ハイパフォーマンス
コンピューティング 57−17
（１９９５．８．２５）

HPFコンパイラ用プロファイリングシステム

金城ショーン　　進藤達也

富士通(株)ＨＰＣ本部
〒２１１川崎市中原区上小田中１０１５
{syk, shindo}@flab.fujitsu.co.jp

HPFプログラマにとって，プログラムの動作情況を把握するために，プロシージャごとあるいはループや条件分岐に関わる命令ごとの実行時データを収集し報告するプロファイリングシステムが必要である．本論文では，HPF用のプロファイルシステムの設計について述べ，最適化を施したコードに対してプロファイルをとる為の手法とプロファイル用の実行時ライブラリのオーバーヘッドコストを小さくする手法について言及する．実験により，システム全体の機能をデモンストレートし，提案するプロフィリング法におけるオーバーヘッドコストがとても小さいことを検証する．

A Profiling System for an HPF Compiler

Shaun Kaneshiro　　Tatsuya Shindo

Fujitsu High Performance Computing Group, Fujitsu Ltd.
1015 Kamikodanaka, Nakahara, Kawasaki 211 Japan
{syk, shindo}@flab.fujitsu.co.jp

To better understand the behavior of their programs, HPF programmers require a profiling tool which collects and reports detailed execution time data at the procedure level and statement level for selected statements. This paper describes the design of a complete HPF compiler profiling system, and addresses the issues in instrumenting optimized code, minimizing the profile library overhead cost, and succinctly displaying the profile data. Preliminary experiments demonstrate the basic functionality of the system and verify that only a small overhead cost is associated with profiling in this implementation.

1 Introduction

High Performance Fortran (HPF) is a high-level language which simplifies parallel programming by abstracting away communication and synchronization details from the user. In order to help HPF programmers better understand the behavior of their programs, they require a tool which reports details of the execution while at the same time revealing the hidden costs in an effective manner. The profiling system proposed in this paper satisfies those two requirements.

To collect profile data, instrumentation must be inserted into the program code to mark the profiled constructs. Since the profile must provide an accurate report of the execution, the instrumentation must occur after the original code is optimized by the compiler. If the instrumentation was performed on the unoptimized code, the performance would be markedly different particularly for distributed memory machine, and is therefore, not an option. Instrumenting the optimized code is an involving process, requiring the map between the optimized code and original code to report the profile data relative to the original code. This process is a key feature in this system.

This paper presents a complete HPF compiler profiling system consisting of an instrumentation module, runtime library, graphic profile data viewer, and compiler feedback module. Several design and performance goals spanning all aspects of the system are incorporated in this implementation. The first is the selection of useful profile constructs and data. The second is the mapping between the original source code and the optimized source code during instrumentation. For the runtime library, the third and fourth requirement is the modularity in the library routine interface, and minimizing the profile overhead. Lastly, the profile data viewer should display the profile data in a succinct format.

The paper begins with an overview of the profile system, followed by a description of the instrumentation component, runtime library, and profile data usage components in Sections 3, 4, and 5. Section 6 reports the profile overhead in the system, and Section 7 briefly describes related work.

2 Profile system overview

Although this system has been designed specifically for profiling HPF programs executed on parallel distributed memory machines, it is robust in that it can be used for profiling other single-threaded languages such as Fortran90, and Fujitsu's VPP-Fortran parallel Fortran dialect. The system also supports both single processor and multiprocessor executions. Aspects of the system specific to HPF and distributed-memory machines are the selection of profiled statements and the detailed data collected by the runtime library.

The system has four basic components: automatic optimized source code instrumentation module, runtime library, profile data display application, and compiler feedback module. To use the system, the optimized user code must be instrumented with calls to the profile runtime library during compilation. The code is then executed during which time the runtime library collects the profile data. After the execution completes, the data can be viewed by the user or directed back into the compiler for aiding automatic optimizations.

The system collects profiling data at two levels, the procedure level and statement level for selected statements. As is typical of other profilers, this one profiles conditional and loop statements. But, to accommodate HPF-specific constructs in the profiling, array assignment expressions, WHERE statements, and FORALL statements are also included in the profile. These constructs were chosen with the assumption that the majority of the execution time will be spent in looping and communication-intensive constructs.

The data reported by the system is in the form of elapsed time, frequency counters, and dynamic call tree. Parallel execution reports include the classification of elapsed time into function-based categories: user code, communication, synchronization, runtime library, and global operation. The classification is intended to aid the user in locating inefficient code requiring a large amount of communication and synchronization time. Frequency counts are used for counting the number of invocations, iterations, and taken conditional branches.

3 Automatic instrumentation of optimized code

The purpose of the optimized code instrumentation module is to map the original source code to the optimized code which requires minimum transformation history maintenance

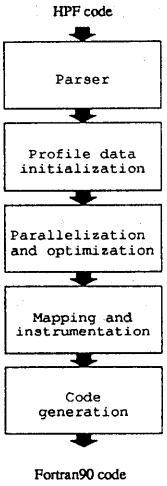

Figure 1: In the compiler, the profile initialization takes place prior to any optimizations and the profile instrumentation occurs after all optimizations are completed.

in the compiler optimization modules. The data structures used to maintain the source code information and the transformation history are described in this section followed by the code transformation, mapping algorithm, and instrumentation algorithm.

3.1 Source code data representation

Two data structures are used to maintain the original source code and transformation history information, referred to as `StmtInfo` and `SetOfStmtInfo`, respectively. The `StmtInfo` information is collected for each original statement prior to any code optimizations during the initialization stage, shown in Figure 1. `StmtInfo` contains information to identify the statement location, aid in the mapping of original code to transformed code, and report the profiled data in a readable format. The structure consists of a filename, file line number, unique serial number, pointer to the enclosing parent statement's `StmtInfo`, statement type, flag indicating whether the statement contains an array expression, and string description. To maintain the code transformation history, each current statement is augmented with a list of `StmtInfo`, called `SetOfStmtInfo`. The inclusion of a `StmtInfo` in the list indicates the current statement is derived from the `StmtInfo` statement.

3.2 Transformation history

Five basic transformations on the source code are performed by the compiler: null, merge, eliminate, expand, and move. Each transformation corresponds to an operation performed on the `SetOfStmtInfo` as described below. It is assumed that statements inherit the `SetOfStmtInfo` of their enclosing parent statement.

- The null transformation is the introduction of statements into the code which have no relationship to any original statements. The `SetOfStmtInfo` for the introduced statement is empty.
- The merge transformation is a transformation in which multiple statements are condensed into a single statement or set of statements. The `SetOfStmtInfo` for the newly created statement is the union of the `SetOfStmtInfo` of the merged statements.
- The code elimination transformation is the removal of statements from the code. The `SetOfStmtInfo` is not altered.
- The expand transformation is the expansion of a statement into a single statement or several statements. The generated statement's `SetOfStmtInfo` should be a copy of the original statement's.
- The code movement transformation is the modification of the statement ordering. If the movement is from within to outside of a statement body, the `SetOfStmtInfo` in the statement being moved is augmented with the enclosing statement's `SetOfStmtInfo`. Otherwise, no adjustment is necessary.

3.3 Mapping algorithm

After all optimizations are completed in the compiler, the mapping algorithm is applied to the optimized code and its transformation history recorded in `SetOfStmtInfo`. For each original statement, the algorithm identifies the related original statements and optimized statements by iteratively building a set of original and optimized statements which contain each other. From there, for the loop and conditional cases, the algorithm identifies the "true" loop or "true" conditional among the group of related optimized statements which correspond to the original statement. In the case where the original statement is significantly transformed so that the "true" statement cannot be identified, the statement is

profiled as a region for loops, and is not profiled at all for the conditional case. The algorithm makes conservative assumptions about the code transformations in order to ensure the profile report is correct.

3.4 Instrumentation algorithm

During the instrumentation phase, the profile runtime library routine calls, shown in Table 1, are inserted into the optimized code based on the statement mapping information. Because the mapping phase identifies the related statements and the optimized statements which correspond to the "true" loop or "true" conditional, the insertion of the start and end subroutine calls is a straight-forward process. If the loop or region area is noncontiguous, the contiguous areas are first identified and then instrumented ensuring the invocation count is incremented properly.

4 Profile runtime library

The profile runtime library collects the profile data during execution based on the instrumentation inserted in the user code. In this section, the library interface and internal organization are presented.

4.1 Profile runtime library interface

The main goals in designing the profile interface is modularity and simplicity in order to minimize the code restructuring by the instrumentation module to the optimized code. As described in the previous section, the instrumentation module inserts library subroutine calls, and passes the original source code information via the introduced variables.

The profiling runtime library interface consists of eleven subroutines shown in Table 1, which mark the beginning and end of the program, procedure, loop, and region, and mark the start of the conditional branch. Region is a term for a general looping construct like a FORALL, WHERE statement, array expression, and merged loop. The start and end routines are used to control the counters as well as timers because the timing is collected via starting and stopping timers, as opposed to instruction pointer sampling. The name argument in the start routines points to the original source code description, and the record argument caches the profile record pointer. The value argument for the

Runtime library subroutine
end_loop
end_procedure
end_profile
end_region
incr_loop
start_arith_if(name,record,value)
start_profile(name,record)
start_procedure(name,record)
start_log_if(name,record,value)
start_loop(name,record,entry)
start_region(name,record,entry)

Table 1: Profile runtime library subroutines the program, procedures, regions, loops, and conditionals.

conditional interface is the conditional value, and the entry argument for the loop and region indicates whether the invocation count should be incremented for the profiled construct. Since the end routines and incr_loop operate on the records at the top of the runtime library internal stack, they require no arguments.

4.2 Execution organization

The profiled run is divided into three major phases—initialization, user program execution, and cleanup—which separates the time-sensitive user code from the profile runtime system initialization and cleanup operations. The profile execution commences with a call to start_profile and ends with end_profile, delimiting the phases by the start_procedure and end_procedure calls to the main subroutine.

The initialization phase initializes the internal profile data structures, and starts the timer for the user code. In the user program execution phase, the code is executed while collecting profile data until the call to end_profile. It is during this phase when the profile overhead cost is kept to a minimum, and no communication is performed for profiling purposes. In the cleanup phase, the profile data is summarized across all processors, the statistics are output to a file, and memory is deallocated.

4.3 Minimizing overhead

To implement the profile runtime library subroutines, the runtime system maintains an internal stack, heap, and caller-callee table.

The stack maintains caller procedure information, the heap allocates the profiled statement records, and the caller-callee table manages the procedure information based on the caller-callee pair.

The main goal in the runtime library implementation is to minimize the profile overhead and keep the costs constant in the time-critical phase. More concretely, the goal is achieved by reducing the frequently-performed operations to simple calculations and counter increments, as described in the following three points. The first is manually managing an internal heap, eliminating the unknown cost associated with memory allocation. The second is the underlying implementation of the stack, heap, and table data structures as a table of large pages for dynamic allocation support. The initial page is allocated during initialization, and subsequent pages are allocated as needed. Although the dynamic allocation of pages introduces an unknown overhead cost, the allocation does not occur frequently and the allocation cost is relatively small. Lastly, record lookup is entirely eliminated by caching record pointers in source code variables. The only lookup occurs when locating specific caller-callee records. To reduce this cost, a two-dimensional table indexed by the caller and callee is implemented as a one-dimensional table keeping the lookup cost small and constant.

5 Using the profile data

To complete the profile system description, this section turns towards the components which use the profile data: the profile graphic display tool and the compiler feedback library. The graphic display, unlike other existing ones, guides the user to bottlenecks in the code by graphically displaying (1) the execution time at the procedure and statement level, and (2) the average, standard deviation, minimum and maximum values of the function-based categories of the elapsed time, user code, communication, synchronization, runtime library, and global operations. The standard deviations for each category value across processors indicate the degree of load balancing for each procedure and statement. The profiler viewer, partially shown in Figure 2, can be used in conjunction with a source code browser tool.

The profile data can also used by the compiler to aid in automatic optimizations based on the collected data. The feedback library reads the profile report and relates the information to the original source code.

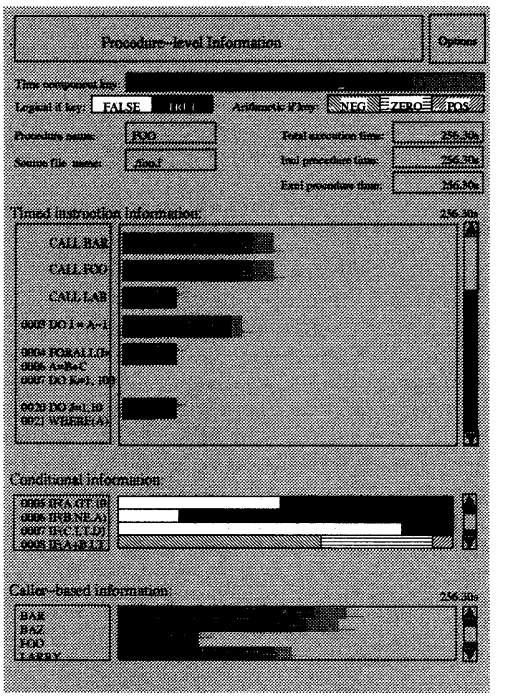

Figure 2: The timing and conditional profile information for a procedure is graphically displayed to the user.

6 Perturbance measurement

To measure the overhead introduced by the profile runtime library, the execution time for the Livermore kernel7 and SPEC CFP92 tomcatv and swm256 benchmark programs were collected on the Fujitsu AP1000, a parallel, distributed-memory machine. The benchmark versions in Table 2 are similar to the original benchmark but differ in parallel annotations and data distribution.

The data in Table 2 refers to three execution types, *NoProf*, *TimeLib*, and *Prof*. The *NoProf* version is the basic execution with no data collection. The *TimeLib* version is *NoProf* with the maintenance of several timers, categorizing the elapsed time based on the performed functions (e.g. user code, communication, synchronization, runtime library, and global operation). The *Prof* version is implemented on top of *TimeLib*, collecting the

Benchmark	*Prof*(s)	$\frac{Prof}{NoProf}$	$\frac{Prof}{TimeLib}$
kernel7.auto	1.197	1.008	1.004
swm256.1dim1	10.398	1.136	1.035
swm256.1dim2	5.412	1.149	1.022
tomcatv.hpf	203.735	1.236	1.111
tomcatv.intrin	10.577	1.274	1.064

Table 2: Elapsed time on the AP1000 for benchmarks without profiling, with timing library, with profiling excluding summary, and profiling.

profile data using the *TimeLib* timers.

The benchmarks were executed on a 16 node configuration AP1000. Table 2 shows the raw execution time for the *Prof* version, and the profile overhead with respect to the *NoProf* and *TimeLib* versions. Since *Prof* uses the timers maintained by *TimeLib*, the $\frac{Prof}{TimeLib}$ ratio provides an accurate cost of the profile library overhead. The overhead is small, ranging from 0.4% to 11.1% of the *TimeLib* execution. The variation is a function of the presence of profiled constructs in the program. When considering the profiling system as a whole, the $\frac{Prof}{NoProf}$ ratio is more appropriate. Although the only difference between the *Prof* and *TimeLib* is the time function calls and counter increments, the *TimeLib* overhead is significant because the time function implementation on the AP1000 is expensive.

7 Related work

This section briefly discusses the related work in instrumenting optimized code, runtime library organization, and profile data viewer application.

In many implementations, compiler optimizations are disabled when profiling to simplify instrumentation and eliminate the mapping between the original code and optimized code. Since there is little literature on profiling optimized code, literature in debugging optimized code, which also requires the mapping between the original and optimized code, was considered. In [1], the compiler maintains a detailed history of each code transformation which is later processed and interpreted in the debugger. This system is based on a simpler version of this approach.

The runtime library interface, organization, and basic data structures are similar to other existing implementations[3][4], but differ in the data collected and caller-callee pair maintenance. Existing systems report processor averages of the elapsed time and function-based classification, and allocate the maximum number of caller-callee pairs during initialization based on processing of the object code, whereas this system collects additional information and maintains the caller-callee pairs dynamically.

Several text and graphic-based profile data viewers are currently available, APR Forge90[2], and Fujitsu F90 Workbench. This system's viewer application adopts and improves the best features from those tools.

8 Summary

This paper presents an overview of a profiling system for an HPF compiler which aids the user in understanding the behavior of programs and locating bottlenecks. The system consists of an optimized code instrumentation module, runtime library, profile data graphic display, and compiler feedback module. Each module was successfully designed and implemented satisfying the initial functional and performance goals. Performance data collected for this prototype shows the overhead of the profile runtime library is relatively small, ranging from 0.4% to 11.1% of the non-profiled execution time.

9 Acknowledgments

The authors would like to thank members of the compiler group—Mr. Iwashita, Mr. Doi, Mr. Hagiwara, Mr. Taguchi, and Mr. Miyoshi—for their fire fighting efforts. A special thanks goes to Dr. Ishii, Mr. Shiraishi, and Mr. Ikesaka for their continued support.

References

[1] G. Brooks, G. Hansen, and S. Simmons. "A New Approach to Debugging Optimized Code," in SIGPLAN '92 Conf. Programming Language Design and Implementation, pp 1-11, 1992.

[2] FORGE Explorer User's Guide, Version 1.2, Applied Parallel Research, May 1994.

[3] S. Graham, P. Kessler, and M. McKusick. "gprof: a Call Graph Execution Profiler," in SIGPLAN '82 Symposium on Compiler Construction, pp 120-126, June 1982.

[4] C. Ponder, and R. Fateman. "Inaccuracies in Program Profilers," Software-Practice and Experience, 18(5), 459-467, May 1988.