Toward a Methodology of Optimizing Programs for High-Performance Computers

Rudolf Eigenmann
for Supercomputing Research and Development (CSRD)
University of Illinois at Urbana-Champaign, USA *

Abstract

This report describes experiences in porting the Perfect Benchmarks® programs to the Alliant FX/8 and Cedar machines. Although there is much to learn before we can really say we know how to optimize programs successfully for parallel computers in general and for hierarchical shared-memory architectures in particular, it is hoped that this report will be helpful to those who are beginning to port and transform programs for parallel machines. Perhaps it is even an interesting reference for experts. In addition, this paper summarizes our experience with the programming tools we used and discusses tools that could have helped to increase the productivity of the optimization process. This information is believed to be a useful basis for future tool development projects.

1 Introduction

This paper describes the process of optimizing the Perfect Benchmarks® programs for the Alliant FX/8 and the Cedar multiprocessor [Sta91]. This optimization effort, made at CSRD, led to significant speed improvements as shown in Figure 1. For many codes the gained performance is the best reported in the Perfect database [Poi90] for the Alliant FX/80 machine.

The intent of this paper is to exploit this success for two purposes. First, many of the optimization steps are similar for many of the programs, and these steps can be expressed as a set of rules, that is, a methodology of how to proceed in optimizing programs. Certainly there is additional expertise a user must acquire in order to transform a program into its optimal form. Nevertheless, the methodology developed in this paper is believed to be a significant aid for the non-expert and perhaps an interesting point of comparison for the master.

The second purpose of this paper is especially important: to describe tools that could reduce the time invested in the program optimization process. This is a particular concern because it appears that, despite many research projects to design better programming environments, there is little evidence that newly available tool sets improve productivity in optimizing ordinary programs. This is a striking contrast to the desire for better tools encountered among software engineers. It motivates our attempt to lay a solid foundation by describing time-consuming development phases and opportunities for supporting tools of a project that has already proven to lead to successfully optimized programs. In fact, an effort to address the tool questions raised has already been initiated and is described in [EM93]. The experience from early tool prototypes of this project is included in the present report.

Porting by Optimizing at the Program Level

The main goal to which this paper contributes is to facilitate the transformation of programs originally written in a sequential language (Fortran77, in this case) so they will run at high speed on a parallel machine. Usually this is called porting, optimizing, or parallelizing programs. Methods and tools to achieve this objective are complementary to those applied for writing new parallel programs [DSB89]. It is not uncommon to port code by studying in detail the problem being solved and then finding parallelism that can be exploited by the machine. In contrast to this application-level approach to porting programs, the method considered in this paper is termed *optimization at the program level*. Program-level improvements exploit information that is derivable from the program text only. Typically one tries to find parallelizable loops that a compiler could not optimize automatically. Although optimizing at the application level can lead to higher speedups, program-level optimizations are less time-consuming.

Porting programs is different from developing new parallel programs, although there are development steps

* This work was supported by Army contract #DABT63-92-C-0033 and the U.S. Department of Energy under grant #DOE DE-FG02-85ER25001. This work is not necessarily representative of the positions or policies of the Army or the Government.

that are similar. New developments require more effort in designing and prototyping a program for a given scientific problem. Although issues of the program design phase are important, porting and optimizing existing application codes to new machine architectures is believed to account for the major effort currently invested in high-performance application software.

How much potential for performance improvement is there when optimizing at the program level? Table 1 shows the Perfect Benchmarks® programs and their execution times from automatic parallelization, program-level optimization, and the best time in the Perfect Benchmarks reports [CKPK90] for the Alliant FX/80 machine as well as for the Cedar machine. For most programs, program-level optimization achieved a significant speed improvement over the automatically parallelized version. The Benchmark reports for the FX/80 give an even higher number in only few cases, which indicates that the program-level method of optimizing programs is a very beneficial approach. It can be taught and learned in a systematic way and supporting tools can be provided, as we show later.

Program	Alliant FX/80			Cedar	
	Automatically compiled	Program-level optimized	Best reported for Benchmark	Automatically compiled	Program-level optimized
ARC2D	8.0	10.6	7.9	13.5	20.9
FLO52	9.0	14.6	11.4	5.7	15.1
BDNA	1.9	5.6	3.8	1.8	8.5
DYFESM	3.6	10.3	5.4	2.2	11.4
ADM	1.2	7.5	1.5	0.6	10.1
MDG	1.2	7.3	4.9	1.0	20.0
MG3D	1.6	13.3	1.1	0.9	48.8
OCEAN	1.5	8.9	1.4	0.7	16.7
QCD	1.2	7.0	3.5	0.5	3.9
SPEC77	2.3	10.2	8.4	2.4	15.7
SPICE	1.1		4.2		
TRACK	1.0	5.1	1.1	0.4	5.2
TRFD	2.2	16.0	2.3	0.8	43.2

Table 1: Speedups of Perfect Benchmarks programs on Alliant FX/80 and Cedar. The numbers show performance improvements over the serial program execution. (See [EHJP92] for details on used compilers and applied program transformations.)

This paper is based on experiments that originate from a research effort to identify program transformations that can be automated in parallelizing compilers. The intent was to apply and measure manual changes to the Perfect Benchmarks suite. These experiments have been described more detailed in [EHLP92, EHJP92]. The effort led beyond the original goals of the project in that for some of these programs the best performance reported in the Perfect Club was achieved, as shown in Table 1. Although, for the purposes of this report, the most important result of this optimization effort is that it led to true program performance, the reader may find it helpful to know about the compiler background of the project.

2 Porting and Optimizing Parallel Programs

Our basic approach to optimizing programs was straightforward: first, the time-consuming parts of the programs were determined; then these code-sections were improved one by one.

Most often the analysis was done on a loop-by-loop basis. The first step was to get a rank list of time-consuming loops. For each of these loops potential improvements were then determined by looking at the automatically parallelized code, the compiler listing, and timing profiles. The growing list of transformations that had already proven useful in the experiments was another valuable source of information. Applying transformations and experimenting with them was the next major step; often, this resulted in a changed loop profile, yielding a new display of the most time-consuming program sections. This process was reiterated until there were diminishing returns.

The following sections give a more specific view of this optimization cycle. Existing tools and new, desirable ones will be described, and a catalog of successful program transformation techniques is outlined.

2.1 Instrumentation for Performance Measurement

The optimization of a given program always started by identifying the time-intensive loops of the program. All loop nests in the serial source code were instrumented. All optimized program variants were derived from the instrumented source code. As a result, loop-by-loop speedups could be derived from the profiles, which provided valuable initial hints about how well loops were optimized.

Fully manual instrumentation is not practical. A simple automatic instrumentation script was written soon after the project of optimizing the Perfect Benchmarks programs began. It works well most of the time. In a few notable cases, manual modification of the instrumentation code was necessary: where instrumentation of inner subroutines perturbed the program execution unduly, and where instrumentation of outermost loops was too coarse to discriminate between inner parallel loops.

The usual instrumentation tool found in Unix environments is called *gprof*. It provides the user with a subroutine-level timing profile. Since this profile is too coarse for loop-level analysis, it motivated the design of

the new tool mentioned above. Compatibility between related tools is certainly very desirable. Notable complementary tool efforts have also been made at CSRD [TJC91, SBSS+92].

2.2 Manual Restructuring

2.2.1 Alternative ways of improving program performance

There are three different ways of restructuring a program. One can search for the best set of compiler options, one can modify the sequential source code, or one can improve the parallel code after it has been transformed by the parallelizing compiler.

Finding a good set of compiler options is a fast and practical way to improve performance. If a candidate set of options is known, it is straightforward to generate executables for all these variants, profile them, and select the best variant for each subroutine. The subroutine-oriented *gprof* profiling facility is sufficient for that purpose. Similar to selecting compiler options is the choice of the best preprocessor. In the experiments described in this paper, two parallelizers were used: Kap and Vast. Although their overall performance was similar, interesting differences could be seen at the individual loop level. Up to 50% speed improvements were observed when experimenting with compiler options.

Modifying the sequential source code can be done in two ways: directives can be inserted that tell the compiler to apply certain transformations, and one can change the program text so the compiler can recognize more parallelism. Usually this requires some understanding of how the parallelizing compiler works.

Modifying the parallel program output of the restructurer can be easier than modifying the source, since it does not require this understanding. However, the user is now restructuring a parallel language and he or she has to learn both the new language syntax and the semantics of parallel execution. Output languages of parallelizing compilers have two possible forms, they can resemble Fortran77 with additional directives that, for example, tell the back-end compiler to produce parallel code for a given loop, or they can contain explicit parallel constructs, for example DOALL statements for concurrent loops or *triplet notation* for vector statements. The language used in the project described here is CEDAR FORTRAN [Hoe91], a parallel Fortran dialect developed at CSRD. Most of the optimizations to the Perfect programs were done by modifying the CEDAR FORTRAN code, usually starting from the variant produced by the restructurer when given the best options.

2.2.2 Issues in manually restructuring programs

There are two distinct tasks in manually parallelizing programs: finding parallelism and improving the efficiency of the generated parallel code.

The first one is mainly an analysis task. The loop that can be run in parallel must be determined and transformed into parallel syntax. In our experiments this usually caused the major performance gain. This is because in many Perfect Benchmarks programs the automatic restructurer did not find parallelism in the most time-consuming loops, although they could be parallelized manually.

The second issue is machine-specific. One has to find a form of each parallel loop that takes advantage of the architectural features. Observed speed improvements from such transformations range from 20 to 70% on the Alliant FX/80 machine. This machine fits the loop-oriented parallelization scheme of available restructuring compilers. The issue of parallel code generation is more important in new machine structures. On Cedar, the issue of code generation is performance-crucial; in multiprocessors with an even higher access cost to non-local memory, the generation of data transfer instructions may even be pivotal.

2.2.3 Sources of information

There is a range of information sources that were used in the Perfect code optimization project for analyzing the potential parallelism and finding appropriate code transformations.

The profile data show the time-consuming parts of a program and the speedups achieved by individual loops. (see Section 2.5 *Performance Analysis*).

The restructuring compiler supplies optimization notes that help to explain the applied transformations and provides reasons for those that failed. It can also ask questions about the value of certain variables which are crucial for disproving data dependences. In addition, the used compiler can list data dependences and their attributes. All parallelizing compilers generate this information at least internally, and it seems that better communication to the user would enhance the optimization task significantly. In fact, listing data dependences was not an original feature of the compiler used for our effort, but was implemented as part of the project. The provision of more detailed compilation reports of applied and failed transformations is another desirable feature of future restructurers.

The catalog of transformation techniques that have been successful in our optimization efforts, is one of the most helpful sources of information. These transformations were reported in [EHLP92, EHJP92]. Section 2.2.4 outlines the most important of these transformations.

Is is useful to know the maximum possible speed of a given loop, and tools giving this information are being developed at CSRD [CY91, PP93]. They do not directly give the potential that is exploitable by a compiler; rather they indicate the parallelism observed at runtime by performing a thorough data-flow analysis.

Nevertheless this information is valuable as a rough estimate of an upper speed limit.

It cannot be denied that much analysis is needed in terms of browsing through the source code and searching for opportunities for improvements. This can be a time-consuming task, although experience in this process can significantly reduce the time required. It seems possible that some of this expert knowledge can be captured in an environment that gives guidance to the user.

2.2.4 Manually restructuring the parallel code

The following sections include the most important transformations that can be credited for the program performance improvements shown in Table 1.

Array privatization. Many of the major loops of the Perfect Benchmarks programs were parallelized with the help of the array privatization technique. This was done by analyzing definition/use patterns of arrays inside loops. Arrays whose elements are defined before they are used within a loop iteration (i.e., they first appear on the left-hand side of an assignment statement) were declared loop-local. This technique is a natural extension of the scalar privatization (or scalar expansion) technique, which is in the repertoire of available compilers. Compilers recognize scalar variables that are defined before they are used inside loops and transform them into one of the following parallelizable forms:

```
      REAL a(n),b(n),c(n),t
      DO 10 i=1,n
         t    = a(i)+b(i)
         c(i) = t + t**2
10 CONTINUE
```

⇓

```
      REAL a(n),b(n),c(n),t(n)          REAL a(n),b(n),c(n)
      DOALL 10 i=1,n                    DOALL 10 i=1,n
        t(i) = a(i)+b(i)          or        REAL t
        c(i) = t(i) + t(i)**2               t    = a(i)+b(i)
10 CONTINUE                                 c(i) = t + t**2
                                  10 CONTINUE
```

Similarly, an array used as a temporary variable can be privatized:

```
      REAL t(m)                         DOALL 10 i=1,n
      DO 10 i=1,n                         REAL t(m)
        DO 5 j=1,m                        DO 5 j=1,m
5         t(j) = ...      ==>     5         t(j) = ..
        ...                                 ...
        DO 10 j=1,m                       DO 10 j=1,m
           ... = t(j)                       ... = t(j)
10 CONTINUE                       10 CONTINUE
```

Array privatization is more difficult than scalar privatization because we have to be sure all array elements are assigned before they are used. This is not always as obvious as in the given example. Sometimes parts of the array are assigned in various statements, which requires a careful subscript analysis. Often, variable values need to be analyzed symbolically. In addition, we have found many important loops that contain subroutine calls. The def/use analysis of arrays had to be done interprocedurally.

Parallel reductions. Another frequent program pattern is the accumulation of values into a scalar variable. In simple cases compilers can recognize and parallelize these patterns. For example the loop

```
      DO 10 i=1,n
         sum = sum + a(i)*b(i)
10 CONTINUE
```

will be recognized as a reduction operation (a *dotproduct*). It can be vectorized using special machine instructions and executed concurrently by computing partial sums on all processors followed by a sum over all partial results.

There exist more complex patterns in our program suite that were not transformed automatically by the compiler. Examples are multiple statements per loop summing into the same variable and accumulations into array elements. In many cases the accumulated values were not read elsewhere in the loop, which allows one to reorder the sum statements as necessary for building partial results in parallel. Partial results accumulated per processor (or loop iteration) can be added to the global (original) variable in two different ways: in a synchronized section within the loop or in an additional sum statement after the loop. Sum statements after the loop were usually preferred, because they permit the loop to run completely parallel. The sum statement at the end of the loop was usually less costly than loop synchronization.

Generalized induction variables. Available compilers can recognize *induction variables* that get incremented in each loop iteration by a constant term. The compiler substitutes them by an explicit function of the loop index variable, which removes the parallelism-inhibiting data dependences. In the codes OCEAN and TRFD, more general forms of induction variables were found, those that are modified using multiplication instead of addition or others that are placed in *triangular* loops. (In triangular loop nests an inner loop limit depends on the value of an outer loop index.) Applying this techniques in the program OCEAN, a major loop could be parallelized and sped up by a factor of 8.1.

Runtime dependence test. Sometimes data dependences cannot be disproven because the values of certain variables are not known. If there is reason to assume that in some instances of the executions of a given loop the values are such that the loop iterations are independent, one can insert code that tests for these values and branches to either a serial or parallel execution of the loop. For example in the OCEAN code a major

loop could be improved by testing whether certain array subscripts form a linearized array pattern. It turned out that, at runtime, the loop was always parallel.

Permutation vector analysis Compilers do not parallelize loops if written arrays contain subscripted subscripts. Sometimes one can determine from the program text that the subscript arrays are permutations, which means all index values are different (e.g., they represent a permutation of a sequence of numbers). Such loops can usually be savely parallelized. Quite often the assignments of index vectors are done in different subroutines. Hence, the analysis must be done interprocedurally.

Value analysis. Values of variables need to be known for various reasons. Loops containing induction variables can sometimes be parallelized only if the value of the increment is known. To decide whether to apply certain transformations, it is necessary to know the number of loop iterations. For example, stripmining can degrade the performance when applied to loops with small iteration counts. We have also found loops that were parallelized by the compiler because it did not identify that there was only one iteration.

Compilers can analyze these values if the assignment is in the same procedure as the loop being optimized. Manual value analysis is necessary to do this interprocedurally. Sometimes simple constant propagation is sufficient, for example where variables are assigned once in the program initialization phase. We have also found cases where the values originated from the index of an outer loop. In our experiments, a profiling tool that displayed the number of iterations on each loop was useful.

Finding outer loops. A general trait of the automatically parallelized Perfect Benchmarks programs was that innermost loops were parallelized. We have measured performance gains of up to 50% on the Alliant FX/8 machine from parallelizing outer loops even compared to code variants where all inner loops were completely parallel. On the Cedar machine this difference is much more significant, since the latency for starting a loop across all clusters is relatively high (approximately 100 μs).

Finding outer parallel loops is often no different from finding parallel loops in general. All the techniques described here need to be considered to achieve this. However, we have also seen situations where internal limitations prevented the compiler from parallelizing outer loops. These loops may be recognized easily as parallel.

Balanced stripmining. The stripmining transformation splits a loop into two nested loops; usually for exploiting two levels of parallelism (such as vector and concurrent execution). This can improve performance significantly provided the loop has enough iterations. However, it can degrade performance if there are only few iterations. Therefore it is very important to consider the number of loop iterations.

On the Alliant FX/8 machine it has turned out to be good practice to stripmine singly-nested loops into number-of-processors outer iterations for concurrent execution. Usually it takes a vector length of at least four to amortize the vectorization overhead. Thus, on the 8-processor Alliant machine the threshold for successful stripmining is about 32 iterations. Cedar has four times more processors, which requires the loop trip count to be at least 128. In addition, the loop body has to be sufficiently large so the startup latency gets amortized. Loops that are shorter than one millisecond should usually not be considered for stripmining and spreading across Cedar.

Globalization and localization. The placement of variables in the Cedar memory hierarchy can affect the program performance significantly. In a simple loop execution model all shared variables are placed in global shared memory. Cluster memories are populated with scalar and array variables that can be localized as described in the description of *Array privatization* above. The currently available Cedar restructurer applies this model. In addition, the compiler places all subroutine interface variables (parameters and common blocks) in global memory.

More refined data placement is desirable. Parameters of subroutines that are called concurrently (as a result of manual parallelization) should be placed in cluster memory if they are declared loop-private. Partitioning and distributing data onto the cluster memories was applied in only few situations of the Perfect programs so far.

Defining runtime parameters. The most performance-critical runtime parameters are the number of clusters and processors used, and the cross-cluster loop scheduling scheme.

Programs can be written so they adapt to the number of clusters and processors. For example, this is important for the stripmining transformations, which can make use of CEDAR FORTRAN constructs for deriving the right strip size. The actual number of clusters and processors used at runtime can be defined interactively at the Unix level. This is useful for experimental purposes, such as determining speedup curves.

The scheduling scheme is chosen by linking with the appropriate run-time library. There are two libraries of main interest. Loops can be spread across Cedar by using either a dynamic or a static scheduling scheme. For most Perfect programs the dynamic scheme was applied. However, some loops showed better performance when

scheduled statically. The static scheduling scheme is of further importance when data are distributed onto the Cedar cluster memories because it chooses a fixed binding of iterations to clusters.

2.2.5 Editing transformations

The Perfect programs were restructured using common text editors. The starting point was usually a subroutine restructured by the parallelizing compiler. The editing process was not unduly time-consuming, relative to the whole optimization process. The more costly part, as a consequence of the manual transformations, was sometimes the debugging phase; this point is discussed in Section 2.4 below.

A straightforward idea for improving the editing process is to use an interactive restructurer. A reduction in development time can be expected mainly from the reduced error rate. To be of significant help such a tool would have to check the correctness of transformations interprocedurally and it would need to know the repertoire of new important transformations as described above.

2.3 Generating and Executing Code Variants

Generating executables was on one hand a file management issue. The situation is described in Section 2.6. Another issue was to compose individually optimized subroutines into an executable file. Many of the Perfect Benchmarks have tens of subroutines. The standard Fortran way of dealing with this situation is to split up the master file, compile the subroutines individually, and link the individual pieces for different program variants. Because of the variety of program variants this was unwieldy, and it led to many version conflicts. The alternative was editing a single file containing the whole program. This was impractical since recompiling the whole file was time-consuming, and manual cutting and pasting of individually optimized subroutines was not only cumbersome but error prone. The tool desired was a command like "link these individual subroutines plus take the rest from the master file variant XY".

When the codes were being linked and prepared for execution, additional options needed to be considered such as various runtime libraries or trace facilities. Even more options had to be dealt with at runtime because these programs could be run on multiple machines and with various runtime parameters, for example single-user or multi-user mode. Multiple data sets are not involved in the Perfect Benchmarks experiments, but it is easily conceivable that they could be, which would lead to an even larger set of executable and output files and thus again raise the file management issue.

2.4 Debugging

This paper discusses the transformation of existing Fortran77 programs into efficient parallel codes. This task leads to a restricted class of programming errors compared to new program designs. In our task, the original sequential program is usually correct. The errors to find are those introduced by the applied transformations. (Although this is mostly true for the Perfect Benchmarks considered here, this cannot be taken for granted in general. All software contains hidden bugs that are not apparent but surface as a result of the exposure to a new machine and software context. Fortran codes are particularly prone to these bugs, partly because many compilers permit violations of the FORTRAN77 standard.)

Two classes of errors seem worth mentioning: simple-to-detect errors that are often due to mismatches in variable names or incorrect variable declarations, and more subtle errors that are caused by conflicts between called and calling subroutines. The latter bugs are not only due to the human weakness in grasping two subroutines at once; it is aggravated by the compilers' inability to do consistency tests across subroutine boundaries.

Since the experiments described here have always started from a correct program variant, the usual debugging method was to stepwise undo the transformations until the program was correct again. For this purpose it was important to carefully save intermediate modification steps so that the error could be traced back. Again, the file-naming scheme is important: Every modification should be given a new name, and once program execution results are available, they should be named correspondingly and no further changes made to these files. Maintaining a record of compilation commands was a further important aid. Keeping this discipline often saved a lot of debugging time. Tools that enforce this discipline are very desirable. In our experiments small utilities were created to this end, such as a Unix alias for the compilation command that records the command plus time stamp in a log file, and a script that runs a program and renames the output files according to our naming scheme. Tools that provide this functionality in a more generic way seem to be worth creating.

Where multiple subroutines are possible culprits of a problem, it helps to have both original and transformed subroutines available in object form and then stepwise exchange new and old routines. This process was even helpful in some cases where an error had to be found that did not originate from the manual modifications, such as a compiler bug. One can start using 50% of the original subroutines and, as in binary search, more routines can be included or excluded. This process seems clumsy; however, its power is in the automatability. Note that the program-level analysis considered in this paper does not necessarily require the user to understand

the problem behind the algorithm. Debugging methods that do not require this knowledge and those that can be automated, even if the debugging process has to be submitted "over night", are superior.

As can be expected, conventional debugging methods were used also. Sometimes the simple and time-consuming way of inserting print statements had to be chosen. A low-level debugger built into the runtime system [EM91] was of further help. It could indicate the error address which, together with some utilities, pointed to the Fortran source line. Further, it showed the current call chain and facilitated the complete inspection of the machine state at the binary level.

2.5 Performance Analysis

The profile data resulting from many program variants needed to be collected and tabulated in some comprehensible form. For all loop intervals, minimum, maximum, average, and accumulated times were recorded, and the first question to be asked was: "Where is the most time spent in the program and what speedups do these loops show?" Later, one wanted to understand more exactly why loops exhibited a certain speedup. Questions were asked such as: "What is the iteration count of a loop; what is the loop execution time; what does the source and optimized code look like; what transformations were applied; what performance improvement is attributable to certain transformations?" Finally, data must be abstracted and compressed into fewer numbers and tables, and graphical representations need to be generated for the reports that must be delivered. The following section describes the process of analyzing the performance data and finding transformations for Cedar programs in a systematic way.

2.5.1 Optimization factors

The performance analysis process was most crucial for finding transformations that improved the performance. Since many of these analysis steps were similar for many programs, they can be expressed as a methodology of performance analysis, which represents an important part of the methodology of porting and optimizing parallel programs. Basically one determines a set of *optimization factors* for each loop nest of a program, which then leads to a number of questions whose answers help to explain the loop performance and point to optimizing transformations. The following description is held Cedar-specific so the discussion can be concrete. However, the basic method is believed to be applicable to a wide range of architectures, primarily those that offer a shared address space and fast access to some local memory. The notion of clusters is important as well. Many scalable machines are structured into clusters in terms of groups or rings of processors within which communication costs are less than to other processors. For architectures that have no cluster structure the following methodology is still meaningful by considering one-processor "clusters". Examples of Cedar-related systems are the Kendall Square Research and the Cray MPP machines.

The optimization factors are a set of numbers that characterize the improvements and overheads introduced by the transformation steps from a given serial or parallel loop version to its Cedar-optimized variant. The execution time of four variants of the program are measured initially:

1. $O1$: The baseline version for one Cedar cluster. This is the available automatically parallelized or manually optimized program for the FX/8 multiprocessor.
2. $O1g$: The 1-cluster version with the data put in global memory (i.e., in shared address space). Ideally all data that need to be shared by loops that could be spread across Cedar will be put in global memory. The compiler supplies an approximation to this.[1]
3. $C1$: The baseline Cedar version which is run on one cluster.
4. $C4$: The baseline Cedar version which is run on four clusters.

The optimization factors are now derived from the loop-by-loop timing information of the four program variants:

$globalization\ factor(GP) = O1/O1g$ [2]
$spreading\ setup(XO) = O1g/C1$[3]
$spreading\ speedup(XS) = C1/C4$
$cluster\ speedup(CS) = O1/C4$

The interpretation of these factors is as follows: An important reference machine for Cedar performance is the Alliant FX/8 multiprocessor. The FX/8 is a traditional "flat" shared-memory architecture and four such machines are the basic Cedar building blocks, that is, there are four Cedar clusters. Ideally a program would run on Cedar four times as fast as the FX/8-optimized version; that is, the cluster speedup is 4. The reasons for which a code does not have this ideal cluster speedup can be categorized into three areas. First, the 4-cluster version has some data placed in the global memory, which results in higher access costs compared to cluster memory. This is represented by the globalization penalty. Second, the program needs to be transformed in such a way that it exploits all Cedar clusters and processors. For example, the loop iteration spaces are split into four parts to employ all clusters. The remaining parallelism within one cluster is less than before, which may cause more execution overhead. This performance loss is rep-

1. in Cedar this takes Kap options `-c=1 -g`
2. Sometimes it is convenient to speak of a *globalization penalty* in percentages, which is $(O1g/O1 - 1) * 100$, or its negative, the *globalization benefit*.
3. Similarly, the *spreading overhead* is $(C1/O1g - 1) * 100\%$

resented by the spreading overhead. Third, once a loop is in a form that can be executed using all clusters, there are still reasons why it may not run four times faster than the same loop executed by one cluster; for example because of contention in the shared memory. Thus, the spreading speedup is less than four.

2.5.2 Performance analysis methodology

This section presents the main reasons behind good or bad optimization factors. In addition, where available, recipes are given for transformations that can improve the optimization factors. The programmer is advised to discuss which of these reasons apply for the optimization factors of each time-consuming loop and derive appropriate transformations.

Globalization Penalty Discussion. GP values have been observed between -50 and +150%. Best values are attributable to long vector operations and ideal strides, whereas high penalties can be seen for scalar accesses. Non-vectorized loops will have a high GP. In between are vector operations that involve subscripted subscripts or large strides. GP values can be further biased by how well the 1-cluster code is able to exploit the cache. The reasons for this are as follows:

The global memory can be de-referenced efficiently in prefetch mode. Prefetch code is generated automatically by the compiler for vector statements that read data from the global memory. Several compiler and hardware details are useful to know in this context: Prefetched data accesses from global memory can be as fast as from cluster cache under ideal circumstances, such as long vectors and best stride. In [GJT+91], best performance was measured for a stride of 16. Prefetch lengths of 512 are ideal. Up to 10% performance loss was measured for prefetch lengths of 32, which is what the compiler generates. There are circumstances that may stop a prefetch operation at runtime. One of them occurs when the memory address of a prefetch operation crosses a page boundary of the physical memory. The prefetch hardware does not know the map of virtual to physical addresses, so it has to wait at the end of a page until the computation catches up and issues the next virtual address which causes a physical address request to be issued to the prefetch hardware. Vector operations with a large stride are frequently interrupted this way. Another cause that can discontinue a prefetch operation at any time is a context switch.

Vector operations involving subscripted subscripts are executed by first reading the subscript in a prefetched vector operation. Then a *gather instruction* is executed to read the data. This operation cannot be prefetched.

There are a few optimization recipes: Vector strides can be altered by loop interchanging and changing stripmining ratios. The parallelizing compiler tends to generate stride-one references and place concurrent loops in outermost positions. In non-vectorized loop nests that have enough concurrent iterations one may stripmine and interchange one loop part to the innermost position for vectorization.

Loops with scalar accesses are candidates for data localization. Scalars that are read-only may be copied to cluster memory in a vector operation before or at the beginning of the loop. Written data that are read in later iterations may be duplicated in cluster memory. In addition to localizations that are limited to the scope of one loop, there may be opportunities for placing data in cluster memory across the execution of multiple loops. This is a very important transformation; however, there are no general recipes available. [ME91] describes the data localization techniques applied to the Conjugate Gradient algorithm. Strategies for data partitioning and distribution are currently being investigated in many related research groups.

Spreading Overhead Discussion. XO values can range up to a few 100%. Often this is due to the reduced inner parallelism that remains after using the outermost level of parallelism for cross-cluster execution. In addition, the parallelizing compiler may have introduced unnecessary transformations to loops inside the cross-cluster loop. For example, it may have chosen inner loops for cross-cluster execution before the programmer finds an outer loop to be parallel. Loops that are too small can cause additional overhead because of the work necessary at runtime to prepare a loop for spreading across clusters (this is called the *SDOALL setup cost*[4]).

Although the spreading overhead is a useful measure, it must be interpreted with care. In some cases it is not possible to discriminate spreading overhead against lack of spreading speedup. For example, where a spread loop is executed from 1 to current-number-of-clusters. The 1-cluster execution of this loop would still have the same amount of cluster parallelism as the one-cluster-optimized version, but only one fourth of it in the four-cluster execution. Hence, the overhead from reducing the cluster-parallelism would be reflected by a diminished spreading speedup. The same holds for fully dynamically scheduled cross-cluster loops (XDOALLs).

Negative spreading overhead can occur as well. This may be considered an anomaly, for example when the inner parallel loop (CDOALL) that is chosen for cluster parallelism has more iterations than the outermost (SDOALL) loop. In the one-cluster optimization the outer loop was likely to be the concurrent loop, which performed less well than the second loop. So now, when the one-cluster execution of the cross-cluster variant is

4. `sdoall`,`xdoall`,`cdoall`, and `cdoacross` are keywords of the Cedar Fortran language [Hoe91]

compared with the one-cluster optimized version, there is a resulting speed improvement. Negative spreading overhead can also appear as a result of optimizations applied to the cross-cluster program. If much more optimization effort is invested in the cross-cluster variant than in the one-cluster program, an apparent negative overhead results.

Recipes for improving the spreading setup factor (i.e., reducing spreading overhead) are checking inner loops for unnecessary compiler-introduced transformations and undoing them, finding outer parallel loops if the given loop is too small (in terms of loop execution time), and, as a last resort, confining the loop to one-cluster execution. Usually loops that are 1 millisecond and shorter should be executed by one cluster only. Finding outer parallel loops is of course a difficult task and no different from finding parallel loops in general. However, it is important to recognize when a given loop is too small for multicluster execution, so that the effort of trying it can be saved.

Spreading Speedup Discussion. XS values usually range from 1 to 4. Factors that reduce the ideal XS are synchronization overhead, unbalanced number of iterations, paging effects, and long spread loop preambles. The XS is very sensitive of synchronized sections. Reducing such sections can improve the performance significantly.

A number of 5 equally long iterations spread across 4 clusters can speed up at most a factor of 2.5. This has to be kept in mind when stripmining loops for inter- and intracluster concurrency. Balanced iteration counts are important.

Paging effects become important even if the used amount of memory is well within the physical memory size. They occur when clusters access a page of memory for the first time. When helper clusters participate in the operation on some data for the first time, they have to first realize that they are allowed to access the data. To do this they take a *soft page fault*. This can introduce a significant imbalance in the execution time of loop iterations. Usually this effect gets amortized over the program execution time because of the repetitive patterns of programs.

An XS of 1 indicates the loop is either not parallel or not spread across clusters. In Section 2.2.4, the most successful transformations to parallelize loops were described. Non-spread loops are often synchronized *doacross* loops. These are loops whose iterations may get delayed to wait for values computed in previous iterations. The compiler generates doacross loops that execute within a cluster only, because cross-cluster synchronization is expensive. In some cases these loops can be manually parallelized for cross-cluster execution, for example with techniques described in Section 2.2.4

2.5.3 Supporting tools

The performance analysis process caused the primary desire for supporting tools. They should be capable of selecting loop profile data from certain program variants or executions, pick columns and hide others, sort, compute speedups between two columns, show the corresponding source code and compiler listing information, and generate graphs. Since the amount of data generated in the Perfect code experiments was large, their display needed to be changed often, which requires a certain interactivity of the tools. Ideally the tools would support the methodology described, compute the optimization factors, guide the user through the questions raised in Section 2.5.2, and point out possible transformations.

2.6 File/Data Management and Experiment Control

It was mentioned repeatedly that there are a considerable number of source, intermediate, executable, and result files created during such experiments. This variety was dealt with mainly by keeping an appropriate structure of directories and sticking to a file-naming convention that made it possible to identify the type and variant of a file. In addition, a log file of all the make (compilation, link, etc.) commands was kept that recorded the component files of each executable code. All of this was possible at the Unix level. It worked well as long as the number of files was relatively small. The main reason for the need for additional tools was an increase in version conflicts and apparent performance anomalies, which could often be tracked down to a forgotten compiler option or a misspelled filename. The wish arose to have a "supervisor" that checks, or better even, creates the filenames and compiler options automatically. Certainly this must be customizable, since different users have different naming conventions.

As data accumulated, the question arose whether a management system exists that could help to store and retrieve data in a flexible but consistent manner. Adopting such a system is likely not only to assist the users, but also to introduce some complications. At the very least the user has to learn the database management system, whose terminology is usually quite different from the Unix-level file management facilities. An integrating environment that provides the power of the data management capabilities without introducing a whole new world of notions would have to be designed very carefully.

3 Conclusions

The objective of this paper was to document experiences gained in a project that optimized real programs for parallel computers. The Perfect Benchmarks® programs are the currently most widely accepted represen-

tation of a supercomputer workload. The performance achieved in the described effort is significant. Therefore this report provides a solid basis for discussions that refer to real codes on real machines. Of course, there exist important programs that are even larger than the Perfect programs, and in future projects we will have to show to what extent our methodology applies to these programs.

The paper emphasizes two aspects: tool requirements and the software engineering methodology. Two important development phases that could benefit from new tools are the generation and execution of a variety of experimental program variants, and the identification of new program transformations from result data, such as the compiler listing and profile information.

The methodology applied for optimizing the programs is termed *optimization at the program level*. It is less time-consuming than application-level program transformations, because it does not require an understanding of the underlying application problem. Instead, transformations are derived from the program text, similar to an advanced parallelizing compiler. The important lesson to learn from this paper is that significant program improvements were achieved this way. In many cases the timings are the best ones reported for the given programs and machines. The program-level methodology is relatively systematic, and supporting tools are currently being designed. Thus, there is great potential for boosting the productivity of the process of porting and optimizing programs for parallel computers.

Acknowledgment

This report is based on research and development efforts from many people. Jay Hoeflinger, Greg Jaxon, Zhiyuan Li, and David Padua participated in the program optimization effort, which is described in [EHLP92, EHJ+91]. Bill Blume wrote the original version of the instrumentation tool, optimized the ADM code, and provided many facts about available compilers [BE92]. Jay Hoeflinger wrote a library that reduces the traces at runtime, recording minimum, average, maximum, and accumulated timings. Patrick McClaughry provided tools that really helped our effort [EM93].

References

[BE92] William Blume and Rudolf Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks^{TM} Programs. *IEEE Transactions of Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks^{TM}. *Proceedings of ICS, Amsterdam, Netherlands*, March 1990.

[CY91] Ding-Kai Chen and Pen-Chung Yew. An Empirical Study of DOACROSS Loops. *Proceedings of Supercomputing'91, Albuquerque, NM*, pages 630–632, November 18-, 1991.

[DSB89] J. Dongarra, D. Sorensen, and O. Brewer. Tools and methodology for programming parallel processors. In M. H. Wright, editor, *Aspects of Computation on Asynchronous Parallel Processors*, pages 125–137. Elsevier Science Publishers B. V. (North-Holland), 1989.

[EHJ+91] Rudolf Eigenmann, Jay Hoeflinger, Greg Jaxon, Zhiyuan Li, and David Padua. Restructuring Fortran Programs for Cedar. *Proceedings of ICPP'91, St. Charles, IL*, 1:57–66, August 12-16, 1991.

[EHJP92] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. The Cedar Fortran Project. Technical Report 1262, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, 1992.

[EHLP92] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmarks Programs. *Lecture Notes in Computer Science 589, Springer Verlag, NY*, pages 65–83, 1992.

[EM91] Perry Emrath and Bret Marsolf. mdb - Xylem Parallel Debugger User's Guide. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., December 1991. CSRD Report No. 1180.

[EM93] Rudolf Eigenmann and Patrick McClaughry. Practical Tools for Optimizing Parallel Programs. *Presented at the 1993 SCS Multiconference, Arlington, VA*, March 27 - April 1, 1993.

[GJT+91] K. Gallivan, W. Jalby, S. Turner, A. Veidenbaum, and H. Wijshoff. Preliminary Basic Performance Analysis of the Cedar Multiprocessor Memory Systems. *Proceedings of ICPP'91, St. Charles, IL*, I:71–75, August 12-16, 1991.

[Hoe91] Jay Hoeflinger. Cedar Fortran Programmer's Handbook. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., October 1991. CSRD Report No. 1157.

[ME91] Ulrike Meier and Rudolf Eigenmann. Parallelization and Performance of Conjugate Gradient Algorithms on the Cedar Hierarchical-Memory Multiprocessor. *Proceedings of the 3rd ACM Sigplan Symp. on Principles and Practice of Parallel Programming, Williamsburg, VA*, pages 178–188, April 21-24, 1991.

[Poi90] Lynn Pointer. Perfect: Performance Evaluation for Cost-Effective Transformations Report 2. Technical report, University of Illinois at Urbana-Champaign, Center for Supercomputing Res & Dev, March 1990. CSRD Report No. 964.

[PP93] Paul M. Petersen and David A. Padua. Static and Dynamic Evaluation of Data Dependence Analysis. In *Proc. of ICS'93, Tokyo, Japan*, July 1993.

[SBSS+92] S. Sharma, R. Bramley, P. Sinvahl-Sharma, J. Bruner, and G. Cybenko. P3S: Portable, Parallel Program Performance Evaluation System. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., September 1992. CSRD Report No. 1170.

[Sta91] CSRD Staff. The Cedar Project. *Proceedings of ICPP'91, St. Charles, IL*, August 12-16, 1991.

[TJC91] Allan Tuchman, David Jablonowski, and George Cybenko. Run-Time Visualization of Program Data. *Proceedings of Visualization '91, San Diego, CA*, October -25, 1991.