# Optimization of Ada'95 Tasking Constructs

Dong-Ik Oh T.P. Baker
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019
Internet: {doh,baker}@cs.fsu.edu Phone: (904)644-5452

## ABSTRACT

The 1995 revision of the Ada programming language added several new features related to multi-tasking. These new features open up new implementation questions. In particular, it is appropriate to revisit the subject of compiler and runtime system optimization techniques for Ada multitasking. The existence of GNAT provides an opportunity to study these problems, as well as some more general and longer-standing questions. We study two areas of optimization. The first area is reduction in the number of operating system service calls. We concentrate on the use of different locking policies which control the degree of parallelism within the runtime system. The second area of optimization is to restrict the GNARL implementation by implementing the new pragma `Restrictions`. In particular, we produce a restricted RTS which excludes task abortion and the asynchronous transfer of control. Performance comparisons between different versions of GNARL and ways of managing them are also provided.

## 1 INTRODUCTION

Much of the pioneering work on the implementation of Ada'95 multitasking was done by the GNAT tasking team, at the Florida State University[3, 1, 2, 10]. The original GNAT tasking implementation was primarily viewed as a research and instructional tool. Runtime efficiency was considered less important than semantic correctness, completeness, and portability. However, now that these other goals have been largely met, and people begin to use GNAT as a production compiler, it is time to pay more attention to efficiency.

The objective of the current study is to discover and evaluate techniques for "optimizing" the implementation of Ada multi-tasking constructs, to improve their execution time performance.

Most of the research that has been done on optimization in programming language implementation has focussed on the back-end of the translation process, which is largely language-independent. This kind of optimization has been studied extensively, and a considerable body of knowledge has been developed, but it does not apply to the present study.

Multi-tasking constructs are generally implemented by a combination of high-level source code transformations, in the compiler's front-end, and calls to Ada runtime library routines, which may in turn make operating system (OS) service calls. Thus, the opportunities for optimization involve alternate source-code transformations, and alternate algorithms and data structures in the runtime library routines.

While some prior research has been done on optimization of Ada tasking constructs[7, 4, 5], it was done in the context of the Ada'83 language. Since then, much has changed. Ada'95 made several additions to the multitasking model and added a new pragma that allows for "subset" implementations. Another large change has been the introduction of support for multi-threaded processes in commercial operating systems. These changes in the "rules of the game" for task optimization mean that optimizations that made sense for Ada'83 might be less valuable or even incorrect for Ada'95, and there may be new optimizations that were not considered for Ada'83 that might pay off well for Ada'95.

In this study, we implement and test two different kinds of optimizations, both of which may involve the use of alternate runtime library configurations. The first approach is directed toward reducing the number of operating system service calls. The second approach is directed toward reducing the runtime system (RTS) features, implementing new Ada'95 pragma `Restrictions`, particularly reducing the distributed overhead of task abortion and the new asynchronous transfer of control (ATC) feature. We discuss several ways one can manage different versions of the RTS, hence making the use of optimized versions of RTS practical.

## 2 Optimization by Reducing System Calls

The GNAT Ada Runtime Library (GNARL) is implemented as a layer over the top of threads layer, as is shown in Figure 1. The GNARL layer implements the specific semantics of Ada tasks. It is independent of the machine architecture and operating system. It pro-

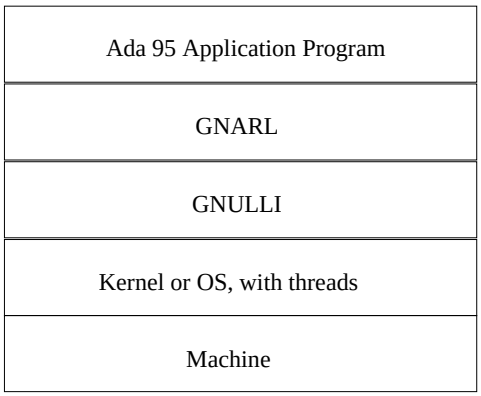


Figure 1: GNARL components

vides services that are called by the compiler-generated code to implement the semantics of Ada tasking. These are high-level services, to which the syntactic structures that relate to tasking can be mapped very directly by the compiler. The next lower layer, called GNULLI exists only for portability; it provides a standard interface to services that are typically provided by an operating system or real-time kernel, isolating dependences on a particular host from the rest of GNARL. The original implementation of this lower level uses the POSIX interfaces[11, 12], but it has been ported to several other non-POSIX standard systems as well.

Since GNARL is layered on top of an OS and usually an OS system call is expensive, we performed timing measurements on various GNULLI-supported system calls to see their runtime costs. By doing this we identified several places where reduction of system calls could pay off.

The cost of a system call depends on the particular OS and threads library to which GNARL is ported. Therefore, we conducted timing measurements on three different targets. The first is FSU Threads[8] on SunOS (version 4.1.4). The second is Solaris (version 2.4) using its own native threads implementation. The third is FSU Threads on Solaris 2.4.

We found that the `Clock`, `Signal_Masking` and `Wakeup` operation are expensive for all targets. `Self`, `Set_Priority`, `Yield` and `Lock/Unlock` operations are also expensive on some targets. Efforts to reduce the number of system calls mentioned above have been conducted previously. Related GNARL execution improvements are reported in [9]. However, there are other kinds of system call reduction which can be beneficial only under certain circumstances. Reduction of `Lock/Unlock` operations used in GNARL is an example of such optimization.

GNARL uses `Lock/Unlock` operations in order to maintain data consistency under concurrent read/update operations by multiple threads of control. It does quite a few more `Lock/Unlock` operations than is typical of older Ada runtime systems. The difference is that GNARL was designed to be multi-threaded, whereas most earlier Ada runtime systems were designed as a monolithic monitor[6]. That is, the older style of Ada runtime system only allowed one task to be executing in the RTS at a time (we call this single-lock mode), but with GNARL several tasks may be executing in the RTS concurrently. Rather than just one lock that protects the entire RTS, there are individual locks for several RTS global data structures, and a lock for each task control block (we call this multiple-lock mode). Multiple-lock mode allows more concurrency between tasks. According to conventional wisdom, more concurrency is generally better. It permits more parallel execution if there are multiple processors, and even if there is only one processor it may permit quicker response to high-priority real-time events.

However, there is a trade-off here, between concurrency and overhead, and it may well depend on both the application and the environment on which GNARL is executed. If the parallelism in GNARL is not needed we may be able to eliminate a lot of `Lock/Unlock` operations, which may give us noticeable performance improvement. As a first step toward gaining a better empirical understanding of this trade-off, we produced a modified GNARL implementation which uses only a single lock to protect all the RTS data. We performed performance tests of the two versions, varying both the number of processors and the degree of task interaction, to obtain empirical guidelines for when each of the two locking policies is likely to be best.

The number of locking operations we save on a uniprocessor depends on the GNARL call being invoked. For many calls, the multiple-lock mode involves one to three `Lock/Unlock` pairs. For a termination related operation the number of locking pairs required is around ten. For task activations, we need to lock all the Ada control block of tasks being activated. Thus, we would expect to get better performance improvement on the termination related tasking features using the single-lock mode.

In a multiprocessor system, there are situations in which the multiple-lock mode may not payoff. If by the nature of a program, only a single thread of control dominates the execution, we will not see a performance benefit using the multiple-lock mode. In theory, if there are more than one task attempts to execute in the runtime system at the same time and they do not need to access the same data structures we expect the use of multiple-lock mode to pay off. However, this may not be true on some targets.

*In the full version of this paper we will include data*

*showing the performance improvement using the single-lock mode. Also, we will provide information on what circumstances the use of multiple-lock mode is beneficial.*

## 3 Optimization Using pragma Restrictions

Reducing the number of system calls may give us considerable RTS performance improvement. However, there must be a limit to what can be done with this kind of optimization. Therefore, we consider another approach to improve performance, by taking advantage of RTS restrictions.

Ada'95 introduced the new pragma, named `Restrictions` (ARM 13.12), to address the desire of some users for a simpler language. The intention was that where there is a need for greater efficiency or trust, users and implementors could agree on a language subset. This new pragma represents a departure from Ada'83, and one of the original policies behind the Ada language, which was to reduce the proliferation of languages and dialects used within the U.S. Department of Defense. During the language design process, there was some controversy about the wisdom of allowing explicit "subsetting" in Ada'95. This resistance was partially overcome by recognition that it would otherwise not be possible to reconcile the demands for new features from some quarters of the user community with the demands for simplification from other quarters. The fear of dialects was also allayed by the fact that each compiler is still required to support the entire language, for source code that does not use the `Restrictions` pragma, and that it is required to check that the source code does adhere to the restrictions.

The Ada Reference Manual suggests several tasking features whose usage may be restricted, but these are just suggestions. An implementor is free to recognize and take advantage of these, or any other restrictions that the implementor defines, to allow a simpler or more efficient runtime system. Clearly, it is impractical to try to provide a tailored runtime system for each possible combination of restrictions that a user might specify. To avoid this combinatorial explosion, the implementor must identify a few combinations of restrictions that produce a large improvement in performance (and whatever other criteria are important, such as RTS simplicity, analyzability, and timing predictability). The reduction in expressive power of these restrictions must also be acceptable to users.

We examined the asynchronous select statement and the task abort statement, a pair of very closely related language features, for candidates to be placed in the `Restrictions` pragma. We picked these to be the first features to be restricted since we had observed that there are bits of code that are only needed to support the implementation of ATC and task abortion, scattered throughout the entire GNAT runtime system. These features are expensive since the notification of abortion or call cancellation through ATC is implemented using POSIX signals and Ada exceptions, with which we need to perform context saving/restoring operations. Worse, the call/cancellation can be nested and many context specific information need to be managed throughout the GNARL implementation. In addition, GNARL uses call/cancellation mechanism of ATC in order to implement other language features, namely `Timed_Entry_Call` and `Selective_Wait_With_Timeouts`.

We have prepared a version of GNARL which does not support task abortion or ATC. This simplifies the logic in several places. Especially we do not need to wait or check for pending aborts.

*In the full version of this paper we will provide performance comparisons for various tasking features and show what the distributed overhead imposed for supporting task abortion and ATC is. The tests will include PIWG and ACES tests on various tasking features.*

## 4 Configuration Management

In order to take advantages of both of the two optimization approaches mentioned in previous sections we would like to include them in the standard GNAT compiler distribution. That presents a logistic and a management problem. We need to find an efficient way to manage them while maintaining the performance improvement of the optimizations we have achieved. In particular, some method will need to be designed to allow user selection among the four different RTS combinations:

1. multiple-lock mode, with no restrictions
2. single-lock mode, with no restrictions
3. multiple-lock mode, with no ATC or task abort
4. single-lock mode, with no ATC or task abort

To achieve the maximum performance benefit, each of these optimizations would rely on compile-time or link-time selection of an alternate RTS configuration. For example, it would be possible to control whether single-lock or multiple-lock mode is used in the RTS via a software flag and if-statements, but testing the flag at each potential locking point would be an addition to the execution time overhead of the RTS. To get the maximum benefit, the decision as to granularity of locking must be made prior to compilation of the runtime library. The need for compile-time specialization of the

RTS code is even clearer for the other kind optimization we are considering here.

*In the full version of this paper we will provide pros and cons of various methods for maintaining different versions of GNARL. The methods will include:*

- *Pre-processing of the source code.*
- *Use of optimizable Ada if-statements with static conditions.*
- *Use of dynamic if-statements.*
- *Use of "Soft links".*
- *Alternate versions of RTS source files.*

## References


[1] E. W. Giering and T. P. Baker. The GNU Ada runtime library (GNARL): Design and implementation. In *Wadas '94 Proceedings*, 1994.

[2] E. W. Giering and T. P. Baker. Implementing Ada protected objects—interface issues and optimization. In *Tri-Ada '95 Proceedings*, pages 134–143, Anaheim, California, 1995.

[3] E. W. Giering, F. Mueller, and T. P. Baker. Implementing Ada 9X features using POSIX threads: Design issues. In *Tri-Ada '93 Proceedings*, pages 214–228, Seattle, Washington, 1993.

[4] A. N. Habermann and I. R. Nassi. Efficient implementation of Ada tasks. Technical report, Carnegie Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania, 1980.

[5] P. N. Hilfinger. Implementation strategies for Ada tasking idioms. In *AdaTEC Conference*, pages 26–30, Arlington, Virginia, 1982.

[6] C. A. R. Hoare. An operating system structuring concept. *CACM*, 17(10):549–557, October 1974.

[7] A. Jones and A. Ardo. Comparative efficiency of different implementations of the Ada rendezvous. In *AdaTEC Conference*, pages 212–223, Arlington, Virginia, 1982.

[8] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, 1993.

[9] D. Oh and T. P. Baker. Gnu Ada'95 tasking implementation: Real-time features and optimization. To appear in the proceedings of the 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS), 1997.

[10] D. Oh, T. P. Baker, and S. Moon. The GNARL implementation of POSIX/Ada signal services. In *Ada-Europe '96 Proceedings*, pages 276–286, Springer, Verlag, 1996.

[11] Portable Application Standards Committee of the IEEE. *Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language]*, 1990. IEEE Std 1003.1-1990.

[12] Portable Application Standards Committee of the IEEE. *Draft Standard for Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) – Amendment 2: Threads Extension [C Language]*, 1994. IEEE Std 1003.1c-1994.