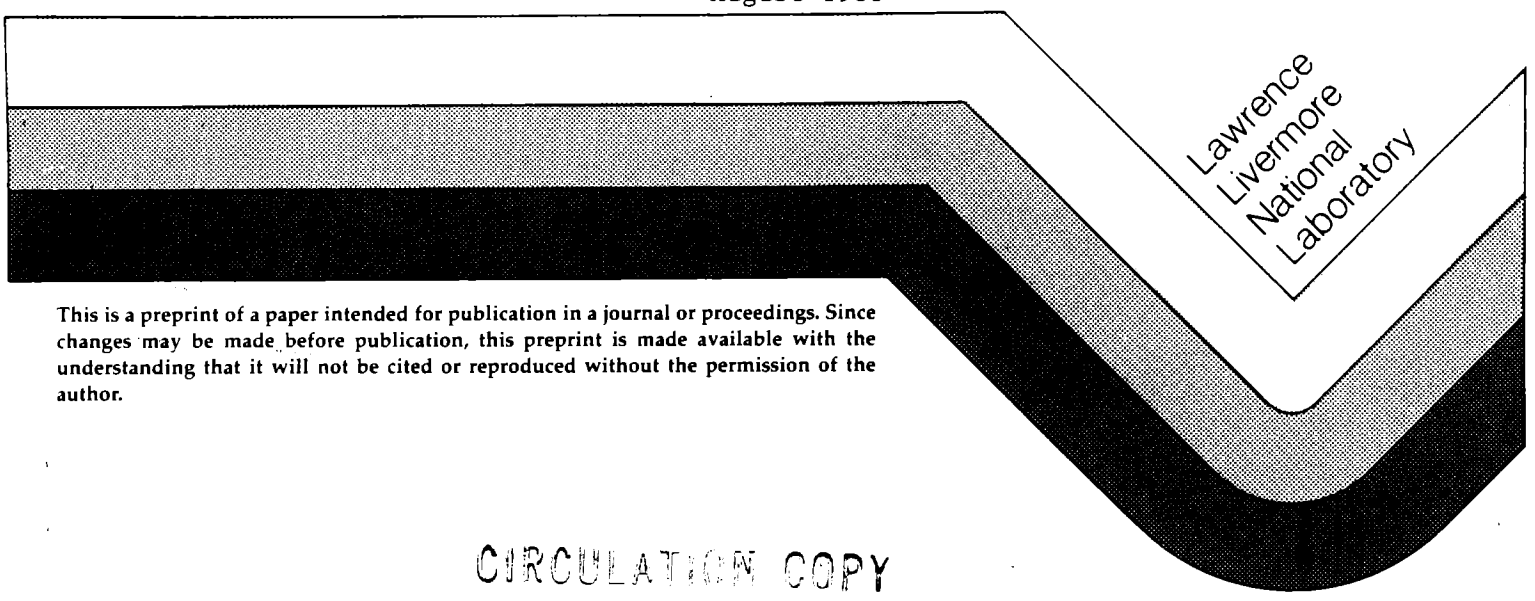
UCRL- 99708
PREPRINT

Sisal Implementation and Performance

Stephen Skedzielewski
Lawrence Livermore National Laboratory
Livermore, California

This paper was prepared for submittal to
IFIP Working Group 2.5
Workshop 5
Stanford, California
August 25, 1988

August 1988



Sisal Implementation and Performance

Stephen Skedzielewski *

Computing Research Group
Lawrence Livermore National Laboratory

Abstract

This paper discusses the effect of several code-improvement techniques on programs written in the SISAL [MSA*85] language. SISAL is an *applicative* language, and the debate about the efficient implementation of such languages has continued for several years[MKW84]. We discuss several sources of inefficiency in the current SISAL compilation and runtime system, and extrapolate the effect of new code improvements and code generators from small examples that we are running in the next generation of SISAL compilers.

1 Introduction

The SISAL project comprises language definition, and the design and development of compilers and runtime systems for existing multiprocessors. Evaluation of the project based on results from an intermediate code interpreter [SYO87], a Sequent Balance implementation [OC88], as well as other multiprocessors [LSF88] have appeared in the literature. In this paper we outline several of code improvement techniques that we are currently developing for the next generation of SISAL compilers, and the effect that we think they will have on the quality of code that we generate.

2 Code Improvers

We implemented several code improvement algorithms since the first release of the compiler. They attack the problems of array storage management and program partitioning.

*This work was supported (in part) by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

2.1 Reference Counting

The first release of the compiler performed naïve reference counting; each reference to a structured (non-scalar) object entered a critical region to adjust the reference count of the object. Since entering and leaving critical regions creates a bottleneck for any other users of the object, and since it can be expensive to implement critical regions on some parallel processors, we found that we were losing a significant amount of performance due to this code generation technique.

We have implemented a static reference count analyzer that removes a large number of reference counting operations from the compiled code. without reducing the parallelism available in a SISAL program.

On nine benchmarks ranging from simple sorting routines to a 1500 line hydrodynamics code, we removed approximately 65% of the reference counting operations. These numbers are static counts, but they lead us to believe that we will gain back much of the reference counting overhead in our next release.

2.2 Build-in-Place

A common complaint about applicative languages is that they are inefficient in array operations, especially because they seem to require excessive array copying in order to build up large arrays. In order to attack this problem, we tried to find opportunities where we could preallocate storage in one chunk, and then allow the parallel processors to fill in the subarrays independently [Ran87]. This analysis was very successful in the Simple program [CHR78]. The analysis completely avoided any array copying in the Node_Reflect function, which builds the boundary conditions for the grid.

Similar boundary condition problems occur in most grid problems, so we think that this technique will be very helpful in reducing the need to copy arrays at runtime. This analysis also preallocates memory in other situations, such as iterative loops that run over simple integer ranges, but we haven't yet determined how much gain we can expect from this analysis.

3 Minimal Reevaluation

As part of his doctoral research at U.C. Davis, David Zimmerman investigated the utility of capturing just the last result of a function execution, and saving it for possible reuse later. This method, which he called *minimal reevaluation* is similar to the technique called *memoization* in functional programming, but different. Zimmerman's technique (MR) only captures the *last* value computed, where memoization usually tried to build a table of many or all previous results. MR also works for any set of inputs to an operation, while other techniques often limit themselves to either functions of small numbers of scalars, or require the user to provide a mapping function from the actual inputs to a scalar domain.

The work found that even in highly numerical codes such as Simple, it was profitible to use minimal reevaluation to decrease the runtime of the code. These experiments were performed using an interpreter, rather than native code, so we hope to produce an experimental compiler that can incorporate MR techniques into native code.

4 Partitioning

The current runtime system extracts parallelism in the following ways:

1. All function calls are spawned
2. Producers and comsumers of streams are spawned as separate processes
3. The user may request that the outermost loops in a function be sliced. However, either all, or none, of the loops will be sliced

This method has worked remarkably well, giving good speedups on several benchmark programs (including a monte-carlo photon transport code). However, a bit of analysis can improve performance nicely.

We have implemented an analyzer that repackage a program to try to fit it to the architecture of a given machine [SSM88]. For example, it will spawn only those functions that it feels are big enough to overcome the overhead of the spawning. It automatically inlines functions, including recursive functions, until it builds up a large enough function to be worth spawning. It slices only those loops that seem large enough, and the algorithm will vary the size of a loop slice, depending on the size of the loop body.

The results of this analysis have shown roughly a factor of two improvement over the straightforward code generation that exists in the first release of the compiler.

5 New Code Generators

The current code generator for SISAL produces C code that is linked to a machine-specific runtime library. This library performs all of the task spawning and synchronization that is needed to automatically run SISAL programs on a parallel processor. Unfortunately, we must use whatever C compilers are given to us, and they often are not very good. Therefore, we are beginning a new code generator effort that has two goals: to produce C code that even "dumb" compilers can compile well, and to produce assembly language code so that we can control crucial items such as register allocation and code inlining.

The new code generator is giving us roughly two- to three-fold improvement over previous compilations of codes that we've tested on the VAX and the SUN-3.

6 Conclusions

Analyzing the behavior of the code that we generate for SISAL programs has helped us identify very fruitful areas of research into code generation for applicative languages. We see the difference in performance between conventional languages like FORTRAN and SISAL shrinking (even on uniprocessors) and expect to generate code that will be hard to beat using "parallel" extensions to conventional languages.

References

[CHR78] W. P. Crowley, C. P. Henderson, and T. E. Rudy. *The Simple Code.* Technical Report UCID 17715, Lawrence Livermore National Laboratory, Livermore, CA, Februrary 1978.

[LSF88] C.C. Lee, S.K. Skedzielewski, and J.T. Feo. On the implementation of applicative languages on shared-memory, MIMD multiprocessors. In *Proceedings of the ACM conference on Parallel Processing: Experience, Applications and Languages*, pages 188–197, July 1988.

[MKW84] James R. McGraw, David J. Kuck, and Michael Wolfe. A debate: retire Fortran? *Physics Today*, 37(5):66–75, May 1984.

[MSA*85] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. SISAL: *Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2.* Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[OC88] R. R. Oldehoeft and D. C. Cann. Applicative parallelism on a shared memory multiprocessor. *IEEE Software*, 5(1):62–70, January 1988.

[Ran87] John E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, University of California at Davis, Computer Science Department, Davis, California, 1987.

[SSM88] Vivek Sarkar, Stephen Skedzielewski, and Patrick Miller. An automatically partitioning compiler for Sisal. In *Proceedings of CONPAR'88*, Manchester, England, September 1988. (to appear).

[SYO87] S. K. Skedzielewski, R. K. Yates, and R. R. Oldehoeft. DI: an interactive debugging interpreter for applicative languages. In *Proceedings of the ACM SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques*, pages 102–109, June 1987.