An Approach for Proving the Correctness of Inspector/Executor Transformations

Michael Norrish[1] and Michelle Mills Strout[2]

[1] NICTA, michael.norrish@nicta.com.au
[2] Colorado State University, michelle.strout@colostate.edu

Abstract. To take advantage of multicore parallelism, programmers and compilers rewrite, or transform, programs to expose loop-level parallelism. Showing the correctness, or legality, of such program transformations enables their incorporation into compilers. However, the correctness of inspector/executor strategies, which develop parallel schedules at run-time for computations with nonaffine array accesses, rests on the correctness of the inspector code itself. Since inspector code is often provided in a run-time library, showing the correctness of an inspector/executor transformation in a compiler requires proving the correctness of any hand-written or compiler-generated inspector code as well. In this paper, we present a formally defined language (called PseudoC) for representing loops with indirect array accesses. We describe how using this language, where the reads and writes in array assignments are distinguished, it is be possible to formally prove the correctness of a wavefront parallelism inspector in HOL4. The key idea is to reason about the equivalence of the original code and the inspector/executor code based on operational semantics for the PseudoC grammar and properties of an executable action graph representation of the original and executor loops.

1 Introduction

Inspector/executor strategies refer to a category of program transformations that apply to loop nests with non-affine memory references and loop bounds. The executor is a transformed version of the original code, and the inspector code implements parallelization and/or run-time reordering by performing a data reordering and/or a reordering of loop iterations into a new schedule. The new schedules and/or data orderings are created to parallelize the loop and/or improve the data locality.

In this paper, we focus on reordering iterations of a sequential loop into a sequence of wavefronts, where all of the iterations within a wavefront can be executed in parallel. A wavefront parallelism inspector/executor transformation consists of the compile-time transformation of loops with non-affine array accesses, like the one in Figure 1, into an inspector and an executor as shown in Figure 2. Assuming that the index arrays h, g, and f from Figure 1 are not modified in the loop being transformed, the dependencies between iterations in the original loop can be determined at runtime immediately before the execution of the loop itself.

```
// ==== original executor
for (i=lb; i<ub; i++ ) {
     A[h[i]]=...A[g[i]]
               ...A[f[i+1]]...
}
```

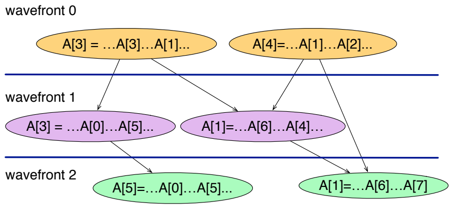

Fig. 1: Example original code with indirect array accesses and an example action graph.

```
// ==== inspector
// Code that generates the data
// structures to represent
// iterations in each wavefront.

// ==== transformed executor
for (w=0; w<numwave; w++) {
  parfor ( i in wavefront(w) ) {
    A[h[i]]=...A[g[i]]
              ...A[f[i+1]]...
  }
}
```

Fig. 2: The inspector and executor code after applying the wavefront parallelism transformation.

Figure 1 also illustrates what we call an *action graph* for the example loop given that `lb = 0`, `ub = 6`, `h[] = {4,3,3,5,1,7}`, `g[] = {1,3,0,0,6,6}`, and `f[] = {-1,2,1,5,5,4,7}`. Each node in the action graph contains all of the read and write access information for a statement instance and enough information about the statement to enable execution of that statement. The read and write information as well as original execution order is used to connect nodes with data dependence edges. The action graph differs from a dependence graph in that a data-flow dependence graph just indicates the partial ordering between iterations and is not executable.

Inspector strategies are quite prevalent in the literature [1–10]. They are implemented manually [9], with code generators for specific kernels [10], and with compilers inserting calls to run-time library inspectors [11]. The issue with all of these approaches is that the inspector implementations are assumed to be correct. The inspector code is non-trivial because because its efficiency often derives from specializing the implementation for a specific kernel and/or a restricted set of kernels with certain characteristics, such as the sparse matrix data structure being used.

In this paper, we present a formal strategy for proving the correctness of inspectors that we are in the process of applying to the wavefront parallelization inspector/executor transformation. The first realization is that we can break out the reads and writes in the representation of the original computation, providing a convenient picture of a program's memory behavior. This leads to our language PseudoC, which is given a formal small-step operational semantics (one that allows for interleaving across concurrent computations). We continue with a general theorem connecting our formalization of the action graph view of a program to its realization in PseudoC. This connection allows proofs of program equivalence in the presence of iteration reordering and/or parallelization. Preconditions on such results then serve as the formal specification of inspectors. Still using the HOL4 theorem prover [12], we then show the correctness of a simple inspector that uses wavefronts. Mechanizing the full wavefront parallelization inspector/executor is ongoing work.

Section 2 describes the applicability and limitations of the example original computation and inspector/executor strategy. Section 3 indicates why the typical approach to showing the legality of auto-parallelization and other program transformations using data dependence abstractions is problematic. when applying this approach to inspector/executor strategies. Section 4 presents the PseudoC grammar that breaks out reads and writes to enable formal reasoning about data dependencies. Section 5 describes how the proof is done in HOL4 using the action graph and small-step operational semantics for PseudoC. Section 6 summarizes related work, and Section 7 concludes.

2 Applicability of Wavefront Parallelization

Wavefront parallelization has been applied to applications such as sparse triangular solvers [13], `doacross` loops in PERFECT benchmarks [14], and more recently in various benchmarks and a proxy application called UMT (Unstructured Mesh for Transport code) [15]. We developed a microbenchmark and wavefront inspector/executor to use as an initial case study for developing an approach to formally proving the correctness of inspector/executor transformations. This section describes the micro-benchmark and presents some experimental results to illustrate the applicability of the wavefront parallelization transformation.

The micro-benchmark, with the original code and executor shown in Figure 3, reads in a square, sparse matrix and uses the pattern of non-zero entries in the sparse matrix to provide a `do across` loop dependence pattern[3]. For each non-zero entry, the benchmark performs a parameterized summation involving exponent calls to enable experimentation with the arithmetic intensity (flops/byte). If the amount of work is set to zero, then no `exp()` calls will occur but at least four adds will occur each iteration. The particular summation in the inner loop was selected because it converges to a small number, which enables easy testing. The result of the summation is added to the data array as indexed by the row and column of the non-zero entries.

The wavefront executor is shown in Figure 3. The initial inspector for our case study is serial, but we also plan to implement the parallel wavefront inspectors by Rauchwerger et al. [14] and Zhuang et al. [15]. The serial inspector code in our case study visits each iteration of the original loop in order and keeps track of the last write and read for each data element in an array. This information is used to determine the earliest wavefront in which to place an iteration while still satisfying data dependencies.

We ran wavebench on an HP-Z800-Xeon E5520-SATA, 8 core 2.26GHz with 12GB of RAM, 32KB L1, 256KB L2, and two 8MB L3 caches, where each L3

[3] If there is a non-zero at location A_{ij} in the sparse matrix and $i < j$, then there is a dependence between iteration i and j (i must execute before j).

```
// =========== Original Code ===========
// foreach non-zero $A_{ij}$ in sparse matrix:
for (int p=0; p<nnz; p++) {

  // sum = $\Sigma_{k=0}^{w-1}$ 1 / exp( k * data[row[p]] * data[col[p]] )
  double sum = 0.0;
  for (int k=0; k<workPerIter; k++) {
    sum += 1.0/exp((double)k*data[row[p]]*data[col[p]]);
  }

  data[row[p]] += 1.0 + sum;
  data[col[p]] += 1.0 + sum;
}

// =========== Executor =======
// for each wavefront
for (int w=0; w<=max_wave; w++) {
  // foreach non-zero $A_{ij}$ in sparse matrix in wavefront
  #pragma omp parallel for shared(data)
  for (int k=wavestart[w]; k<wavestart[w+1]; k++) {
    int p = wavefronts[k];

    // same loop body as in Original Code
  }
}
```

Fig. 3: The wavebench micro-benchmark original code and executor.

cache is shared by 4 cores. The sparse matrices used are from the Davis Florida Collection [16] and range from 3.1 to 8.5 million non-zeroes.

Figure 4 shows the speedup of the executor for the five matrices we ran when the number of `exp()` calls were set to 4. The average parallelism in the action graph for each sparse matrix is shown in parentheses next to the sparse matrix name. Average parallelism is the number of iterations divided by the number of wavefronts, or critical path length. The inspector overhead was less than running the executor once. Note that the 1d5pt.mm does not have any parallelism to exploit, so wavefront parallelization only adds additional loop overhead.

As an exemplar, this micro-benchmark has many benefits. We can determine when wavefront parallelism will be beneficial by modifying the amount of work being performed in the inner loop. When the number of `exp()` calls is 4 or above, all of the input matrices with parallelism have a speedup of 2 or higher. This indicates that for this machine a computation would require a lot of work within the loop for wavefront parallelism to be useful. The other benefit of this micro-benchmark is that the dependencies between iterations in the loop are

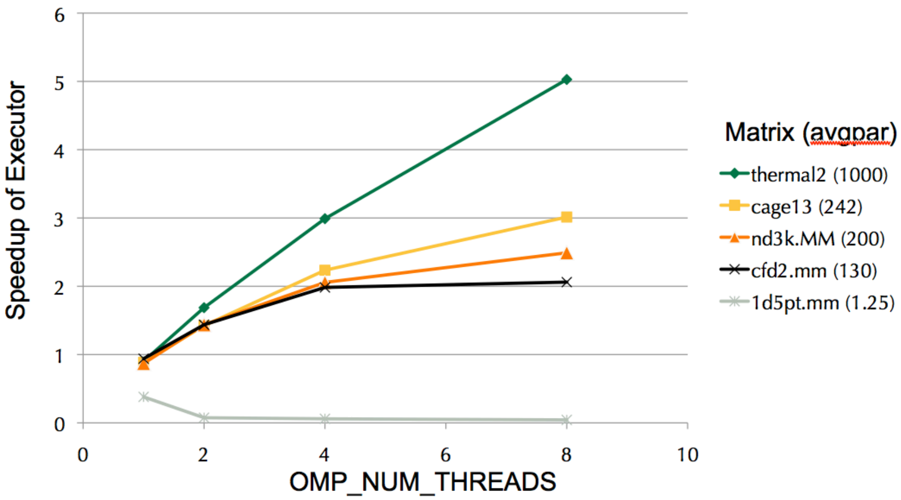

Fig. 4: Speedup of the wavebench micro-benchmark on a Xeon 2x quad core.

not artificially generated. They derive from sparse matrices applicable to real applications.

The limitation of the benchmark is that it is an artificial computation kernel and we are only performing one inspector/executor strategy to it, specifically wavefront parallelism. The simplicity of the computational kernel and the single, non-parallel inspector ease the process of developing an approach to performing formal verification of inspector/executor transformation correctness.

3 The Legality of Program Transformations

In the compilers research area, the legality of auto parallelization and program transformations are determined by performing a data dependence analysis and then checking whether the transformed representation of the program still satisfies the original dependencies [17–20].

This approach for reasoning about transformation legality starts to collapse under the weight of too many assumptions when applied to transformations like wavefront parallelization that must satisfy dependencies in partially parallel loops. In this Section, we review how dependence analysis has been used in the past to determine transformation legality and indicate issues that arise when applying this approach to inspector/executor transformations.

The data dependencies between computations (i.e., iterations in a loop) are derivable from how each iteration reads and writes data. Data dependence analysis represents such data dependencies with various levels of precision [21]. For computations with indirect array accesses, Wonnacott and Pugh [22] introduced the idea of using uninterpreted function symbols to represent dependence relations in such codes at compile time. They used this representation to improve the detection of fully parallel loops at compile-time, and they also suggested using the approach to create run-time checks to determine parallelism. Hybrid Analysis [23, 24] uses something they call a uniform set representation to perform as much data dependence analysis for parallelism detection at compile time as possible with generated code that performs any necessary dynamic checks.

The Sparse Polyhedral Framework [25, 26] took the use of uninterpreted functions in data dependence relations further than just detecting parallelization to the determination of constraints on index arrays generated by inspector/executor strategies. Performing a data dependence analysis with petit (a tool available with the Omega project [27]) and specifying the transformations as relations, a manual proof of correctness using pre and post conditions was performed for the full sparse tiling transformation applied to the Gauss-Seidel computation [28].

As an example of deriving such constraints, the data dependence relation for the original kernel in the wavebench micro-benchmark in Figure 3 is:

$$\begin{aligned}\{[p] \to [p'] \mid (p < p') \wedge (0 \leq p, p' < nnz) \\ \wedge((row(p) = row(p')) \vee (row(p) = col(p')) \\ \vee(col(p) = row(p')) \vee (col(p) = col(p')))\},\end{aligned}$$

where $row()$ and $col()$ are uninterpreted function symbols representing the corresponding index arrays. Note that the variable `sum` is privatizable. It would also be possible to detect that the full loop is a reduction, but for simplicity of the micro-benchmark we ignore that and derive the dependencies caused by the increment on the accesses to the `data` array.

The wavefront parallelization transformation can be expressed in the Sparse Polyhedral Framework using the following relation: $\{[p] \to [w, p] \mid w = wave(p)\}$, where w is the iterator over wavefronts in Figure 3 and the p loop in the transformed code is parallel. The transformed space $[w, p]$ is executed in lexicographical order except for the p loop, which since it has been specified as parallel, will execute in parallel for each instance of w. The waves loop with iterator w is executed in sequential order. Applying this transformation to the data dependence relation for the original loop results in the transformed dependencies:

$$\begin{aligned}\{[w, p] \to [w', p'] \mid w = wave(p) \wedge w' = wave(p') \\ \wedge(p < p') \wedge (0 \leq p, p' < nnz) \\ \wedge((row(p) = row(p')) \vee (row(p) = col(p')) \\ \vee(col(p) = row(p')) \vee (col(p) = col(p')))\}.\end{aligned}$$

The transformed data dependence relation, and thus the transformation that created it, is legal if the dependencies are all lexicographically positive. For this example, this leads to the constraint that if there was an original dependence between iteration p and p', then it must be the case that $wave(p)$ is strictly less than $wave(p')$ to ensure that iteration p is executed before iteration p' in the new wavefront parallel schedule.

Current inspector/executor implementations of wavefront schedules such as [14, 15] present their inspector algorithms and argue that they place iterations from the original loop into wavefronts so that the original dependencies are satisfied. This approach for determining the correctness of inspector/executor transformations assumes that the data dependence analysis (often performed manually for a particular class of computations) is correct and uses informal arguments to show the inspector implementation is correct.

In this paper, we present an alternative approach that enables formally proving the correctness of an inspector/executor transformation without assuming correct data dependence analysis or a correct inspector implementation. We use a representation for the original code and the executor that separates the reads from the writes in array assignment statements[4]. We then have a function that interprets the PseudoC code while creating an action graph that tracks all reads and writes. An action graph includes the data dependence relation information as if the data dependence relation with uninterpreted functions had been explicitly realized at interpretation time, or run-time. Each node in the action graph also maintains enough information so that the action graph is executable. We have shown that execution of an action graph is equivalent to the PseudoC code execution. Therefore, if transformations result in equivalent action graphs, they are legal.

4 Expressing Codes in PseudoC

Figure 5 shows the grammar for our PseudoC language, which is rich enough to represent original code with for loops and indirect array references, and the case study inspector and executor. The operational semantics for some example PseudoC statements are shown in Figure 6. The semantics relates pairs of memories and pieces of PseudoC syntax.

The original code and the executor perform assignments to array elements via the Assign production rule where the reads are represented as $\langle dexpr \rangle$s so that the operational semantics can indicate the ability for reads and writes from different iterations within a parallel loop to be interleaved. The first inference rule in Figure 6 is for the `Assign` statement, which assigns a value to an array element. Given that the conditions above the horizontal rule are true, then the memory state m_0 will updated to the memory state m and the `Assign` statement will be replaced with the `Done` assignment, representing successful termination. The value $\mathsf{V}(i)$ is an expression that has been fully evaluated to an integer i, in this case representing the index at which the array will be updated.

When a `ForLoop` or a `ParLoop` is interpreted, the loop body is replicated as many times as there are loop iterations. (In the original code and the executor, we assume that all loops are bounded). Each copy of the loop body has instances of the iterator variable replaced the appropriate value from the iteration's domain.

[4] Handling of scalar assignments is forthcoming, but those that are not privatizable result in little if any parallelism in the loop.

The copies are then collected underneath a `Seq` or `Par` construct (Figure 6 shows a `ForLoop` and therefore a `Seq` construct).

The final two rules in Figure 6 illustrate how loop bodies then evaluate with `Seq` and `Par` constructors. In the `Seq`, the first iteration has to evaluate to completion before any others can evaluate. In the `Par`, any loop body can take a step at any stage, representing the possible interleaving that parallel loops afford.

```
⟨value⟩ ::= Int ⟨int⟩
          | Real ⟨real⟩
          | Bool ⟨bool⟩
          | Array ⟨value list⟩
          | Error

⟨expr⟩ ::= VarExp ⟨ident⟩
         | ISub ⟨ident⟩ ⟨expr⟩
         | ⟨expr⟩ ⟨binop⟩ ⟨expr⟩
         | ⟨unaryop⟩ ⟨expr⟩
         | Value ⟨value⟩

⟨dexpr⟩ ::= DValue ⟨value⟩
          | ARead ⟨ident⟩ ⟨expr⟩
          | VRead ⟨ident⟩

⟨domain⟩ ::= D ⟨expr⟩ ⟨expr⟩

⟨stmt⟩ ::= Assign ( ⟨ident⟩, ⟨expr⟩ ) ⟨dexpr-list⟩
           ⟨rhs-f⟩
         | AssignVar ⟨ident⟩ ⟨expr⟩
         | IfStmt ⟨expr⟩ ⟨stmt⟩ ⟨stmt⟩
         | Malloc ⟨ident⟩ ⟨int⟩ ⟨value⟩
         | ForLoop ⟨ident⟩ ⟨domain⟩ ⟨stmt⟩
         | ParLoop ⟨ident⟩ ⟨domain⟩ ⟨stmt⟩
         | Seq ⟨stmt-list⟩
         | Par ⟨stmt-list⟩
         | Abort
         | Done
```

Fig. 5: The abstract syntax tree grammar for the PseudoC language. To break out the reads from the writes, the array element assignment `Assign` and the variable assignment `AssignVar` statements include a list of ⟨*dexpr*⟩s and then a lambda function called the right-hand side function ⟨*rhs-f*⟩ that can map a list of ⟨*value*⟩s to a single value.

$$\frac{(\forall r \in \mathit{reads}.\ \mathsf{isValue}(r)) \quad \mathsf{upd\_array}(m_0, id, i, \mathit{rhsf}(\mathit{reads})) = m}{(m_0, \mathtt{Assign}((id, \mathsf{V}(i)), \mathit{reads}, \mathit{rhsf})) \longrightarrow (m, \mathtt{Done})}$$

$$\frac{\mathsf{evalexpr}(m, e_1) = l \quad \mathsf{evalexpr}(m, e_2) = u}{(m, \mathtt{ForLoop}(id, \mathsf{D}(e_1, e_2), s)) \longrightarrow (m, \mathsf{Seq}[s[id := l], s[id := l+1], \ldots, s[id := u-1]])}$$

$$\frac{(m_0, s_1) \longrightarrow (m, s)}{(m_0, \mathsf{Seq}\ [s_1, s_2, ..., s_n] \longrightarrow (m, \mathsf{Seq}\ [s, s_2, ..., s_n])}$$

$$\frac{(m_0, c_0) \longrightarrow (m, c)}{(m_0, \mathsf{Par}\ [p, ..., c_0, ..., s] \longrightarrow (m, \mathsf{Par}\ [p, ..., c, ..., s])}$$

Fig. 6: Example operational semantic rules for PseudoC.

5 Proof of Correctness

5.1 Original Code and Executor Equivalence

The heart of the correctness problem is verifying that, after the inspector has done its work, the original code and the executor have the same effect. However, the original code and the executor may execute their array assignments in radically different orders. To formally capture the equivalence between old and new programs we map their syntax into a domain where non-conflicting assignments can be executed in any order and compare the results there.

The domain we choose is the *nested action graph*. The nodes of these graphs are either (recursively) sub-graphs themselves, or at the base case, pending array assignments. Each node of the latter sort contains a list of read locations, a write location and a function for turning the read values into a value to be written. The read list of a sub-graph node is the union of all the reads in the sub-graph. Sub-graph nodes also contain write lists, consisting of all the writes made in the sub-graph. There is an edge between two nodes if they have overlapping write lists, or if one has a read contained in the other's write list. Given such a pair of nodes, the direction of the edge depends on the ordering between those nodes' assignments in the original program.

We refer to a node containing a pending assignment as *atomic*. Though such a node contains memory reads and an eventual write that might interleave with other nodes' execution, such a node can only be executed when the dependence edges allow it, meaning that modeling its execution as atomic is justified.

The behavior of graph on an input memory is computed by repeatedly removing nodes with no incoming edges, and executing their assignments. Though non-deterministic in its choice of nodes, this behaviour can only result in one possible output memory. Similarly, parallel programs may exhibit transient non-determinism, but if they are race-free, their execution will have only one possible eventual result. If a program is not race-free, the action graph semantics categorizes this as an error: such a program has no corresponding action graph.

5.2 Creating Action Graphs from PseudoC

The function `graphOf` maps PseudoC programs to graphs. It is defined recursively over the syntax of an executor program. Not all forms of PseudoC program are convertible to action graphs; in particular, original code and executors cannot include unbounded loops (*e.g.*, `while` loops), nor `Malloc` calls.

Because the execution of `graphOf` must both unwind executors with parameterized (but bounded) loops, and decide which branch of `if`-statements are taken, the `graphOf` function takes an initial memory value as a parameter. It also computes the memory that will result from the execution of the graph, so that updated memory values can be passed through the program recursively. This can be seen in the clause for the `ForLoop` mapping in Figure 7.

The treatment of `for`-loops also demonstrates the use of and need for *nested* action graphs: a loop's body may contain multiple atomic assignments, or indeed

```
graphOf $m_0$ (ForLoop $v$ $d$ $body$)  =
  do
    $dvs$  ←  dvalues $m_0$ $d$;
    FOLDL ($\lambda acc\ v.$
              do
                $(m, g)$  ←  $acc$;
                $(m', sg)$  ←  graphOf $m$ ($body[vnm := v]$);
                SOME( $m'$,
                        $g \oplus \langle$ reads := greads($sg$); writes := gwrites($sg$);
                             data := $sg\rangle$)
              od)
           (SOME($m_0, \varepsilon$))
           $dvs$
  od
```

Fig. 7: Calculating the graph of a `for` loop. This is a simplified version of the function defined in HOL4. The FOLDL folds (or "reduces") the computation of the graph. It begins with the input memory m_0 and an empty graph (ε). Each iteration over the loop values (dvs) then returns an updated version of each, calculating a sub-graph (sg), embedding this into a node (between the angle-brackets), and adding that node to the accumulating graph.

more, nested loops. When we perform our wavefront analysis, we want to schedule entire loop bodies as a unit. The unit of computation with action graphs is a single node, so loop bodies consisting of multiple atomic actions need to have those actions bundled together into a sub-graph that is then treated as a single node at the higher level.

The graph construction for parallel loops is given in Figure 8.

Theorem 1. *For all PseudoC programs p, if there exists a g such that $g =$ graphOf(p), then the program has only one possible behavior, which is the same as that obtained by executing the graph g.*

(This result has been mechanically verified in HOL4 for a version of the graphOf function which maps to non-nested, "flat" graphs. The mechanical verification of the translation to nested graphs is underway.)

5.3 Characterizing Equivalence

With a correct action graph semantics in hand, the next step in our approach to showing inspector/executor correctness is to characterize the conditions under which the equivalence will hold. In essence, the task is to prove a theorem of the form

$$\langle\text{inspector post-condition}\rangle \;\Rightarrow\; \mathsf{graphOf}(old) = \mathsf{graphOf}(new)$$

We have not yet proved a theorem of this shape for the parallelizing wavefront transformation, but we can illustrate our approach with a simple example that

```
graphOf $m_0$ (ParLoop $vnm$ $d$ $body$) =
  do
    $dvs$ ← dvalues $m_0$ $d$;
    $ns$ ← MAP ($\lambda v.$ do ($m, sg$) ← graphOf $m_0$ ($body[vnm := v]$);
                                  SOME(⟨ reads := greads($sg$); writes := gwrites($sg$);
                                         data := $sg$⟩)
                         od) $dvs$;
    assert($\forall i\,j.\ i < j < |ns| \Rightarrow ns_i \not\leadsto_t ns_j$);
    $g$ ← SOME(FOLDR($\lambda a\,g.\ a \oplus g$) $\varepsilon$ $ns$);
    $m$ ← nagEval $g$ (SOME $m_0$);
    SOME($m, g$)
  od
```

Fig. 8: Calculating the action graph of a ParLoop. The sub-graphs are calculated independently, under the MAP, which iterates over the loop's domain values *dvs*. The memory resulting from each sub-graph computation (m) is ignored rather than being the basis for the execution of the next iterations as in the handling of ForLoop. The assert checks that none of the calculated sub-graphs have a dependence between each other. The final graph is assembled by adding all of the nodes in *ns* to an empty graph, and the final memory is calculated by evaluating that final graph with the graph evaluation function, nagEval.

has been mechanically verified. This iteration reordering imagines a loop with a body of the form

```
A[$W$(i)] = ... A[$R_1$(i)] ... A[$R_n$(i)] ...
```

where the assignment target is a function W of the iteration index (perhaps *via* an indirection array), and where the n reads are mediated through the various R_i functions (again, possibly indirection arrays).

We have mechanically proved the following:

Theorem 2. *With W and R specifying the program's write and read information as above, let ddepR(W, R, i, j) be true iff iteration i comes before iteration j and they have the same write, or if one writes to one of the other's reads. Then*

$$\begin{array}{l} \quad (\forall i\,j.\ i < j < N \wedge \mathsf{ddepR}(W, R, i, j) \Rightarrow \delta^{-1}(i) < \delta^{-1}(j)) \wedge \\ \quad \delta : \{0 \ldots N-1\} \rightarrow \{0 \ldots N-1\} \text{ a bijection} \\ \Rightarrow \\ \quad (\texttt{for(i=0;i<N;i++)}\ \{A[W(\texttt{i})] \texttt{ = } \ldots A[R_j(\texttt{i})] \ldots\}) = \\ \quad (\texttt{for(i=0;i<N;i++)}\ \{A[W(\delta(\texttt{i}))] \texttt{ = } \ldots A[R_j(\delta(\texttt{i}))] \ldots\}) \end{array}$$

Alternatively: as long as the inverse of the new indirection array/function does not change the order of iterations that are connected with a data dependence, the new executor will be equivalent to the old.

The proof of this result depends heavily on the existence of the action graph domain into which the for-loops are mapped.

5.4 Verifying the Inspector

When a theorem of the above form has been proved, the final stage of the verification can be performed. That is, we must now confirm that the inspector establishes the desired post-condition. Note that, as in the example above, we aim to prove as general a post-condition as possible. If this is done, correct inspectors may compute different reorderings or parallelizations.

Continuing our iteration reordering example, we have mechanically verified that a permissible inspector is one that ordered iterations by their wavefront number. This is our last mechanically verified theorem:

Theorem 3. *Sorting the iterations of a simple for loop with respect to the iterations' wavefront number generates a δ-function that is both a bijection over the `for`-loop's domain, and respects the dependence relation. This δ can then be used to correctly reorder iterations, as per the equivalence of Section 5.3.*

Even this much leaves aspects of the inspector's implementation under-specified. Open questions include, how should an inspector correctly generate iterations' wave numbers, particularly as it will not have access to the complete action graph since that would be too expensive to compute at runtime?

5.5 Limitations and Future Work

We do not expect that correct transformations will ever be easy to verify. Rather, our goal is to establish a verified library, one that couples verified executor transformations with families of verified inspectors. These inspectors will have been to shown to meet the transformations' preconditions, allowing programmers and compilers to deploy them safely.

The transformations will need to be as general as possible in two ways: with general preconditions allowing multiple inspector strategies, and embodying general patterns of code transformations.

Our mechanization effort is still ongoing, but we are confident that the approach sketched here is a viable strategy. There is still more to be done:

- we want to express inspector implementations in a higher level language than PseudoC, one where relevant notions such as binary relations and the operations on them are first-class;
- nonetheless, such a high-level language should have a verified translation to PseudoC;
- we would like to formally capture the compile-time syntactic analysis that leads to the inspector's static knowledge of where reads and writes occur are possible.

6 Related Work

Inspector/executor strategies were developed to find shared-memory, partial parallelism in `doacross` loops [1, 2] and to implement distributed memory parallelism between `doall` loops that accessed distributed data [3]. Later data reordering transformations [4–7] and more complex sparse tiling transformations [8–10] were developed.

Other related work includes formally showing the correctness of compilers and translators, which is an exciting area of recent research progress. The CompCert compiler by Leroy [29] has been formally shown to correctly translate a mildly restricted subset of C to x86 assembly code with equivalent semantics. The CompCert compiler performs register allocation, instruction scheduling, and some data-flow optimizations, but does not include any parallelization optimizations.

In the context of irregular applications with indirect array accesses, Gilad and Bodik [30] have developed a domain-specific language LL for transforming dense matrices into various sparse matrix formats and showing the equivalence of computations on the sparse matrix format to the corresponding computation on the dense matrix format. This work is relevant to inspector/executor strategies as the conversion between sparse matrix formats is typically done with an inspector. However, this work does not look at parallelization transformations.

7 Conclusions

Parallelization transformations in compilers are typically determined correct based on data dependence analysis and reasoning about how parallelization, possibly in combination with other transformations, continue to satisfy such dependencies. This approach assumes too much especially in the context of irregular applications with indirect memory references where it is necessary to assume that the data dependence analysis is correct and that the inspector code is correct as well. In this paper, we present a new approach for determining the correctness of an inspector/executor strategy, ultimately aiming to provide a library of verified inspector/executor parallelization transformations. All code (the original code, inspector, and executor) are represented in our PseudoC language, which separates reads from writes. This language has two formal semantics: a small-step operational semantics where reads and writes in separate iterations of a parallel loop can execute concurrently, and one in terms of action graphs. The action graph semantics allows the proof of program equivalences subject to certain conditions. Inspector correctness then depends on showing that these conditions are met. Important preliminaries and simple examples have already been fully mechanized in HOL4; and work toward a fully verified wavefront parallelization transformation is ongoing.

8 Acknowledgements and Availability

This project is supported by a Department of Energy Early Career Grant DE-SC0003956. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

Our source code is available from github at `https://github.com/mn200/inspector-strategies`

References

1. Saltz, J.H.: Aggregation methods for solving sparse triangular systems on multiprocessors. SIAM Journal on Scientific and Statistical Computing **11**(1) (1990) 123–144
2. Koelbel, C., Mehrotra, P.: Compiling global name-space parallel loops for distributed execution. Parallel and Distributed Systems, IEEE Transactions on **2**(4) (Oct 1991) 440–451
3. Saltz, J., Chang, C., Edjlali, G., Hwang, Y.S., Moon, B., Ponnusamy, R., Sharma, S., Sussman, A., Uysal, M., Agrawal, G., Das, R., Havlak, P.: Programming irregular applications: Runtime support, compilation and tools. Advances in Computers **45** (1997) 105–153
4. Mitchell, N., Carter, L., Ferrante, J.: Localizing non-affine array references. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Los Alamitos, CA, USA, IEEE Computer Society (October 1999) 192–202
5. Ding, C., Kennedy, K.: Improving cache performance in dynamic applications through data and computation reorganization at run time. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, ACM (May 1999) 229–241
6. Han, H., Tseng, C.W.: Efficient compiler and run-time support for parallel irregular reductions. Parallel Computing **26**(13–14) (December 2000) 1861–1887
7. Mellor-Crummey, J., Whalley, D., Kennedy, K.: Improving memory hierarchy performance for irregular applications using data and computation reorderings. International Journal of Parallel Programming **29**(3) (2001) 217–247
8. Douglas, C.C., Hu, J., Kowarschik, M., Rüde, U., Weiß., C.: Cache Optimization for Structured and Unstructured Grid Multigrid. Electronic Transaction on Numerical Analysis (February 2000) 21–40
9. Strout, M.M., Carter, L., Ferrante, J., Freeman, J., Kreaseck, B.: Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In: Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC), Berlin / Heidelberg, Springer (July 2002)
10. Mohiyuddin, M., Hoemmen, M., Demmel, J., Yelick, K.: Minimizing communication in sparse matrix solvers. In: Supercomputing, New York, NY, USA, ACM (2009)
11. Das, R., Uysal, M., Saltz, J., Hwang, Y.S.S.: Communication optimizations for irregular scientific computations on distributed memory architectures. Journal of Parallel and Distributed Computing **22**(3) (1994) 462–478
12. Slind, K., Norrish, M.: A brief overview of HOL4. In Mohamed, O.A., Muñoz, C., Tahar, S., eds.: Theorem Proving in Higher Order Logics, 21st International Conference. Volume 5170 of Lecture Notes in Computer Science., Springer (2008) 28–32 See also `http://hol.sourceforge.net`.
13. Saltz, J.H.: Automated problem scheduling and reduction of communication delay effects. Technical report, Yale University (1987)
14. Rauchwerger, L., Amato, N.M., Padua, D.A.: A scalable method for run-time loop parallelization. International Journal of Parallel Programming **23**(6) (1995) 537–576

15. Zhuang, X., Eichenberger, A., Luo, Y., O'Brien, K., O'Brien, K.: Exploiting parallelism with dependence-aware scheduling. In: International Conference on Parallel Architectures and Compilation Techniques (PACT), Los Alamitos, CA, USA, IEEE Computer Society (2009) 193–202
16. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1) (December 2011) 1:1–1:25
17. Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D.A.: Automatic program parallelization. Proceedings of the IEEE **81**(2) (1993) 211–243
18. Lengauer, C.: Loop parallelization in the polytope model. In: CONCUR '93, Lecture Notes in Computer Science 715, Springer-Verlag (1993) 398–416
19. Feautrier, P.: Automatic parallelization in the polytope model. In: The Data Parallel Programming Model. (1996) 79–103
20. Kennedy, K., Allen, J.R.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
21. Yang, Y.Q., Ancourt, C., Irigoin, F.: Minimal data dependence abstractions for loop transformations: Extended version. International Journal of Parallel Programming **23**(4) (1995) 359–388
22. Pugh, W., Wonnacott, D.: Nonlinear array dependence analysis. In: Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, Troy, New York (May 1995)
23. Rus, S., Hoeflinger, J., Rauchwerger, L.: Hybrid analysis: static & dynamic memory reference analysis. International Journal Parallel Programming **31**(4) (2003) 251–283
24. Oancea, C.E., Rauchwerger, L.: Logical inference techniques for loop parallelization. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation. PLDI '12, New York, NY, USA, ACM (2012) 509–520
25. Strout, M.M., Carter, L., Ferrante, J.: Compile-time composition of run-time data and iteration reorderings. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), New York, NY, USA, ACM (June 2003)
26. Strout, M.M., LaMielle, A., Carter, L., Ferrante, J., Kreaseck, B., Olschanowsky, C.: An approach for code generation in the sparse polyhedral framework. Technical Report CS-13-109, Colorado State University (December 2013)
27. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega calculator and library, version 1.1.0 (November 1996)
28. Strout, M.M., Carter, L., Ferrante, J.: Proof of correctness for sparse tiling of gauss-seidel. Technical report, UCSD Department of Computer Science and Engineering, Technical Report #CS2003-0741 (April 2003)
29. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7) (July 2009) 107–115
30. Arnold, G., Hölzl, J., Köksal, A.S., Bodík, R., Sagiv, M.: Specifying and verifying sparse matrix codes. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP). ICFP '10, New York, NY, USA, ACM (2010) 249–260