Array Optimizations in OCaml

Michael Clarkson
Cornell University
clarkson@cs.cornell.edu

Vaibhav Vaish
Cornell University
vaibhav@cs.cornell.edu

May 7, 2001

Abstract

OCaml is a modern programming language with many advanced features, yet its strict sequential semantics and consequent lack of support for locality-enhancement optimizations on arrays prohibit its use in numerically intensive computation. In this paper we show that it is possible to perform such optimizations by creating safe regions in which array bounds checking is unnecessary. These optimizations can be performed (mostly) automatically by a prototype implementation that we describe. Performance gains from the optimizations range from 37-94% on simple matrix kernels.

1 Introduction

OCaml is a functional programming language with support for most constructs found in modern programming languages. Apart from providing first-class functions, it supports polymorphic type inference, recursive types, objects, and imperative constructs such as references, arrays, and for and while loops. OCaml code may be compiled to platform independent bytecode, or to native code (which is faster, but not portable). It would be highly desirable to have an OCaml compiler which could produce code that performs competitively against C or Fortran programs, thereby giving the programmer useful high-level constructs in the language without sacrificing performance.

High performance compilers apply loop transformations, such as tiling, to enhance locality and reduce cache misses. These optimizations are legal in C and Fortran, as the program behavior is undefined when an array access violates the array bounds. However, OCaml has a semantics that specifies that each array access is preceded by a bounds check, and an exception is raised if there is a violation. Consequently, loop transformations may violate the semantics by reordering the traversal of the loop iteration space. Thus loop transformations are generally illegal in OCaml.

This problem was addressed by Moreira et. al. [2] for Java, which suffers from the same defect. The idea is to find regions of the loop iteration space that are provably safe – i.e., in which no array bounds violation can occur. The loop nest can then be decomposed into safe and unsafe regions. We can disable bounds checking for the safe regions, and apply loop transforms legally. In the subsequent sections, we describe how we implemented this idea in the OCaml compiler.

2 Guard generation

2.1 Arrays in OCaml

OCaml's standard library has a type `'a Array`, which represents unidimensional arrays of any OCaml type. Multidimensional arrays are represented as arrays of arrays. Bounds checking for these arrays may be disabled at compile time by compiling with the `-unsafe` option. These arrays have a maximum size limit (2M of `floats`). Programs that require large, multidimensional arrays should instead use the Bigarray library. Implemented almost entirely in C, this provides support for arbitrarily sized arrays, up to 16 dimensions for various precisions of integers and floats. The programmer may choose a C-style or Fortran-style layout for each array instance. In our implementation, we considered array accesses only for arrays of the Bigarray library. However, we report a comparison of running times of the same algorithms using the arrays of the standard library.

2.2 Moreira's Approach

Moreira et. al. assumed that each array access $A[f(i)]$ was an affine function of a single loop variable: $f(i) = ci + d$, where $c > 0$, and c and d are loop invariant. This enabled them to determine the range of values of i for which the access is safe: $\frac{lo(A)-d}{c} \leq i \leq \frac{hi(A)-d}{c}$, where $lo(A), hi(A)$ are the lower and upper bounds of the array A. They describe a procedure to recursively tile a loop nest at each level, starting from the outermost, into safe and unsafe regions. They also consider parallelizing the loop execution and thread safety.

2.3 Our Implementation

We wanted to consider the more general case, when array accesses may be affine functions of several variables. Since we no longer have closed-form expressions for the bounds of the safe region, recursively tiling the loop nest at each level becomes significantly harder. We took an alternative approach: we computed a predicate whose evaluation would determine whether an array bounds violation could occur anywhere within the loop nest. Thus, rather than tile the loop into safe and unsafe regions, we test the entire nest for safety. The generation of the guarding predicate is done at compile time using an integer linear programming (ILP) calculator called Omega[1]. The guard consists of affine inequalities in terms of array sizes and loop invariants. These can be evaluated efficiently at runtime when the actual array sizes are known. We generate an optimized

Figure 1: MVM Kernel in OCaml

```
(* Matrix-vector multiplication: perfect loop nest. c = c+a*b *)
let mvm c a b =
  let rows = Array1.dim c and
      cols = dim2 a in
    for i = 0 to rows-1 do
      for j = 0 to cols-1 do
        c.{i} <- c.{i} +. a.{i,j} *. b.{j}
      done;
    done
```

version of the loop nest at compile time. At runtime, the guard is evaluated. If the loop nest is deemed unsafe, it is executed without optimization and with bounds checking. If the loop nest is safe, the optimized version is executed instead.

2.4 Interfacing with Omega

A significant part of our effort went into building the interface between OCaml and the Omega library. We wrote an OCaml module to interface with Omega and have inserted it as part of the standard library. To compute the guard, we define within Omega the set of points in the iteration space which violate at least one array bound. We then project out the loop variables to existential quantifiers and obtain a set of inequalities which constitutes precisely the guard we want. Our interface lets us specify the constraints to construct the set and retrieve the guard within OCaml. As an example, we include an MVM kernel in Figure 2.4 and its computed guard in Figure 2.4.

3 Code generation

There were three tasks that needed performed in order to generate code for array optimizations: identify where to insert the optimization pass in the OCaml compiler, identify what loops would be targeted by the pass, and perform the transformation on these loop nests.

3.1 The OCaml compiler

The OCaml compiler is structured like most modern compilers. It uses automated lexing and parsing tools to build an abstract syntax tree (AST), performs semantic analysis upon the AST, converts the AST to an intermediate representation (IR), then generates code. The compiler can generate either bytecode for an architecture independent virtual machine, or it can generate native code for several processors.

Figure 2: Computed guard for MVM kernel

```
if ( ((-1  + 1*cols -1*(Bigarray.Array1.dim b ) >= 0 ) &&
(-1  + 1*rows >= 0 ) &&
(-1  + 1*cols >= 0 )) or
((-1  + 1*cols -1*(Bigarray.Array2.dim2 a ) >= 0 ) &&
(-1  + 1*rows >= 0 ) &&
(-1  + 1*cols >= 0 )) or
((-1  + 1*cols >= 0 ) &&
(-1  + 1*rows -1*(Bigarray.Array2.dim1 a ) >= 0 ) &&
(-1  + 1*rows >= 0 )) or
((-1  + 1*cols >= 0 ) &&
(-1  + 1*rows -1*(Bigarray.Array1.dim c ) >= 0 ) &&
(-1  + 1*rows >= 0 )) or
((-1  + 1*cols >= 0 ) &&
(-1  + 1*rows -1*(Bigarray.Array1.dim c ) >= 0 ) &&
(-1  + 1*rows >= 0 )) ) then
(* Orig loop *)
 else
 (* Optimized loop *)
```

Array optimizations are easiest to perform when both array accesses and the for loops containing them are still present in the compiler's representation of the program. This is the case for the IR used by the OCaml compiler, which is a data structure called Lambda. The Lambda tree is generated from the type-checked AST, transformed to an equivalent Lambda tree by a simplification pass, and then either bytecode or native code is generated from it. Thus, the array optimization pass was inserted between the simplification and code generation passes. It rewrites the Lambda tree to an equivalent tree with optimized loops nests. A benefit of inserting the optimization pass at this point is that both bytecode and native code can take advantage of the optimizations that are performed.

3.2 Identification of optimizable loops

Identification of loops in the Lambda tree is an easy task, since one of the node types in the tree is a for loop node called Lfor. The real issue is identifying loops that meet the assumptions that the guard generation process makes. This requires a fairly simple recursive traversal of the tree to gather information about what type of code each loop nest contains. The array optimization pass doesn't attempt to transform any for loop nest that is not perfectly nested, contains complicated functional structures such as letrec or lambda, or contains array accesses that are not affine functions of loop indices and loop invariant variables.

3.3 Transformation to optimized loop nests

The goal of the transformation stage of the array optimization pass was to rewrite each loop nest into an optimized version. At a high level, this meant performing the following transformation:

```
for(index, lo, hi)
      ...
```

Figure 3: Original code

```
if (safe region guard) then
      optimized for nest
else
      original for nest
```

Figure 4: Transformed code

Several choices were available for implementing this transformation, though only one was realistic given time constraints. These choices are described below.

3.3.1 Transformation library and interface

First, a transform library was needed in order to perform loop transformations such as tiling, skewing, height reduction, etc. The ideal situation would be to have an OCaml library for doing this that was part of the OCaml compiler. No such library exists, however, so the SUIF compiler framework [3] was chosen as the next-best alternative.

The next issue was how to interface with SUIF, which works primarily with C (though extensions for other languages are available). SUIF had to be able to get code from OCaml to optimize, and OCaml had to be able to get the optimized results from SUIF. These problems could both have been solved elegantly if an OCaml front-end were available for the SUIF compiler, or a C front-end were available for the OCaml compiler. Since neither exists, however, a hybrid OCaml-C solution was necessary.

The final implementation generates C source code that is equivalent to the OCaml source for each loop nest that is to be optimized. The OCaml source is rewritten to make an external function call to the C source in order to execute the optimized version of the loop nest. The C source can then be independently optimized by the OCaml compiler. The OCaml and C object files are then linked to make a single executable.

3.3.2 Problems encountered

Two problems were encountered in generating the C source code. At a high level, it is a standard problem: a syntax-directed translation of one tree into another. However, types and linking made it somewhat more difficult.

OCaml is a statically typed language, just as C, but it uses type inference rather than requiring the programmer to write type annotations. The OCaml compiler does not preserve this type information in the Lambda tree, yet the C source generated from it must declare types. Solving this problem required a very small dataflow analysis, structured as a single traversal over the tree, to conservatively determine types for all of the variables that appear in the Lambda tree. All variables are initially assigned an unknown type. When a variable is used as a loop or array index it is promoted to an integer. Any variable used as an array base is assumed to be an array of doubles (possibly multi-dimensional). Finally, any other variable is assumed to be a double used in intermediate calculations.

Array dimensions were an important subproblem of typing. C requires that the sizes of all but the leftmost dimension of an array be declared. However, OCaml arrays are dynamically sized, so it is not possible to determine the size of an array at compile time. As an interim solution, the implementation requires that the programmer manually edit the generated C source code to insert these array sizes.

Linking was problematic because the OCaml/C interface differs both for the bytecode and native compilers, and for calling C functions that take more or less than five arguments. In addition, the arguments passed to the C function do not have native C types, rather, they have types declared in an include file from the OCaml compiler. The implementation generates a stub entry function that unpacks arguments from these types to their native C types, then calls a function containing the optimized loop nest. Currently, the implementation assumes the bytecode compiler is being run on functions passing at least five arguments. This requires manual insertion of dummy arguments if the function does not already take five, but could be automated.

Since the external call must be inserted manually, and the program recompiled, we do not insert the guard in the Lambda IR, but print it to standard output along in OCaml syntax. In our current implementation, the programmer inserts the guard and external call and declaration manually, and recompiles.

4 Results

The experiments described below were performed on a workstation running Windows 2000. It contained a Pentium III 667Mhz processor with a 16Kb L1 cache and a 256Kb L2 cache. Version 3.01 of the OCaml compiler was used, compiled using Cygwin. The C compiler used was gcc version 2.95.3-1. The SUIF compiler used was version 1.3.0.5.

4.1 Baseline OCaml compiler performance

As a baseline, we first established the performance of the unmodified OCaml compiler. We considered three versions of an MMM kernel: one written using the BigArray library, another using a simulated 2D array created with the Array library, and another using a 1D array with linearized accesses[1], again with the Array library. The OCaml compiler allows array bounds checking to be disabled,

[1]By linearized accesses, we mean the each array reference of the form A[i][j] is transformed to A[N*i+j].

Table 1: Unmodified OCaml Performance

	Safe	Unsafe
BigArray	55.22	55.21
2D Array	8.212	5.989
1D Array	5.358	3.896

Time to execute MMM kernel in seconds, N = 400

so we considered versions with array accesses both enabled ("safe") and disabled ("unsafe"). Also, we show the performance of only the native code compiler, since the bytecode compiler is in all cases much slower. The performance of the compiled kernels is given in Table 4.1.

The BigArray library clearly introduces a very large overhead. The library does not even provide an "unsafe" get or set operation, thus disabling array bounds checking has no effect on performance. The Array library performs much more reasonably. Disabling bounds checking results in a 27% speedup for the Array library.

4.2 Optimizer performance

To measure the performance of our optimization pass, we tested it on four matrix kernels: a 5-pointed system, a 9-pointed system, MVM, and MMM. C code for each of these is given in Figure 4.2.

For each of these, we ran three tests:

1. The unmodified native OCaml compiler. (ML)
2. Our optimizer. (OPT)
3. Our optimizer, plus a SUIF pass to do tiling and height reduction on the generated C code. (OPT+SUIF)

For the latter two tests we used the bytecode OCaml compiler, rather than the native code compiler, due to linking issues. The performance of the kernels is given in Table 4.2.

For the 5-pointed and 9-pointed kernels, our optimizer provided a 73% speedup, which the SUIF pass was unable to improve. For MVM, our optimizer provided a 38% speedup, which the SUIF pass improved only negligibly.[2] The most dramatic performance improvement is seen on the MMM kernel. Our optimizer provides a 87% speedup, which the SUIF pass is able to improve another 50%, for a total speedup of 94%! The final performance of MMM surpasses even that of the best-case baseline kernel – the linearized 1D Array with bounds checking disabled – yet without sacrificing the safety of bounds checking or requiring the programmer to linearize accesses.

[2]It is not surprising that the SUIF pass could do so little for these first three kernels, since tiling can do little to improve locality for their access patterns.

Figure 5: Test kernels

```
/* 5 pointed system */
for(i=1; i<SIZE-1; ++i)
  for(j=1; j<SIZE-1; ++j)
    A[i][j] = (4*A[i][j] + A[i-1][j] + A[i+1][j]
              + A[i][j-1] + A[i][j+1])/8;

/* 9 pointed system */
for(i=1; i<SIZE-1; ++i)
  for(j=1; j<SIZE-1; ++j)
    A[i][j] = (8*A[i][j] + A[i-1][j] + A[i+1][j]
              + A[i][j-1] + A[i][j+1] + A[i-1][j+1]
              + A[i+1][j+1] + A[i-1][j-1]
              + A[i+1][j-1])/16;

/* Matrix-Vector Multiply */
for(i=0; i<SIZE; ++i)
  for(j=0; j<SIZE; ++j)
    Y[i] += A[i][j] * X[j];

/* Matrix-Matrix Multiply */
for(i=0; i<SIZE; ++i)
  for(j=0; j<SIZE; ++j)
    for(k=0; k<SIZE; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

Table 2: Optimizer Performance

Kernel	ML	OPT	OPT+SUIF
5-pointed	0.381	0.1	0.1
9-pointed	0.411	0.11	0.11
MVM	0.18	0.111	0.11
MMM	55.22	6.229	3.084

Time to execute kernel in seconds, SIZE = 400

Table 3: Optimizer vs. gcc

Kernel	OPT+SUIF	gcc	norm. OPT+SUIF	norm. gcc
5-pointed	0.1	.06	.03	.03
9-pointed	0.11	.05	.02	.02
MVM	0.11	.051	.03	.021
MMM	3.084	5.608	3.004	5.578

Time to execute kernel in seconds, SIZE = 400

4.3 Comparison to gcc

Finally, we compared the performance of our optimizer to that of a widely used C compiler, gcc. Using equivalent kernels written in C, we obtained the results in Table 4.3.

The gcc code was compiled with optimizations enabled using the -O2 flag. Our optimizer generated significantly better code for MMM, but worse code for the remaining kernels. We believe this is due to startup costs for the OCaml virtual machine. The second set of columns in Table 4.3 represents normalized times of execution. They are normalized by subtracting the amount of time it takes the equivalent program to run, but with all the loop nests commented out. This data shows that our optimizer produces code as good as the gcc compiler for the 5-pointed and 9-pointed kernels, and code that is only a third slower than the gcc compiler on MVM. Thus our optimizer produces code that is quite competitive with gcc.

4.4 Improving performance

Further performance gains could be achieved by integrating our optimizer with the OCaml native compiler. This would eliminate the virtual machine startup costs and improve the performance of all the OCaml source code that is executed. Another gain would be performing additional optimizations upon the generated C code. This would, in particular, help our optimizer compare more favorably with gcc, since it is performing many optimizations that our optimizer is not.

5 Conclusion

The work presented in this paper demonstrates a prototype implementation of an optimizer for improving array reference locality in OCaml, a modern programming language that does not usually allow such optimizations. Creating the implementation was made especially difficult due to the necessity of interfacing with C and C++ code, namely SUIF and Omega. One conclusion that can be drawn from this is that modern compilers need to provide interfaces to non-proprietary libraries for loop transforms and ILP relations.

The performance of code generated by the prototype exceeds that of even the best-case native code generated by the OCaml compiler. Yet while that code requires that bounds-checking be disabled, the code generated by our prototype guarantees bounds safety through the use of guard generation. A full implementation of the ideas found in this prototype would enable the OCaml compiler to generate code that is competitive with commercial compilers. We conclude that OCaml could be a serious contender against languages such as C and Fortran for use in numerically intensive computation.

References

[1] W. Kelly and W. Pugh. The Omega calculator and library. Technical report, University of Maryland, 1996.

[2] José E. Moreira, Samuel P. Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, 2000.

[3] R. P. Wilson, R. S. French, C. S. Wilson, et al. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.