurred from our framework were negligible in this application. We found that on average less than 0.1% of the execution time was spent in the initialization and cleanup phase. Only 1% of the execution time was spent in dynamic code selection and maintenance. The remaining 99% of the time was spent in performing useful work.

4. RELATED WORK

One of the earliest methods for performing runtime optimization was to use multiple version loops [2]. In this technique, several variants of a loop are generated at compile-time and the best version is selected based upon runtime information. Many modern compilers still use this technique for selecting between serial and parallel variants. In our framework, the selection between a call to the Dynamic Selector and the use of the default static version is done through multiversioning.

Gupta and Bodik [8] proposed *adaptive loop transformations* to allow the application of many standard loop transformations at runtime using parameterization. They argue that the applicability and usefulness of many of these transformations cannot be determined at compile-time. Although they do not give criteria for selecting transformations based upon runtime information, they provide a framework for applying loop fusion, loop fission, loop interchange, loop alignment and loop reversal efficiently at runtime.

Diniz and Rindard [4] propose *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 during the sampling phase is used during the production phase. The heuristics we applied for selecting variants in Section 3 included a time interval as a means for determining staleness. This could be likened to a production phase. Our framework is more general in that it does not preclude the use of other selection schemes.

Like dynamic feedback, Saavedra and Park [16] propose *adaptive execution* for dynamically adapting program execution to changes in program and machine conditions. In addition to execution time, they use performance information collected from hardware monitors.

The approaches discussed above selected from previously generated code, or modified program execution through parameterization. Much work has also been done on dynamic compilation and code generation [7; 12; 14; 1; 3; 6; 10]. 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 [7].

We attempt to minimize the need for these specialized compilers by removing code generation from the critical path. Plezbert and Cytron [13] 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 also 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 of the program using machine or input data set information in these approaches. We have shown that it is possible to specialize using relatively constant runtime characteristics by compiling in the background.

5. OPEN ISSUES AND FUTURE WORK

Dynamic Selection Mechanisms

We have implemented only simple heuristics for selecting between code variants. Our framework can easily make use of more complex mechanisms. Accurate mechanisms for selecting between code variants, especially those generated using large combinations of optimization techniques, is still an open area of research. Our approach is general and does not preclude the use of emerging technologies such as machine learning techniques and genetic algorithms.

Retirement of Useless Code Variants

If runtime sampling is used to select a best code variant, that is the execution and timing of variants to determine the fastest executing code, then suboptimal code must be executed in order to make decisions. It was shown in Section 3.2 that the execution of suboptimal variants while sampling can degrade performance. It should be possible to lessen

the impact of "bad" variants, by maintaining a small runtime cache of variants with an LRU replacement policy. If the cache is full, and a new variant is generated, then the variant that has least recently been selected as a best variant will be evicted from the runtime cache. If later it is needed, it can be simply reloaded from the disk. We plan on implementing such a feature in future work.

A similar mechanism could be used for maintaining code variants on disk in order to save disk space. Variants that are not chosen as a best variant for a significant period of time could be removed from the disk, forcing them to be re-generated if they are needed in the future.

Parallelization versus Remote Compilation

Since our framework uses multiple processors and/or multiple machines, would it not be better to simply use the extra processor to execute a parallelized version of the program? The answer to this question depends on the application and the available environment.

Not all applications can be parallelized. In studies of the hand parallelization of regular scientific applications, it has been shown that typically only 1 out of 2 programs can be successfully parallelized [5]. In addition, it may take dynamic techniques just to determine the appropriate code sections to parallelize [20; 17; 15]. In these situations, the presented framework can make good use of parallel processors.

The available environment may also determine the benefits of remote optimization versus parallelization, and even whether both may be used. Parallel applications with fine-grain sharing may not benefit from execution on a loosely coupled system such as a network of workstations. In this case, remote optimization may offer benefits by improving the serial code. If a tightly-coupled multiprocessor is available then perhaps parallelization would be a better option. However, research on systems such as Condor [11] have shown that often in a large institution many workstations are idle at any given time. Therefore it seems that with a large amount of available resources, parallel execution and remote optimization can co-exist. Remote optimization could be used to tune the execution of even parallel codes, as is shown in [19].

The Use of NFS

In our framework, code variants are shared between the remote and local system by use of the network file system (NFS). This appears to limit the choice of remote nodes to those that share a file system with the local node. However, work is being done as part of the PUNCH project [9], Purdue University Network Computing Hub, to develop an internet filesystem that allows access to remote files in a fashion similar to that of NFS. Combined with such a system, our framework would be unlimited in its choice of remote nodes.

Automatic Application of the Framework

In other work, we are implementing comprehensive compiler support in the Polaris optimizing compiler for automatically applying this framework [19]. The compilation process requires no user interaction, unlike most dynamic compilation systems that require user-annotation of the code to be optimized. Initial performance results have been promising.

6. CONCLUSION

Dynamic program optimization can be an effective means for optimizing programs in the presence of information unavailable at compile-time. It allows programs to be continually tuned as program and machine characteristics are discovered or changed during the course of execution. Several techniques have been developed for minimizing the time spent in runtime optimization, since it is often directly reflected in the total program execution time.

We propose a framework for remote dynamic program optimization to mitigate the need for minimizing optimization times. A local thread chooses important code sections that are passed to a remote optimizer for optimization and recompilation. The optimization process is then able to proceed concurrently with program execution. New code sections, generated on the remote node, are dynamically linked into the executing application by the local optimizer thread. A Dynamic Selector is called as each code section is entered to select from the available code variants. The Dynamic Selector bases its decisions on the descriptions of the code variants and the current runtime environment.

In Section 2 we discussed the services provided by our framework and gave an overview of our implementation. In Section 3 we presented an example application and the application of our framework to select the best combination of three optimization techniques: loop unrolling, conditional reduction assignment and loop tiling. Our evaluation showed that the overheads associated with our framework were negligible, and that our approach provided an increase in performance of up to 15% over the best statically optimized version of our example application.

In future work, we will present an evaluation of our framework on a variety of techniques applied to more realistic benchmark applications. We believe that the application of our framework to regular programs, exploiting information that is constant for long periods during execution time, can lead to performance improvements that far exceed those possible from traditional static compilation methods.

7. ACKNOWLEDGEMENTS

This work was supported in part by DARPA contract #DABT63-95-C-0097 and NSF grants #9703180-CCR and #9872516-EIA. This work is not necessarily representative of the positions or policies of the U. S. Government.

8. 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 Programming Language Design and Implementation*, pages 149–159, Philedelphia, PA, May 1996.

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

[3] C. Consel and F. Noel. A general approach for runtime specialization and its application to C. In *Proc. of the SIGPLAN '96 Conf. on Principles of Programming Languages*, Jan. 1996.

[4] P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proc. of the*

ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation, pages 71–84, Las Vegas, NV, May 1997.

[5] R. Eigenmann, J. Hoeflinger, and D. Padua. On the Automatic Parallelization of the Perfect Benchmarks. *IEEE Transactions of Parallel and Distributed Systems*, pages 5–23, Jan. 1998.

[6] D. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proc. of the SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 160–170, Philedelphia, PA, May 1996.

[7] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proc. of the SIGPLAN '99 Conf. on Programming Language Design and Implementation*, pages 293–304, Atlanta, GA, May 1999.

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

[9] N. H. Kapadia and J. A. 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.

[10] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proc. of the SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 137–148, Philedelphia, PA, May 1996.

[11] 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.

[12] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proc. of the SIGPLAN '99 Conf. on Programming Language Design and Implementation*, pages 281–292, Atlanta, GA, May 1999.

[13] M. P. Plezbert and R. K. Cytron. Does "just in time" = "better late than never"? In *Proc. of the ACM SIGPLAN-SIGACT '97 Symposium on Principles of Programming Languages*, pages 120–131, Paris, France, Jan. 1997.

[14] M. Polettto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, Mar. 1999.

[15] 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*, pages 218–232, June 1995.

[16] 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, Oct. 1996.

[17] J. Saltz, R. Mirchandaney, and K. Crowley. Run time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

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

[19] M. J. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. Technical Report ECE-HPCLab-99209, Purdue University School of ECE, High-Performance Computing Lab, 1999.

[20] M. J. Voss and R. Eigenmann. Reducing parallel overheads through dynamic serialization. In *IPPS: 13th International Parallel Processing Symposium*, pages 88–92, San Juan, Puerto Rico, Apr. 1999.