Bounds Checking: An Instance of Hybrid Analysis

Troels Henriksen, Cosmin E. Oancea
HIPERFIT, Department of Computer Science, University of Copenhagen (DIKU)
athas@sigkill.dk, cosmin.oancea@diku.dk

Abstract

This paper presents an analysis for bounds checking of array subscripts that lifts checking assertions to program level under the form of an arbitrarily-complex predicate (inspector), whose runtime evaluation guards the execution of the code of interest. Separating the predicate from the computation makes it more amenable to optimization, and allows it to be split into a cascade of sufficient conditions of increasing complexity that optimizes the common-inspection path. While synthesizing the bounds checking invariant resembles type checking techniques, we rely on compiler simplification and runtime evaluation rather than employing complex inference and annotation systems that might discourage the non-specialist user. We integrate the analysis in the compiler's repertoire of Futhark: a purely-functional core language supporting map-reduce nested parallelism on regular arrays, and show how the high-level language invariants enable a relatively straightforward analysis. Finally, we report a qualitative evaluation of our technique on three real-world applications from the financial domain that indicates that the runtime overhead of predicates is negligible.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming; D.3.4 [*Processors*]: Compiler

General Terms Performance, Design, Algorithms

Keywords subscripts bounds checking, autoparallelization, functional language

1. Introduction

While massively parallel hardware is mainstream, e.g., GPU, the problem of enabling real-world applications to be readily written by the non-expert user at satisfying level of abstraction, and in a portable manner that offers efficiency comparable to hand-tuned code across various parallel hardware, is still far from being solved.

Futhark is a core array language and compiler infrastructure aimed at effectively expressing, extracting and optimizing (massive) parallelism. Its design captures a meaningful middle ground between the functional and imperative paradigms: *One the one hand*, Futhark is purely-functional and supports regular arrays, (tuples,) and nested parallelism by means of a set of second-order array combinators (SOAC), that have an inherently parallel semantics, e.g., map-reduce. The higher-order invariants allow powerful restructuring transformations that are unlikely to be matched in an imperative context. For example the SOACs' compositional algebra allows producer-consumer fusion to be applied at program level, and to succeed, without duplicating computation, even when a producer is used in multiple consumers [11].

On the other hand, several real-world applications[1] we have examined [14] exhibit destructive updates to array elements inside dependent loops, i.e., cross-iteration dependencies, which would be inefficient, or difficult to analyze if expressed with an array deep-copy semantics inside tail-recursive calls. In this regard Futhark provides (`do`) loops and in-place updates that retain the pure-functional semantics: A loop is a special form of a tail recursive call, just as a `map` is a special form of a `do` loop. An in-place update "consumes" the input array and creates a new array, but under the static guarantee, verified via uniqueness types (inspired by Clean[3]), that any array may be consumed at most once. It follows that the update takes time proportional to the size of the array element. Finally, loops and in-place updates provide the mean to transition, within the same core language, to a lower-level IR that:

- uses seamlessly the same compiler repertoire, and furthermore
- can benefit from imperative optimizations that enhance locality of reference and parallelism, e.g., loop interchange, tiling, etc.

This paper presents a bounds-checking analysis for array subscripts in Futhark. The technique is to separate/lift the checking assertions from the code of interest, and to model them as an *exact* predicate of arbitrary complexity that guards at runtime the execution of the code of interest. The code of interest is typically the outermost `map`, loop, etc. The exact predicate is split into a cascade of sufficient conditions of increasing time complexities, that are evaluated at runtime until one succeeds (if none statically simplify to `true`). We see our technique as a pragmatic compromise between language and compiler solutions to bounds checking:

Type-checking solutions involve no runtime overhead but either significantly restrict the language or require complex inference systems, e.g., dependent types, and thorough annotations of program invariants that might not be accessible to the non-specialist user.

Compiler solutions gather program invariants in an abstract-set representation and attempt to remove each assertion individually, e.g., via Presburger or interval arithmetic, as in range analysis [2, 5]. However, often enough, static disambiguation fails for reasons as simple as the use of symbolic constants of unknown range.

In comparison, our technique lifts the bounds-checking invariant at program level but neither restricts the language nor places any burden on the user. While the analysis (only) guarantees to preserve the work and depth [**?**] asymptotic of the original program, in practice the runtime overhead of predicates is negligible in most cases. The intuition is that subscript computation is typically well separable from, and a very cheap slice of, the original computation.

[1] Our kernels from financial and image processing domain measure in the range of hundred/thousand lines of compact code.

Array'14,, June 13th, 2014, Edinburgh, UK..

ACM 978-1-nnnn-nnnn-6/13/14. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

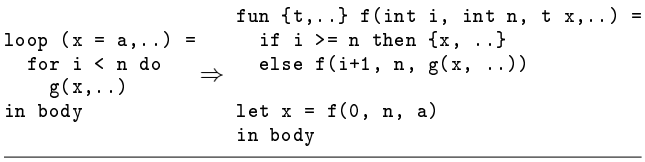

Figure 1. Loop to recursive function

Separating the predicates provides clean executer code and more opportunities for optimizing the inspector (predicate): For example, indirect accesses may create a predicate whose representation is a loop that is invariant, and thus hoistable, across an outer loop. This would require non trivial analysis, e.g., distribution of a dependent loop, if the predicate was not separated from code. Organizing the predicate as a cascade of sufficient conditions allows the analysis to bypass hindrances related to control flow and ranges imprecision and to effectively optimize the common-path predicate.

Finally, our technique is an instance of *hybrid analysis*, which denotes a class of transformations that extract *statically* data-sensitive invariants from the program and aggressively specializes the program based on the result of the *runtime* evaluation of those invariants. Such analyses, reviewed in Section 4, include optimization of the common-execution path in JIT compilation [1], inspector-executor and dependence analysis of array subscripts in automatic parallelization [9, 15–18]. A significant problem however is that, in the imperative context, supporting anything but the simplest, $O(1)$ predicate quickly requires "heroic" efforts.

In this context we report a non-heroic infrastructure that allows arbitrarily shaped/complex predicates to become first-class citizens of the Futhark compiler: predicates are extracted to program level, are aggressively simplified statically, retain the parallelism of the original code, and compose well with the rest of the code. The relatively-simple analysis is enabled by high-level invariants of our functional language. A qualitative evaluation on three real benchmarks supports the claim that in practice the runtime overhead is negligible. The rest of the paper is organized as follows: Section 2 briefly introduces Futhark in an informal manner, Section 3 demonstrate the main steps of the analysis, and Sections 4 and 5 review the related work and conclude.

2. Informal Introduction To Futhark

Futhark[2] is a mostly-monomorphic, statically typed, strictly evaluated, purely functional language, whose syntax resembles the one of SML. Futhark's type system provides (i) basic types `int`, `real`, `bool`, `char`, (ii) n-ary tuple types, which use curly bracket notation, e.g., {α, β, γ}, and (iii) multi-dimensional array types, which use the bracket notation, e.g., [α]. Arrays of tuples are transformed to tuples of array as in NESL [4], and the resulting arrays are checked to be regular. For example array `[{[4],[1.0]}, {[5,3],[2.0]}]`, which belongs to type `[{[int],[real]}]`, is illegal because the first array of its tuple-of-array representation `{[[4],[5, 3]], [[1.0],[2.0]]}` $\in$ `{[[int]],[[real]]}` is irregular, i.e., the size of the first and second row differs $1 \neq 2$.

Most array operations in Futhark are achieved through built-in second-order array constructors and combinators (SOACs), e.g.,

- `replicate(n,e)` creates an array of outermost size `n` whose elements are all `e` (`e` can be an array), and has type {`int`,α}$\rightarrow$[α].
- `iota(n)` constructs the array containing the first `n` natural numbers, i.e., `[0,1,..,n-1]` and has type `int` $\rightarrow$ `[int]`,
- `map(f,[`a_1`,..,`a_n`])` results in `[f(`a_1`),..,f(`a_n`)]`, and has type {$\alpha \rightarrow \beta$, [α]} $\rightarrow$ [β]. The result must be a regular array, whose outermost size matches the one of the input array,
- `reduce(`$\oplus$`,e,[`a_1`,..,`a_n`])` requires $\oplus$ to be a binary *associative* operator (unchecked), results in `e`$\oplus$`a`$_1\oplus..\oplus$`a`$_n$, and has type {$\alpha\rightarrow\alpha\rightarrow\alpha$,$\alpha$,[$\alpha$]} $\rightarrow$ α. The semantics also requires that $\oplus$'s arguments and results have the same shape (if arrays),
- `zip`, `unzip`, `filter`, `scan`, etc., with the usual semantics.

Anonymous and curried functions are permitted only inside SOAC invocations (since Futhark is first-order), and parentheses may be omitted when currying if no curried arguments are given.

Finally, Futhark supports three types of bindings:

- the usual `let id =` e_1 `in` e_2 binds a fresh variable named `id` to the value of expression e_1 in the expression e_2. Pattern matching for tuples is supported as `let {`id_1`,..,`id_k`}=...`
- the `do`-loop syntax is shown in the left side of Figure 1: `n` is the loop count, `i` takes values in `[1..n]`, `x` is a loop-variant variable, and the body of the loop is the call to `g(x,..)`. The loop has the semantics of the tail-recursive function call `f(0,n,a)`, shown on the right side of Figure 1.
- `let b = a with [`i_1`,..,`i_k`] <- v in body` denotes an in-place update: It makes available in `body` a new array variable `b`, which is semantically a deep copy of array `a` but with the element at index `[`i_1`,..,`i_k`]` updated to `v`. In addition, it statically checks that no variable that aliases `a` is used on any execution path following the creation of `b`. Thus, the common shortcut syntax is: `let a[`i_1`,..,`i_k`] = v in body`.

The (last) `let-with` binding guarantees that an in-place update is possible, i.e., at runtime cost proportional to the size of the element `a[`i_1`,..,`i_k`]`, by "consuming" `a`. This behavior is extended across call sites via uniqueness types: In a function `fun *[`α`] f(*[`β`] a, `γ` b) = body` the star before the type of parameter `a` declares that `a` is subject to an in-place update, a.k.a. consumed, inside `body`. The type checking will verify, at any callsite of `f`, that any variable that aliases the corresponding actual parameter of `f` is dead after the call. The star before the result type indicates that the type checking of `f` must verify that the result of `f` does not alias any of the non-unique arguments of `f`, in our case `b`. This invariant is used in type checking the call sites of `f`. In short, the uniqueness-type mechanism supports in-place updates at program level by the means of simpler, intra-procedural analysis.

The compilation pipeline is outlined in Figure 2: Type checking is performed on the original program to ensure that error messages use programmer's names, but during normalisation, all bindings are renamed to be unique - shadowing is not permitted in the internal representation. In general, the internal representation is A-normal form [19], which can intuitively be considered similar to three-address-code statements, e.g., the only direct operands to functions, SOACs and operators are variables or constants. Additionally, we flatten all tuples, and rewrite all arrays-of-tuples to tuples-of-arrays, implying that the tuple type constructor cannot appear inside an array type [4]. Finally, all tuple-patterns are expanded to explicitly name every element of the tuple, with the implication that no variable is ever itself bound to a tuple value. These properties make data-dependencies more precise and explicit, thus enabling easier writing of transformation rules on the syntax tree.

The simplification stage consists of classical optimisations such as inlining, copy propagation, constant folding, common-subexpression elimination and dead-code removal, but also more complex transformations, such as hoisting invariant variables out of loops, and simplifying loops with loop-invariant bodies to closed forms. These simplification rules are critical for optimising the

[2] Futhark (more accurately "Fuþark") are the first six letters of the runic alphabet. In our language they correspond to the six SOACs: `map`, `reduce`, `scan`, `filter`, `redomap`, `multireduce`. (The latter is not supported yet.)

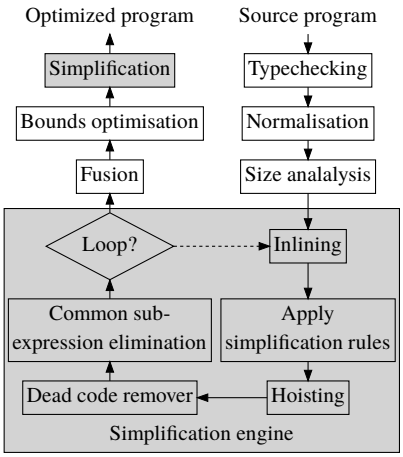

Figure 2. Compiler pipeline

```
fun [[real]] main(int N,int M,[int,M] X,*[[real,M],M] A)=
1.  map (fn *[real] ({*[real],int} e) =>
2.          let {a, i} = e in
3.          loop(a) = for j < N do
4.              let a[j] = a[ X[j] ] * 2.0 in a
5.          in
6.          map (fn real (int j) =>
7.                  if (j <= i-2*N) && (X[j] = j)
8.                  then a[i*j] else 0.0
9.              ,iota(N))
10.     ,zip(A,iota(M)) )
```

Figure 3. Running example: a contrived Futhark program.

predicates described in Section 3.1. Producer-consumer fusion is then applied aggressively, but without duplicating computation. This algorithm is presented in detail elsewhere [10, 11].

3. Bounds Checking Analysis

Figure 3 presents the running example used in this section. The `main` function, which is the program entry point, receives as input two integers `N` and `M`, and two arrays: `X` is a vector of integers of size `M`, and `A` is an `M`×`M` matrix of real numbers. (The arguments of `main` are currently read from standard input.) The program zips each array row with its corresponding index, i.e., `zip(iota(M),A)` ≡ `[..,{i,A[i]},..]`, `i`∈`[0..M-1]`, and maps the resulting array via the anynymous function defined by `fn`. This function:

- uses pattern matching to un-tuple the input, i.e., `let {a,i}=..` binds vector `a` to the i^{th} row of `A`, and `i`∈`[0..M-1]`, then
- executes a (dependent) `loop` which updates in-place the first `N` elements of `a` to values that depend on an element of `a` obtained via the indirect array `X`. Since `a` is "consumed" inside the loop, `a`'s corresponding formal-parameter type must be marked unique, i.e., `*`, in both the anonymous and `main` functions.
- Finally, the result of the outer anonymous function is a vector of size `N` obtained by mapping each element `j` ∈ `[1..N]` via the inner anonymous function, which returns `a[i*j]` if a condition is met or `0.0` otherwise. Thus the result of the program is a matrix with `M` rows and `N` columns.

Examining the running example in Figure 3, one may deduce that the `loop` bounds checking requires `N`≤`M`, due to access `a[j]`, and also `X[j]`≥`0` ∧ `X[j]`<`M`, ∀`j`∈`[0..N-1]`, due to the indirect-

```
fun [[real]] main(int N,int M,[int,M] X,*[[real,M],M] A)=
1.  let p1 = loop (b=True) for j < N do //≡ map-reduce
2.               let c = if j >= M then False
3.                       else (X[j] >= 0) && (x[j] < M)
4.               in b && c                                  in
5.  let p2 = p1 && (N < M + 1)                              in
6.  let p3 = p2 && (M<2*N+2 || M*N<2*M+N-1 || ... )         in
7.  let p = if p3 then True
8.          else ... exact predicate ...          in
9.  let c = assert(p)                                       in
10.  <c>map(fn *[real] ({*[real],int} e) =>
11.              ... original code ...
12.        , zip(A,iota(M)) )
```

Figure 4. Running example: Expected Predicate.

array access `a[X[j]]`. Note that both validity conditions are invariant to the outer `map`, and furthermore the latter condition is fully parallel and can be expressed as a map-reduce computation. The inner map exhibits the access `a[i*j]` and a human would assume that its validity, i.e., `i*j`<`M` is unlikely to depend on the condition `X[j]=j`, and as such, eliminating the variables `i` and `j` of known bounds from the inequality would produce an $O(1)$ sufficient condition involving only symbolic constants `N` and `M`, which it is likely to succeed in most cases. Figure 4 shows the predicate that is expected to succeed: the inspector loop at lines $1-4$ corresponds to the executor `loop` at lines $3-4$ (in Figure 3), which is invariant to and has been hoisted-out from the outermost `map`. Lines 5 and 6 in Figure 4 are the $O(1)$ sufficient conditions for the array indexes `X[j]` and `a[i*j]` at lines 7 and 8 in Figure 3, respectively.

Since the compiler must be conservative, lines 7 and 8 (in Figure 4) cascade the sufficient-condition and the accurate predicates, but the latter is unlikely to be executed on any reasonable data sets.

Finally, the `if j>=M` branch at line 2 is necessary to ensure that the access `X[j]` is within bounds, i.e., assertions are implemented via `if`s, and `<c>map...` on line 10 denotes the assertion under which the execution of the original program is safe. Note that (i) without separating the validity test from the original code the `loop` would still contain the `a[i]` update, which would prevent the loop to be hoisted outside the outer `map`, and (ii) that the `loop` inspector can be easily recognized as a map-reduce computation, albeit it corresponds to a dependent executor loop.

The remainder of this section presents the three main stages of analysis: Section 3.1 describes the transformation that synthesizes at program level an exact predicate, which models the subscript-within-bounds invariant. Section 3.2 describes a top-down (compiler) pass that (i) gathers (symbolic) range information for program variables, and (ii) uses those to compute sufficient conditions for each scalar relation that contributes to bounds checking. For example, inside the inner `map` of Figure 3, `j`∈`[0,N-1]` and `X[j]` requires relation `j < M` to hold, which generates the sufficient condition `N < M+1`. Finally, Section 3.3 presents how sufficient conditions for the exact predicate are extracted at program level.

3.1 Synthesizing The Exact Predicate

Several of the rules that implement the transformation are shown in Figure 5. To simplify notation, we assume that we generate code for the user language, but the program input is in the A-normalized compiler representation, i.e., blocks of bindings except for the last expression of a `then/else` or `loop` or (anonmous) function, etc., which is a (tuple of) variable(s). We also assume that `freshVar()` generates a fresh variable name and `getPredicate(f)` returns the (same) name of the function that is the translation of function `f`.

The application of $\mathcal{E}^{fun}$ to a function `f` constructs the *predicate translation* of `f`, whose (i) name is `f`^{p}=`getPredicate(f)`, (ii) arguments are the same as the original, but (iii) the result type is

$\mathcal{E}^{fun}$(fun β f (α a) = body) $\equiv$
fun {bool,α} f^p(α a) = let p=True in $\mathcal{E}_p^{exp}$(body)
where p=freshVar(), and f^p=getPredicate(f);

$\mathcal{E}_p^{exp}$(let a[i_1,..,i_k] = v in body) $\equiv$
let q = .. && i_k >= 0 && i_k < size(k-1,a) in
let p = p && q
let <q>a[i_1,..,i_k] = v in $\mathcal{E}_p^{exp}$(body)
where q=freshVar(), a is an array var,
size(j,a)=size of dim j of a and v is a var/ct;

$\mathcal{E}_p^{exp}$(let b = f(a) in body) $\equiv$
let {q,b}=f^p(a) in let p=p&&q in $\mathcal{E}_p^{exp}$(body)
where q=freshVar() and f^p=getPredicate(f);

$\mathcal{E}_p^{exp}$(let b = map(f, a) in body) $\equiv$
let {q,b}=unzip(map(f^p, a) in body
let p = p&& reduce(op &&,True,q) in $\mathcal{E}_p^{exp}$(body)
where q=freshVar() and f^p=getPredicate(f);
a is an array variable.

$\mathcal{E}_p^{exp}$(a) $\equiv$ {p,a}

Figure 5. Transformation rules for the exact predicate.

```
fun [[real]] main(int N,int M,[int,M] X,*[[real,M],M] A)=
0.  let A' = copy(A) in
1.  let {ps, rs} = unzip (
2.  map (fn {bool,*[real]} ({*[real],int} e) =>
3.        let p      = True in
4.        let {a, i} = e    in
5.        loop(p,a) = for j < N do
6.            let p1 = (j >= 0) && (j < M)              in
7.            let p  = p && p1                          in
8.            let p1c = assert(p1)                      in
9.            let p2 = (<p1c>X[j]>=0) && (<p1c>X[j]<M) in
10.           let p  = p && p2                          in
11.           let <p1>a[j] = <p2>a[<p1>X[j]]*2.0        in
12.           {p,a}
13.       in let {ps, rs} = unzip (
14.       map(fn {bool,real} (int j) =>
15.           let p  = True                         in
16.           let p3 = (j >= 0) && (j < M)          in
16            let p3c  = assert(p3)                 in
18.           let p  = p && p3                      in
19.           if (j<=i-2*N) && (<p3c>X[j]=j)        in
20.           then let p5 = (j*i >= 0) && (j*i<M) in
21.           then let p5c = assert(p5)             in
22.                {p && p5, <p5c>a[j*i]}
23.           else {p, 0.0}
24.       ,iota(N)) )
25.       in { p && reduce(op &&, True, ps), rs }
26.     , zip(A',iota(M)) ) ) in
27. let p = reduce(op &&, True, ps) in ...
```

Figure 6. Exact predicate before simplification.

a single bool value, denoting the predicate, and the original result type. The body of the function starts by creating a fresh bool variable named p = freshVar() that is initialized to True and used in the expression generated by $\mathcal{E}_p^{exp}(body)$.

$\mathcal{E}_p^{exp}(e)$ translates an expression to the predicated form, and places the result in the last instance of the variable named p, whose name is reused, e.g., let p = p && q in .... To translate an in-place update we create a fresh boolean variable q that verifies that the subscript is within range, we add the contribution of q to p, i.e., p = p && q, protect the indexed array with assertion q, and translate the body. Note that the representation guarantees that $i_1..i_k$ and v are variables/constants, hence they are left unchanged.

```
fun [[real]] main(int N,int M,[int,M] X,*[[real,M],M] A)=
1.  let p = if (M >= 0) then
2.      loop(p=True) = for j < N do
3.          let p1 = (j >= 0) && (j < M)              in
4.          let p  = p && p1                          in
5.          let p1c  = assert(p1)                     in
6.          let p2 = (<p1c>X[j]>=0) && (<p1c>X[j]<M)  in
7.          p && p2                                   in
8.      else True                                     in
9.  let ps1 =
10.      map (fn bool (int i) =>
11.             let ps2 =
12.                map(fn bool (int j) =>
13.                    let p  = True                  in
14.                    let p3 = (j >= 0)              in
15.                    let p4 = (j <  M)              in
16.                    let p5 = p3 && p4              in
17.                    let p5c = assert(p5)           in
18.                    let p  = p && p5               in
19.                    if (j<=i-2*N) && (<p5c>X[j]=j) in
20.                    then let p6 = (j*i >= 0)       in
21.                         let p7 = (j*i <  M)       in
22.                        p && p6 && p7
23.                    else p
24.                ,iota(N) )
25.             in p && reduce(op &&, True, ps2)
26.     ,iota(M) )
27. let p = p && reduce(op &&, True, ps1) in ...
```

Figure 7. Exact predicate after simplification.

The translation of a function-call binding is obtained by performing a call to the predicate translation of the function, adding the predicate contribution of the function using &&, and translating the remaining expression. Finally, a binding of map(f,a), where a is an array variable: (i) calls map(f^p, a), where f^p is the predicated translation of f, (ii) conjugates the predicate-contribution array via reduce(op &&, True), because each array element needs to be safely processed, (iii) adds the result of the reduction to p and translates the remaining expression.

In rule $\mathcal{E}_p^{exp}$(a), a is a variable/constant, which is the result of a block of let bindings. As such the rule simply tuples the predicate p with the original result a. The rest of the rules are similar.

The translation starts with the body of main, which requires first that all unique parameters of main are copied and their names substituted in the translation. Otherwise the uniqueness invariant might be violated because the same unique array might be consumed both in the predicated translation and in the original code. This however does not affect the asymptotic complexity since the arguments of main are typically read from a file anyway. A user-language version of the exact-test predicate is shown in Figure 6. Note that (i) both the inner and outer maps result in an extra boolean array which is reduced with the and operator (&&), (ii) the loop has an extra boolean variant parameter, and (iii) all indexing operations have been asserted via their corresponding predicate.

After aggressive dead-code removal and simplification, the resulting code is similar to the one in Figure 7. Note that (i) all references to the unique (copied) array A' have been removed, and that (ii) the loop has been hoisted outside the outer map, and cannot be asymptotically simplified further due to the indirect access, e.g., X[j] < M, that cannot be proven in a cheaper way. The next sections focus on optimizing the common inspector path of let ps1 = map (fn bool (int i) => ... In particular, factors such as p6 and p7, depend only on symbols i and j, of known ranges [0,M-1] and [0,N-1], whose gaussian-like elimination will produce an $O(1)$ sufficient condition for the outermost map.

3.2 Sufficient Conditions for Scalar Relational Expressions

The second phase of the analysis performs a compiler pass that

- extracts (simple) ranges for program variables, e.g., induction variables of `loops`, `maps`, branch conditions, etc,. and
- uses those ranges to eliminate the symbols for which both ranges exists from the predicate factors of interest, e.g., `p5`, `p6`.

The top-down analysis maintains three symbol tables:

- $\mathcal{E}^V$: `Id` $\rightarrow$ (`Int`, `ScalExp`) maintains for each integral variable (i) the lexicographic order in which the variable appears in the program and (ii) its aggressively expanded scalar expression. The representation uses a simple scalar language, `ScalExp`, that supports value, variable, unary negation, binary plus, minus, multiply, division, and nary Min/Max constructors.
- $\mathcal{R}^V$: `Id` $\rightarrow$ (`Int`, `ScalExp` $\cup\ Udef$, `ScalExp` $\cup\ Udef$) maps each integral variable to a triplet formed by (i) an integer denoting the lexicographic order in which the variable appears in the program, and (ii) the lower/upper known ranges of that variable, where $Udef$ is the undefined range, i.e., $[-\infty, \infty]$.
- $\mathcal{S}^R$: `Id` $\rightarrow$ ([[`ScalExp`]] $\cup\ Udef$), the result of this analysis stage maps a boolean variable to an expression in disjunctive normal form (DNF) in which each integral scalar expression e corresponds to the bool expression $e < 0$. For example `[[M*N<2*M, M>0], [3*N*M>M*M, N>0]]` corresponds to `(M*N<2*M` $\wedge$ `M>0)` $\vee$ `(3*N*M>M*M` $\wedge$ `N>0)`.

While more refined solutions are possible [2], we currently gather simple range invariants, for example from `loops` and `iota` parameters passed to `map`, and filter the ranges based on branch conditions. In the example in Figure 7, `iota(M)` is passed as the array parameter to the outer `map` which semantically means that just inside the outer anonymous function $\mathcal{R}^V$ =`{M` $\rightarrow$ `{1,1,`$Udef$`}, i`$\rightarrow$`{10,0,M-1}}`, i.e., `M > 0`, otherwise `iota(M)` is empty, and `i`$\in$`[0,M-1]` since `i` is an element of `iota(M)=[0,..,M-1]`. The priority of `i`, i.e., the first element of the tuple, is greater than that of `M` signaling that `M` appears before `i` in the program and hence its value cannot depend on `i`.

Just before the `if` branch in the inner anonymous function $\mathcal{R}^V$ =`{N`$\rightarrow$`{0,1,`$Udef$`}, M`$\rightarrow$`{1,1,`$Udef$`}, i`$\rightarrow$`{10,0,M-1}, j`$\rightarrow$`{15,0,N-1}}`, where `j` and `N` have been similarly added. On the `then` branch, we use the factor `j<=i-2*N` of the branch condition to refine the range of `j`, i.e., the variable with the highest priority among those that appear in the expressions. The latter guarantees that variable elimination process will eventually terminate, since the lexicographical order respects the program dependency graph (DAG). The other condition, `X[j]=j` is not useful because the range refinement of `j` would still depend on `j`. The range of `j` inside the `then` branch becomes: `j`$\in$`[0, Min(i-2*N,N-1)]`.

When analysis reaches one of the factors of the predicate, e.g., `p5`, it uses the bindings in $\mathcal{E}^V$ to construct an expanded relational (scalar) expression of the form $e < 0$. In the case of `p6`, the relation is `-i*j - 1 < 0`. At this point it computes the set of variables in e, e.g., `i, j`, and lookups their ranges in $\mathcal{R}^V$ to find candidates for elimination. If at least one of the bounds is defined, the variable is kept, otherwise it is removed from the set. The candidates are ordered by inverse order of priority, such that the "most dependent" variable is the first candidate for elimination. Our integral simplifier attempts to bring the expression to a "normal" form in which the `Min/Max` operators are at the outermost level and their inner expressions are in the sum-of-product form. This is however not possible if a `Min/Max` has a factor of unknown sign.

Figure 8 shows the main "inference" rules for eliminating variables. The first one assumes a sum-of-product form and attempts to find a `Min/Max` factor that uses `i`, the symbol to be eliminated. If this is found, it decomposes the problem based on the sign of the `Min/Max` factor `a`. If no such `Min/Max` exists than the second rule is attempted: `e` is brought to the form `e = a*i + b` by dividing each term of `e` by `i`; if the division succeeds then the divided term contributes to `a`, otherwise, the undivided term contributes to `b`. If the separation succeeds, i.e., `i` does not appear in `b`, then the problem is decomposed again. Otherwise, `i` is removed from the list of candidates, and the process continues until no candidates remain.

For the predicate `p6`, the first to-be-eliminated variable is `j`$\in$`[0,Min(i-2*N,N-1)]`. The second rule of Figure 8 results in:

$\mathcal{SC}$`(j)(-i*j-1<0)` $\equiv$ ($\mathcal{SC}$`(j)(-i`$\leq$`0)` $\wedge$ $\mathcal{SC}$`(j)(-1<0))` $\vee$
($\mathcal{SC}$`(j)(-i`$\geq$`0)` $\wedge$ $\mathcal{SC}$`(j)(-i*Min(i-2*N,N-1)-1<0))`.

The first term simplifies to true via the second rule:

$\mathcal{SC}$`(j)(-i`$\leq$`0)` $\equiv$ $\mathcal{SC}$`(j)(-i-1<0)` $\equiv$ $\mathcal{SC}$`(j)(0-1<0)`

Finally, the predicate obtained after simplifying `e < 0` is brought to DNF form. After this pass, $\mathcal{S}^R$ $\equiv$`{p3`$\rightarrow$`[[True]], p4`$\rightarrow$`[[N-M-1<0]], p6`$\rightarrow$`[[True]], p7`$\rightarrow$`[[M-2<0], [M-2*N -2<0], [N-2<0], [M*N-2*M-N+1<0]]}`.

$\mathcal{SC}$`(i)(terms + a*Max(`b_1`, ..i.., `b_n`)) < 0)` $\equiv$
($\mathcal{SC}$`(i)(a` $\leq$ `0)` $\wedge$ ($\wedge_{k=1}^{n}$ $\mathcal{SC}$`(i)(a*`b_k`+terms < 0)))` $\vee$
($\mathcal{SC}$`(i)(a` $\geq$ `0)` $\wedge$ ($\vee_{k=1}^{n}$ $\mathcal{SC}$`(i)(a*`b_1`+terms < 0)))`
if variable `i` appears inside the `Max` expression;

$\mathcal{SC}$`(i)(a * i + b < 0)` $\equiv$
($\mathcal{SC}$`(i)(a` $\leq$ `0)` $\wedge$ $\mathcal{SC}$`(i)(a *` l `+ b < 0))` $\vee$
($\mathcal{SC}$`(i)(a` $\geq$ `0)` $\wedge$ $\mathcal{SC}$`(i)(a *` u `+ b < 0))`

Figure 8. Extracting sufficient conditions for scalar relations.

3.3 Cascading Sufficient-Condition Predicates

The last stage of the analysis extracts sufficient conditions for the exact test at program level. Our implementation currently covers only $O(1)$ and $O(N)$ predicates, where typically, `N` is the count of the outermost recursion, because cases that requires more effort are rare in practice. The analysis consists of a top-down pass that

- computes, for each variable, the recurrences, e.g., SOAC, `loops`, etc., in which that variable is variant, and
- uses this information to select a set of recurrences L_v=$\{l_1..l_q\}$, of maximal nest depth `k` (in our case `k=1`), such that any factor of the exact predicate, e.g., `p7` in Figure 7, has at least one term of its DNF form that is variant only to recurrences in L_v.

The idea is that, if all the factors of the accurate predicate have such terms, which are only variant to recurrences in L_v, then a sufficient condition of nest depth `k` for the accurate predicate can be built by taking the conjunction of the sufficient-condition terms.

We start with $L_v = \emptyset$ and, at each encountered factor of the big predicate, we add variant loops to L_v as long as there are no more than `k` loops that share a common execution path, i.e., are nested. With L_v available, we perform a second pass that:

- replaces every factor of the exact predicate with the disjunction of its DNF terms variant only in L_v, e.g., `let p4 = N-M-1<0`,
- if the condition of an `if` expression is only L_v variant, then the `if` is preserved, otherwise it is translated to the conjunction of the (sufficient-condition) predicates of the two branches[3],
- updates the assertions that guard array indexing inside the predicate, to refer to the sufficient-conditions of the factor, and furthermore, translates those assertions to `if` branches such that

[3] But only if the lifted code is simple enough to guarantee that an error cannot occur, e.g., division by zero, and fail otherwise.

```
fun int xorInds(int bits_num, int n, [int] dir_vs ) =
  let bits = iota   ( bits_num )                    in
  let inds = filter ( testBit(grayCode(n)), bits )  in
  let dirs = map( fn int (int i)=>dir_vct[i], inds ) in
  reduce( op ^, 0, dirs )
```

Figure 9. Pattern in P0

an error is reported only if none of the predicates evaluates to `true`. In particular, this is necessary on platforms that do not support an assertion system, e.g., OPENCL.

- finally, cascades the predicates obtained for `k=0,1..` and runs the simplification engine to obtain the desired time complexity[4].

The transformed code for out working example is similar to the one in Figure 4: the successful predicate requires negligible $O(\mathtt{N})$ work (in comparison with $O(\mathtt{N} \times \mathtt{M})$ of the original computation).

3.4 Evaluation and Discussion

We use for testing three real-work kernels from financial domain: `P0` is a real-world pricing kernel for financial derivatives [14], `P1` is a calibration of interest rate via swaptions, which computes some of the parameters of `P0`, and `P2` implements stochastic volatility calibration via Crank-Nicolson finite differences solver [13].

`P1` is the largest benchmark, but it is very regular, and the bounds-checking analysis succeeds statically.

`P2` has the structure of an outer convergence loop, of count T, that contains SOAC nests of depth three and of count $\mathtt{K} \times \mathtt{M} \times \mathtt{N}$, e.g., $1024 \times 256 \times 256$. The SOAC nest exhibits (i) several arrays that are assumed to hold at least two elements, i.e., indexed on $[0]$ and $[1]$, and (ii) several stencil-like subscripts, e.g., `let a[i] = if i < N-1 then (a[i] + a[i+1])/2 else a[i]`, where `N` is the array size. (The latter requires the range of `i` to be narrowed to `[0,N-2]` inside the `then` branch.) With our technique, the successful predicate has complexity $O(1)$. In comparison, the classical technique would result in $O(\mathtt{T} \times \mathtt{K} \times \mathtt{N} \times M)$ checks.

The most interesting program is `P0`. It features the pattern discussed in this paper, in which a dependent inner loop features an indirect indexing which is invariant to two outer loops, that have a combined loop count in the range of 10^6. The successful predicate/inspector contains one loop, whose runtime is negligible with respect to the the executor/computation runtime.

Another interesting code from `P0`, corresponding to Sobol number generation, is depicted in Figure 9: The code filters `iota(bits_num)`, where `bits_num` is a symbolic constant, either 32 or 64. It follows that the range of all elements of the result `inds` is `[0,bits_num-1]`. The `inds` array is next used as an indirect array, i.e., `map(fn int (int i)=>dir_vct[i], inds)` returns the values corresponding to the indices `inds` of array `dir_vs`. The check for `dir_vct[i]` succeeds statically, because (i) array `dir_vct` is known statically to have size `bits_num`, and (ii) `i` is an element of `inds`, and as such has range `[0, bits_num-1]`.

In conclusion, for the three examined kernels, our analysis solves most of the bounds checking statically, but several important, deeply nested cases have required predicate separation and sufficient conditions that resulted in negligible runtime overhead. However, there are examples in which the subscript values depend on the computation, e.g., histogram computation. Such cases are ill suited for our analysis, in that it would double the original work. A solution to this problem is to develop a cost model that, for example, would use the traditional method whenever sufficient conditions of "negligible" overhead cannot be extracted. Finally, while on cache based (SMP) systems naive bounds checking might still work well enough, the main motivation for this work has been GPU architectures: In the absence of hardware support for assertions, the computation would need to be protected with a cascade of `if` branches. This would make the resulting code less amenable to subsequent optimizations, and would also significantly impact performance.

[4] The resulted sufficient-condition predicate is guaranteed to be simplified to code exhibiting at most nests of depth `k`, because the predicate result is variant only to recurrences in L_v. The compiler asserts this invariant.

4. Related Work

The main inspiration for this paper has been the work done in the context of automatic parallelization of imperative languages, such as Fortran, where the significant benefit of running the code in parallel has led to aggressive techniques that combine static with dynamic analysis. For example, inspector-executor techniques [18], computes "mostly" at runtime a schedule of dependencies that allows the executer to run in parallel. Since the inspector overhead was significant, analysis has shifted towards static analysis, e.g., Presburger arithmetic is extended with uninterpreted-symbol functions [17], and the resulting irreducible Presburger formulas are presented to the user for verification as simple predicates.

Other work summarizes at program level array accesses, either via systems of affine inequations or as a program in a language of abstract sets, and summaries are paired with predicates that are evaluated at runtime to minimize overheads [9, 15]. However, providing support for arbitrary predicates and summaries, requires in the imperative context many helper intermediate representations and even an entire optimization infrastructure for these new languages, which has been informally characterized as "heroic effort".

Futhark takes the view that a simple, array functional language, but with a richer algebra of invariants might be better suited to support such aggressive analysis, because for example the language already provides the necessary support for predicates, and any improvements in the analysis support would directly benefit user code. After all, summary, predicate and SSA languages are functional.

Futhark is similar to SAC [7] in that it provides a common ground between functional and imperative domains. For example, SAC uses `with` and `for` loops to express `map-reduce` and dependent computation, respectively. Recent work on SAC proposes "hybrid-array types" [8] as a mean to express explicitly certain constraints related to the structural properties or values of argument/result arrays. Similar with our work, hybrid arrays rely on the compiler infrastructure to prove constraints mostly statically, and avoids the pitfall of dependent types by compiling unverified constraints to dynamic checks. The (other) direction of verifying via dependent types array related invariants has been studied in Qube [20], an experimental (simplified) offspring of SAC.

An approach to statically optimising checks can be found in [21], which extends an ML-like language with a restricted variant of *dependent types*. Type-checking then involves statically proving that array accesses are in-bounds, and as a result making runtime bounds-checking unnecessary. Not all bounds checks can be statically eliminated, e.g. indirect indexes, and for these the programmer must leave explicit bounds checks in the program. These would still be amenable to optimisation utilising the techniques outlined in this paper. In contrast to the technique presented in this paper, the dependent types approach demands that the programmer annotate the source program, although this also carries the benefit that size constraints can be expressed in function- and module signatures.

Another technique [12] extracts a a slice of the input program computing the shape of intermediate results, such that static evaluation of this slice will reveal any shape errors (e.g. out-of-bounds accesses) in the original program. This guarantees discovery of all out-of-bounds indexing, but at a cost: the shape of the result of any operation must depend solely on the shape of its input. That is, the shape of a structure in a program must not depend on the data, thus banning operations like `iota` and `replicate`.

Finally, another approach, taken by languages in the APL family, is to extend the language operators in a way that guarantees that out-of-bounds indices cannot occur [6]. This typically introduces non-affine indexing that might hinder subsequent optimizations.

5. Conclusions and Future Work

This paper has presented an instance of hybrid analysis for checking subscript within bounds invariants. The approach enables aggressive hoisting by separating the predicate from the computation, and optimizes the common-execution path of the predicate by extracting and cascading a set of sufficient conditions that are evaluated at runtime, in the order of their complexity until one succeeds. We have shown that such an aggressive analysis is relatively straightforward in Futhark, and that it successfully solves several real-world applications under negligible runtime overhead.

Acknowledgments

We thank the Array'14 reviewers for the excellent feedback. This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (http://hiperfit.dk) under contract number 10-092299.

References

[1] M. R. Arnold, S. Fink, D. P. Grove, M. Hind, , and P. F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of IEEE*, 92(2):449–466, 2005.

[2] H. Bae and R. Eigenmann. Interprocedural Symbolic Range Propagation for Optimizing Compilers. In *Procs. Lang. Comp. Par. Comp. (LCPC)*, pages 413–424, 2005.

[3] E. Barendsen and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. In *Found. of Soft. Tech. and Theoretical Comp. Sci. (FSTTCS)*, volume 761 of *LNCS*, pages 41–51, 1993.

[4] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing*, 21(1):4–14, 1994.

[5] W. Blume and R. Eigenmann. Symbolic Range Propagation. In *Procs. Int. Parallel Processing Symposium*, pages 357–363, 1994.

[6] M. Elsman and M. Dybdal. Compiling a subset of apl into a typed intermediate language. In *Procs. Int. Workshop on Lib. Lang. and Comp. for Array Prog. (ARRAY)*. ACM, 2014.

[7] C. Grelck and S.-B. Scholz. SAC: A Functional Array Language for Efficient Multithreaded Execution. *Int. Journal of Parallel Programming*, 34(4):383–427, 2006.

[8] C. Grelck and F. Tang. Towards Hybrid Array Types in SAC. In *7th Workshop on Prg. Lang., (Soft. Eng. Conf.)*, pages 129–145, 2014.

[9] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural Parallelization Analysis in SUIF. *Trans. on Prog. Lang. and Sys. (TOPLAS)*, 27(4):662–731, 2005.

[10] T. Henriksen. Exploiting functional invariants to optimise parallelism: a dataflow approach. Master's thesis, DIKU, Denmark, 2014.

[11] T. Henriksen and C. E. Oancea. A T2 Graph-Reduction Approach to Fusion. In *Procs. Funct. High-Perf. Comp. (FHPC)*, pages 47–58. ACM, 2013. ISBN 978-1-4503-2381-9.

[12] C. B. Jay and M. Sekanina. Shape checking of array programs. In *Procs. In Computing: the Australasian Theory Seminar*, 1997.

[13] C. Munk. Introduction to the Numerical Solution of Partial Differential Equations in Finance. 2007.

[14] C. Oancea, C. Andreetta, J. Berthold, A. Frisch, and F. Henglein. Financial Software on GPUs: between Haskell and Fortran. In *Funct. High-Perf. Comp. (FHPC'12)*, 2012.

[15] C. E. Oancea and L. Rauchwerger. Logical Inference Techniques for Loop Parallelization. In *Procs. of Int. Conf. Prog. Lang. Design and Impl. (PLDI)*, pages 509–520, 2012.

[16] C. E. Oancea and L. Rauchwerger. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Int. Lang. Comp. Par. Comp. (LCPC'11)*, volume 7146 of *LNCS*, pages 61–75, 2013.

[17] W. Pugh and D. Wonnacott. Constraint-Based Array Dependence Analysis. *Trans. on Prog. Lang. and Sys.*, 20(3):635–678, 1998.

[18] L. Rauchwerger, N. Amato, and D. Padua. A Scalable Method for Run Time Loop Parallelization. *Int. Journal of Par. Prog*, 26:26–6, 1995.

[19] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *SIGPLAN Lisp Pointers*, V(1):288–298, Jan. 1992. .

[20] K. Trojahner and C. Grelck. Descriptor-free representation of arrays with dependent types. In *Procs. Int. Conf. on Implem. and Appl. of Funct. Lang. (IFL)*, pages 100–117, 2011. ISBN 978-3-642-24451-3.

[21] H. Xi and F. Pfenning. Eliminating Array Bound checking Through Dependent Types. *SIGPLAN Not.*, 33(5):249–257, 1998. .