TOWARDS A COMPILATION PARADIGM FOR COMPUTATIONAL APPLICATIONS ON THE INFORMATION POWER GRID*

Michael J. Voss and Rudolf Eigenmann†
Purdue University, USA

Abstract The Information Power Grid (IPG) refers to the world-wide infrastructure of computers and their interconnections. We are only at the very beginning of exploring applications and system software that can exploit these resources effectively and, at the same time, provide adequate interfaces to the programmer and end-user. In this paper we discuss compiler technology that serves this purpose. The core consists of methods and services that enable an application to adapt flexibly to the dynamically changing nature of the IPG. We present several applications that demonstrate performance opportunities in such scenarios. We then present and evaluate new compiler techniques and services that allow an application to be dynamically re-optimized as the execution environments change. One very important property of such system support is that the overheads of dynamic re-optimizations are kept small. We will show that this is the case for the presented techniques. The resulting applications can run close to the best performance that could be obtained with prior knowledge of the runtime situations.

Keywords: Information Power Grid, Adaptive Program Optimization, Dynamic Compilation

*Dedicated to John R. Rice on the occasion of his 65th birthday.
†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.

1. INTRODUCTION

The Information Power Grid (IPG) refers to the increasingly connected infrastructure of computers and world-wide communication networks. The vision of the IPG is similar to that of the electric power grid: an interconnected system of power sources are optimally managed by the facilities of the grid and present a user-friendly interface to the consumer. Similar to the electric power grid, the IPG would provide services such as load balancing of computers across the globe and routing of requests to the right resources. The user interface may be a "Web Appliance", plugged into a wall connector similar to today's electrical outlet in the office and at home.

From a computer system's perspective, the IPG is a complex computer architecture. It is parallel, distributed, and heterogeneous. It is also unreliable, because nodes and interconnections may fail or temporarily be unavailable. System support for the IPG will be distributed and relatively uncoordinated. Compilers, libraries and operating system versions may get upgraded in widely varying time steps and asynchronously.

Translating a computational application onto such a complex computer system is a tremendous challenge. While today the efficient compilation of a large computer application onto a homogeneous parallel processor is a grand challenge, the problems introduced by an IPG may seem unsurmountable. However, given the fact that the IPG is happening, tackling these problems appears very important. In this paper we are beginning to address the issues faced by a compiler when translating an application onto the information power grid. We will discuss interfaces to the underlying "IPG operating system", which facilitates advanced compiler operations. We will also present initial results of experiments that show the potential of new compilation technology as it deals with the dynamic, heterogeneous, and unpredictable nature of an architecture such as the IPG.

1.1. NEW COMPILATION TECHNOLOGY

At the core of the new compilation technology are capabilities that enable an application to dynamically adapt to changing parameters of architectures and environments, to newly available libraries, to compiler upgrades, and to unreliable system components. These capabilities need to overcome one of the most severe restrictions of today's compiler generation, which is the use of conservative assumptions. Conservative assumptions usually must be made because there is insufficient information about a program's input data and machine environment at compile time. Hence the compiler is not able to decide on the optimal program translation. Conservative assumptions prevent incorrect program transformations. They often lead to suboptimal performance even on today's homogeneous multiprocessor machines. On the IPG's drastically more dynamic machine structure, conservative compiler assumptions can have a devastating effect.

To avoid conservative assumptions, the compiler must aggressively defer optimization decisions to runtime. Some optimization decisions may even need to be deferred to after the computation has completed, that is, the optimization is applied speculatively, followed by a verification test. The specific new compiler capabilities that address this issue include

- monitoring techniques for changes in architecture, environments, tools, and program input data
- dynamic re-optimization techniques
- dynamic optimization decision support
- quality assurance support
- management support for program history and code versions.

In the following sections we will describe these capabilities.

1.1.1 An Illustrating Example. An engineer is making use of a device simulation program. The simulation is available on the IPG - for example the PUNCH (Purdue University Network Computing Hubs) system [10]. The IPG determines that a large multiprocessor system in Alaska is currently only lightly loaded and dispatches the simulation job to this machine. After the application starts, its monitoring module determines that, although the executable can run on the current processor architecture, it is only compiled for a single-processor system. While it begins the execution in this suboptimal way, it also spawns a new compilation task on the IPG, generating optimal code for the current machine and the present light load.

After two hours of execution the load changes and the IPG decides to migrate the application to a different machine. The new machine has the same basic architecture but a much larger cache. As a result, the dynamic optimization module readjusts cache optimization parameters. It does this by executing and timing several code variants obtained from the code management support. The code management support, in turn, provides the requested code variants from its database and, where necessary, creates additional code variants via remote dynamic compilation requests [18].

While the code executes, the monitoring module watches the parameters of the machine and the environment. After each significant change, it reevaluates the code optimization decisions. For example, a new compiler release may become available. The decision support chooses to recompile the most time-intensive part of the application with the new compiler's high optimization level. After the compilation job has completed, the application dynamically links the new code and executes it instead of the now stale code. It performs the execution under an exception domain with prior checkpointing, until it decides that the new compiler option is of assured quality.

1.1.2 Interfaces Compilers, Applications, and the IPG.

The example scenario makes use of several system software capabilities, located in the IPG's runtime system, the associated compilers, and the application itself. New interfaces provide for proper coordination of these three agents. For example, the compiler may insert code for dynamic optimization support directly into the application. At runtime, this code queries the IPG to learn about machine and environment parameters. If it decides that a new compilation variant for a section of code is necessary, it invokes the compiler through an appropriate link.

More specifically, the following interface between compiler, application and the runtime system are necessary. We are currently implementing these facilities in the Polaris optimizing compiler and the PUNCH network computing system.

- Monitoring interface. It allows the compiler and the applications to query machine and environment parameters, such as static machine properties (e.g., the cache size), dynamic characteristics (e.g., hardware monitors), and environment parameters (e.g., the current load of the network).

- Management support for program history. This allows the compiler and the application to retrieve and store data about past application performance and to coordinate with the IPG's scheduling intelligence.

- Remote dynamic compilation and code management support. This allows the running application to order the generation of new code variants and to retrieve such variants from a "cache".

- Quality assurance support. It provides support for checkpointing and exception domains, which allows a potentially unsafe code section to be executed in a controlled manner. It also provides interfaces to hardware error monitors, such as race detectors.

In a number of pilot studies we have implemented and tested several of these facilities. Section 2 describes experiments that show that dynamically adaptive optimizations outperform static optimizations and come close to the best performance that one would obtain with full prior knowledge of the runtime situation. Section 3 then describes several of the services introduced above, and evaluates their performance on a simple test program.

2. DYNAMICALLY ADAPTIVE PROGRAMS

In a number of case studies we have looked for answers to the question of how much improvement there can be from adaptively optimizing programs in changing machine environments.

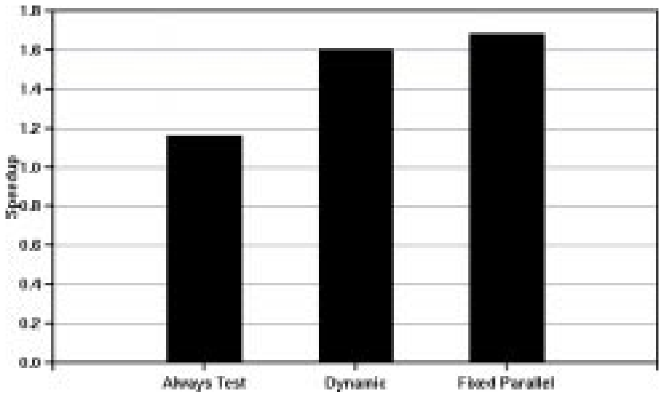


Figure 1 The speedup, on 4 processors of an UltraSPARC Enterprise, of several versions of the MXMULT_do10 loop in program DYFESM. The *Always Test* version refers to the program with a runtime data dependence test applied in each execution of the loop. This is representative of current compiler technology that makes static decisions on where to apply the test. The *Dynamic* version re-tests the loop only when the subscript array is modified, which is detected dynamically. The *Fixed Parallel* version refers to a parallel version with no testing performed, i.e., with prior knowledge that the loop will always be parallel. The results show that the dynamic version performs much better than the compile-time version and comes close to the variant that uses prior knowledge.

Figure 1 shows the results of an experiment with the most time-consuming loop `MXMULT_DO10` of the Perfect Benchmark `DYFESM`, in which we have applied a runtime data-dependence test in a dynamic way. Runtime data-dependence tests can potentially execute program sections in parallel, even if the parallelism cannot be proven at compile time. This is the case in the `MXMULT_DO10` loop. Significant performance gains can result from this optimization. However, current tests can have substantial overheads. To overcome this problem, our dynamic scheme checks if the environment in which the test is applied changes between two invocations of the code section. If not, then we know that the runtime data-dependence test would yield the same result as in the last invocation and therefore does not need to be re-applied. The relevant environment, in this case, is the value of an array used for indirect addressing in the loop. Our method tests dynamically for changes to this array. More details about the experiment of Figure 1 are given in [17].

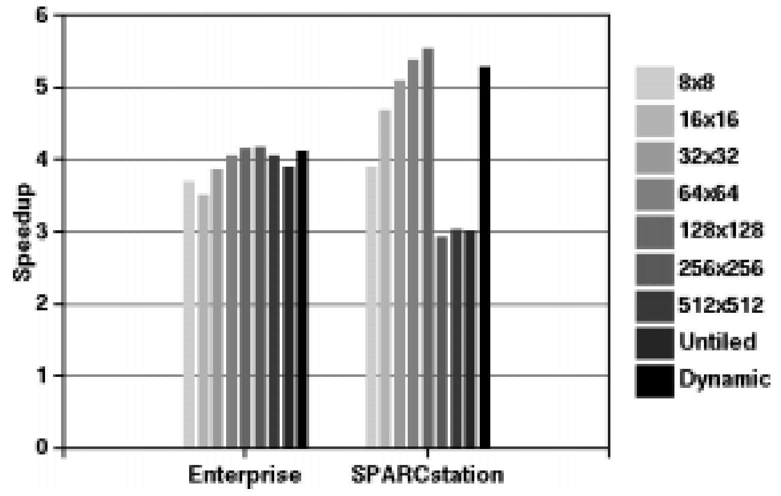


Figure 2 The speedup of an application performing matrix multiplications on two machines and using different tiling strategies. All but the "dynamic" bars represent fixed tile sizes. The dynamic scheme determines the best variant adaptively. The figure shows that the adaptive scheme comes close to the best tiling method on both machines.

Figure 2 shows another case study, in which we assume that an application is compiled without knowledge of the architecture's cache size. This is an important situation because a compiler may generate portable code that will later execute on several different machine configurations. Our goal is to apply the best program optimization for the eventual cache size nevertheless. The test program includes a sequence of 100 matrix multiplication operations. We applied a dynamic algorithm that tries and evaluates at runtime several code tiling variants to determine the best version. The figure compares the dynamic scheme with several fixed tiling parameters. Our method will not only determine the best tiling parameters for a given application as it is invoked on a new machine, but also adapt to a new cache size after the application has migrated to a new machine with different cache parameters. More details about this experiment are also given in [17].

The results of a third case study are shown in Figure 3. A dynamic scheme decides whether or not it is appropriate to execute a parallel loop on multiple processors. Small parallel loops incur a *parallelization over-*

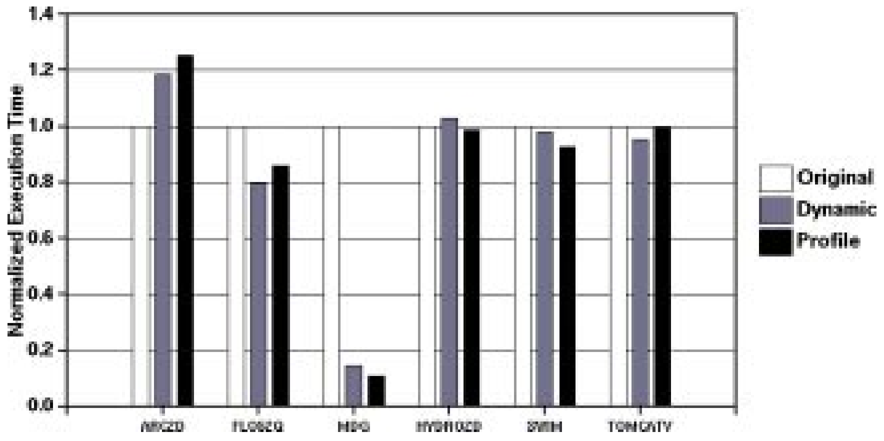


Figure 3 The effect of *dynamic serialization.* Several benchmarks are shown (1) with their original, normalized execution time, (2) with applied dynamic serialization, and (3) with serialization based on full knowledge of the best variant, obtained via profiling. On average, dynamic serialization achieves significant improvements, and is as good as the profile-based method.

head, which may offset the gain from parallel execution. This overhead is very machine-dependent. Our dynamic scheme makes decisions based on information from micro benchmarks, loop timings, and profiles from a reference machine. The results show that, in general, the dynamic scheme performs significantly better than the scheme in which all parallel loops are executed as such. It comes close to the profile-based version, which applies prior knowledge about which loops execute the fastest in which mode. The results also show an effect that needs to be given serious considerations in program optimizations that are performed on individual program sections: In the Arc2D code, several loops perform differently in their final context than in the context in which they were measured. Thus, a parallel loop may be measured as slower than in the fully serial program execution. However, when executed serially in the context of other parallel loops, its performance may degrade further. We attribute this primarily to cache affinity effects between adjacent loops. To address that problem, one needs to apply overall program optimization strategies, in addition to local optimizations. We have not yet done this in our current work. More details of this study are given in [16].

A fourth study is shown in Figure 4, in which our dynamic scheme simply "experiments" with compiler options. Starting from a code variant compiled with no additional flags, the adaptive scheme generates and evaluates program executions, each compiled with a different combination of flags. The dynamic selection scheme of the flags is generic, without any knowledge of the best combination or relationships between

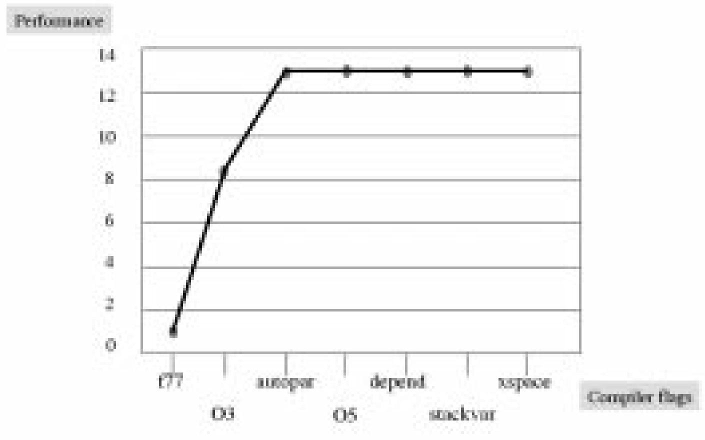


Figure 4 Dynamic selection of compiler flags. The best set of flags are automatically selected. The figure shows the performance of the SPEC95 Swim benchmark as it is executed with a progressive set of compilation flags. In this program relative to the original code with no compiler flags, the `-O3` and `-autopar` options have a significant effect, while other options have a minor performance impact.

compiler optimizations. While this method finds the best variant less quickly than an advanced selection scheme, it is important to note that, for long-running or repeated applications, even this basic scheme converges automatically to the best flag set.

3. SERVICES

The previous section has demonstrated significant performance potential from applying translation schemes that can re-optimize a program as it executes. We have obtained the results using various manual and semi-automatic methods. In this section we describe basic services that will allow us to implement the described optimizations in a comprehensive framework that enables a wide range of adaptive program optimizations to be applied automatically. We will measure these services using a simple test program.

Our scheme optimizes code *intervals*, which are code sections that have a single entry point and a single exit point. We transform intervals into subroutines 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. In contrast to dynamic compilation approaches proposed in related research, our scheme does not interrupt the execution of the application and, hence, keep overheads minimal. Consequently we are able to use standard tools for the program restructuring and compilation. In this section, we give an overview of the services provided by our framework as well as its general structure. Details of these scenarios are given in [18].

3.1. THE FRAMEWORK

Our framework provides services to (1) monitor the characteristics of a 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.

Figure 5 shows the basic structure of our framework. Optimization occurs at the granularity of intervals, which can be either user- or compiler-selected. The compiler replaces each interval in the original code by an `if-else` block that selects between a call to the code manager and the default statically-compiled 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.

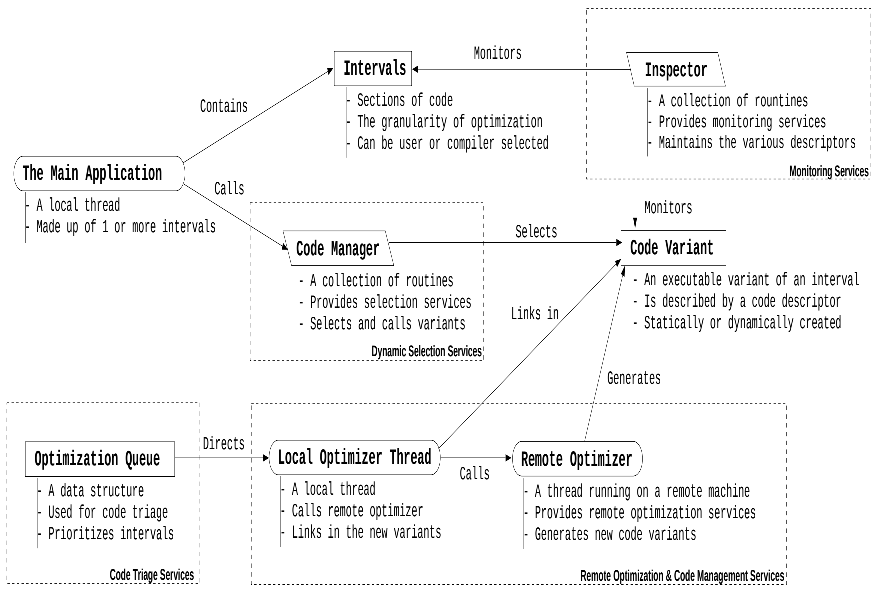

Intervals
- Sections of code
- The granularity of optimization
- Can be user or compiler selected
Monitors
Inspector
- A collection of rountines
- Provides monitoring services
- Maintains the various descriptors
Monitoring Services
Contains
The Main Application
- A local thread
- Made up of 1 or more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

Figure 5 Overview of the optimization framework.

The behavior of each interval is monitored by the Inspector. The Inspector contains a set of routines for collecting environmental and program characteristics. Access to this information is facilitated by an IPG runtime system, which will be available uniformly across machines. Calls to these routines are embedded in the code manager 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. The IPG runtime system will also allow this information to be saved and retrieved for future application runs, hence building up a program optimization history.

Intervals that have sufficiently large execution times will call the code manager to select and execute an appropriate code variant. The code manager 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 it has become stale. Each code descriptor will provide the conditions under which its variant becomes stale, and the code manager 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 Code Manager. 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, 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. In the implementation of this mechanism we assume the availability of a shared file space across the IPG. Such facilities are already available, for example through the PUNCH [10] network computing system.

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. While we use specific compilers and operating systems in our current implementation, an important property of the future IPG will be to make such facilities uniform across machines.

3.2. EXPERIMENTAL RESULTS

We manually applied this framework to a simple application, consisting of a sequence of matrix multiplications of varying matrix size. We encapsulated each interval in a subroutine and placed an `if-else` block at its original location. We transformed all but one program section, which 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 intervals. These heuristics are embedded in the corresponding Code Manager subroutines. At runtime, they select the best combination of loop tiling, conditional reduction assignment and loop unrolling. An example of a conditional reduction assignment (CRA) transformation is shown in Figure 6. The Remote Optimizer was written to generate variants that were not unrolled as well as 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 an *applicable* variant with the smallest average execution time. This '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 Code Manager 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.

In addition to the default constraints described above, we place special restrictions on variants that employ tiling and/or CRA. Whether a variant that uses tiling is applicable is based upon loop bounds. The heuristic aims at exploiting reuse of the matrix in our test program. It assumes that the main matrix multiplication loop needs to be tiled if

```
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 6 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.

and only if $N \geq \sqrt{C}$, where C is the machine's effective cache size and N is the matrix 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 also becomes stale after the user-specified time interval.

Each CRA variant is transformed such that it returns 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%, the CRA optimization is considered to be applicable, and 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. Table 1 summarizes the heuristics used for each optimization.

Table 1 Optimization Technique Parameters

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

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. 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.

3.3. EVALUATION

We ran our example application on both a SPARCstation 20 and an UltraSPARC Enterprise. Both machines were multiprocessors, however, all compilation was still performed on remote machines. When the SPARCstation was running the application, compilation was performed on the UltraSPARC, and when the UltraSPARC was running the application, compilation was performed on the SPARCstation. The SPARCstation 20 has four 100 MHz HyperSPARC processors, each with a 256 Kbyte external cache. The Enterprise has six 250 MHz UltraSPARC-II processors with 64 Kbyte internal data caches and 1 Mbyte unified external caches. All libraries were shared through the network file system. We set both the stale and the reset time intervals to 600 seconds. 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, (2) a 512x512 sparse matrix, (3) a 512x512 dense matrix, and (4) a data set which included the previous three data sets in succession. The algorithm includes a sequence of 100 matrix multiplications. 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.

Figure 7 shows the execution time of the four data sets on each machine. The execution time is given for our framework, as well as for four statically optimized variants: (1) the code with no optimizations applied, (2) the code with CRA applied, (3) the tiled code and (4) the tiled code with CRA applied. Loop unrolling is not shown in Figure 7 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 20% of its execution time. On both machines, the dynamic framework was able to outperform all of the statically optimized variants when the mixed data set was used.

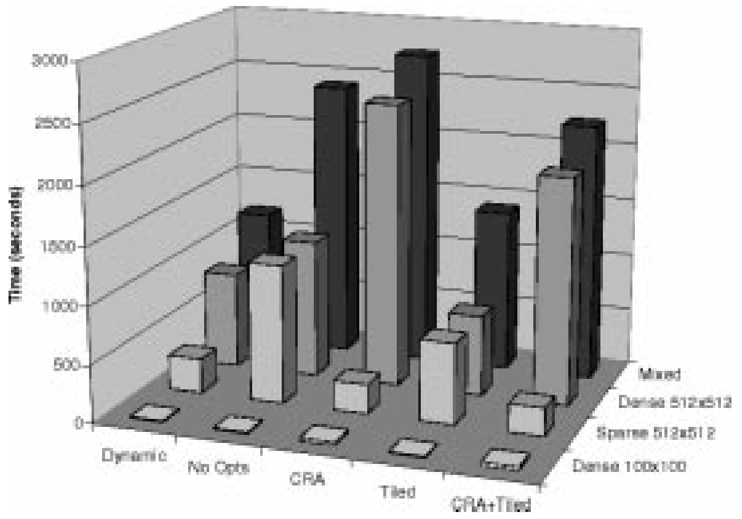
The largest variations occurred in the Dense 512x512 data set on the SPARCstation 20. The CRA optimized variants showed large overheads when run on this data set and machine. In the course of decision making, our framework must execute sub-optimal variants in order to determine the denseness factor as 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 19% slower than the best statically optimized variant. The CRA optimization showed less overhead on the UltraSPARC, and there our framework was within 3% of the best variant on this data set.

When the data sets are mixed, we see that our framework executes 15% faster on the SPARCstation, and 28% faster on the UltraSPARC, than the fastest statically optimized version on each machine. It should be noted that the fastest statically optimized variant is different for each machine. 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.

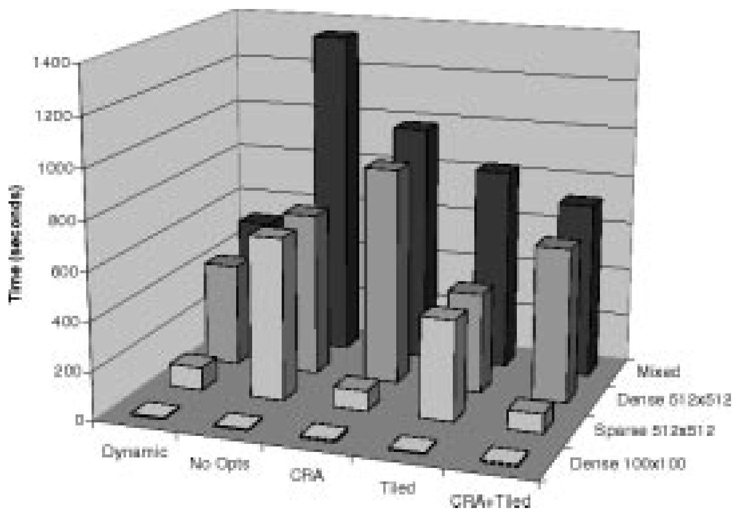
The administrative overheads incurred from our framework were negligible in this application. We found that on average less than 0.1% of the execution time was spent in the initialization and cleanup phase. And only 1% of the execution time was spent in dynamic code selection and maintenance. The remaining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

4. RELATED WORK

To our knowledge the presented work is the first approach to developing compiler technology for direct use by the information power grid. Related system software for IPG applications is being developed by projects such as PUNCH, Legion, and Globus. The PUNCH system comes closest to the idea of an IPG with an end-user interface. It allows users to run a number of pre-installed applications "on the Web". An advanced scheduler decides where to best run a requested application [11]. The predictive capabilities of the scheduler is related to our



(a)



(b)

Figure 7 The performance of the application on (a) a SPARCstation 20 and (b) an UltraSPARC Enterprise. Dynamic refers to our framework. The other data points refer to statically applied optimizations.

adaptive scheme. However, the PUNCH scheduler adapts to the environment on a per-request (i.e., per-program) basis, whereas our scheme adapts to changing environments during the program's execution.

The Globus [6] project differs from our view of the IPG in that it focuses on the provision of services for the development of IPG applications. Examples of these services are authentication, resource allocation, remote data access, and fault detection. These services are largely orthogonal to the ones provided by our compiler scheme. For example, using Globus services the programmer may explicitly migrate a program to a different machine, as a result of which our compiler-inserted modules adapt selected code sections to the new environment.

Legion [8] is an operating system for "Wide-Area Computing" that addresses specifically the needs to share resources across heterogeneous domains with different administrative requirements and procedures. It applies a consistent object-oriented methodology to the construction of the operating system and its services. It provides a single, global name space to all shared resources. Similar to Globus, Legion provides services for distributed applications, addressing issues orthogonal to the goal of the presented compiler techniques.

The compiler research that is most directly related to the presented techniques deals with runtime optimizations techniques. 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 Code Manager and the use of the default static version is done through multiversioning.

Gupta and Bodik [9] 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, being general, does not however preclude the use of other selection schemes. Like dynamic feedback, Saavedra and Park [15] 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, 13, 14, 1, 3, 5, 12]. 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.

5. CONCLUSIONS

The discussions, experiments, and results presented in this paper represent a first step towards our goal of creating compiler technology that can translate applications for the Information Power Grid (IPG). The IPG is viewed as a very complex target computer system that has a large number of parameters unknown at compile time. To deal with such parameters the compiler needs to aggressively defer program optimization decisions to runtime. We have presented a framework of services that facilitate optimization decisions at runtime. These services include environment monitors, management support for program history, remote dynamic compilation, code management support, and quality assurance facilities. Some of these services are inserted into an application by the compiler while others are part of an IPG runtime system.

We have presented preliminary measurements that demonstrate opportunities for improving performance in programs that adapt dynamically to changing environments. We have also implemented some of the needed services and evaluated them in a simple test application. Our measurements show that the overheads associated with our dynamic optimization methods can be kept low. Even the overhead of dynamically invoking regular compilers can be amortized, thanks to the opportunity of the IPG to execute dynamic compilation requests on remote systems. Based on these experiments and results we believe that there are both significant needs and opportunities for creating a new generation of compilers that can generate applications capable of adapting to the dynamically changing nature of the Information Power Grid.

References

[1] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pp. 149–159, May 1996, Philedelphia, PA.

[2] M. Byler, J. R. B. Davies, C. Huson, B. Leasure, and M. Wolfe. Multiple version loops. *International Conference on Parallel Processing*, pp. 312–318, August 1987.

[3] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. *Proceedings of the SIGPLAN '96 Conference on Principles of Programming Languages*, January 1996.

[4] P. Diniz and M. Rinard. Dynamic feedback: an effective technique for adaptive computing. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pp. 71–84, May 1997, Las Vegas, NV.

[5] D. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. *Proceedsings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pp. 160–179, May 1996, Philedelphia, PA.

[6] I. Foster and C. Kesselmann. Globus: a metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, January 1997.

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

[8] A. S. Grimshaw and W. A. Wulf et al. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40:1, 1997.

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

[10] N. H. Kapadia and J. A. B. Fortes. On the design of a demand-based network-computing system: the Purdue University network computing bubs. *Proceedings of IEEE Symposium on High Performance Distributed Computing*, pp. 71–80, 1998, Chicago, IL.

[11] N. H. Kapadia, and C. E. Brodley, J. A. B. Fortes, and M. S. Lundstrom. Resource-usage prediction for demand-based network-computing. *Workshop on Parallel and Distributed Systems (APADS)*, October 1998.

[12] P. Lee and M. Leone. Optimizing ML with run-time code generation. *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pp. 137–148, May 1996. Philedelphia, PA.

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

[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, March 1999.

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

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

[17] M. Voss and R. Eigenmann. Dynamically adaptive parallel programs. *Proceedings of the International Symposium on High Performance Computing*, May 1999, Kyoto, Japan.

[18] M. Voss and R. Eigenmann. A framework for remote dynamic program optimization. *Proceedings of Dynamo'00: ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, January 2000, Boston, MA.