Mesh Independent Loop Fusion for Unstructured Mesh Applications*

Carlo Bertolli
Department of Computing
Imperial College London
c.bertolli@imperial.ac.uk

Adam Betts
Department of Computing
Imperial College London
abetts@imperial.ac.uk

Gihan R. Mudalige
Oxford e-Research Centre
University of Oxford
gihan.mudalige@oerc.ox.ac.uk

Paul H.J. Kelly
Department of Computing
Imperial College London
phjk@imperial.ac.uk

Michael B. Giles
Oxford e-Research Centre
University of Oxford
mike.giles@maths.ox.ac.uk

ABSTRACT

Applications based on unstructured meshes are typically compute intensive, leading to long running times. In principle, state-of-the-art hardware, such as multi-core CPUs and many-core GPUs, could be used for their acceleration but these esoteric architectures require specialised knowledge to achieve optimal performance. OP2 is a parallel programming layer which attempts to ease this programming burden by allowing programmers to express parallel iterations over elements in the unstructured mesh through an API call, a so-called OP2-loop. The OP2 compiler infrastructure then uses source-to-source transformations to realise a parallel implementation of each OP2-loop and discover opportunities for optimisation.

In this paper, we describe how several compiler techniques can be effectively utilised in tandem to increase the performance of unstructured mesh applications. In particular, we show how whole-program analysis — which is often inhibited due to the size of the control flow graph — often becomes feasible as a result of the OP2 programming model, facilitating aggressive optimisation. We subsequently show how whole-program analysis then becomes an enabler to OP2-loop optimisations. Based on this, we show how a classical technique, namely loop fusion, which is typically difficult to apply to unstructured mesh applications, can be defined at compile-time. We examine the limits of its application and show experimental results on a computational fluid dynamic application benchmark, assessing the performance gains due to loop fusion.

*This research is partly funded by EPSRC (grant reference numbers EP/I00677X/1, EP/I006079/1), the UK Technology Strategy Board, and Rolls Royce plc through the SILOET programme.

CF'12, May 15–17, 2012, Cagliari, Italy.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

Keywords

Unstructured Mesh Applications, Compilers, Loop fusion, Whole Program Control Flow Analysis

1. INTRODUCTION

An unstructured mesh is an irregular collection of linked vertices, edges, and polygons which provides an effective abstraction within the computational sciences. For example, it can be utilised in the finite volume method that gives approximate solutions to Partial Differential Equations (PDEs). To achieve a reasonable degree of accuracy, however, requires that the unstructured mesh be composed of many millions of elements, leading to a large amount of computation. One such industrial application is HYDRA, which is used by Rolls Royce Plc. in turbomachinery design and is the driving force of our research. In HYDRA the mesh can grow to sizes of over 100 million edges. Furthermore the HYDRA software is extremely complex, with about 1000 separate parallel loops over the mesh. For this reason, HYDRA is currently accelerated with MPI through the OPlus library [6, 5], a predecessor of our research work.

One potential solution to the speedup issue is to utilise multi- and many-core processors, especially because unstructured mesh computations are massively data parallel. However, this assumes that the programmer retains sufficient expertise to program to such a specialised architecture; for example, using the CUDA programming model. A more crucial drawback is that performance portability is greatly inhibited: re-targeting the code base towards different backends requires a fresh implementation, and the idiosyncrasies imposed by particular hardware to achieve optimal performance are unlikely to be satisfied.

A programming layer called OP2 [16, 11, 10] attempts to simultaneously ease the programming burden and performance portability problems. It provides abstractions to declare unstructured meshes in an intuitive manner (i.e., as we show in Section 3, through the sets comprising the mesh) and to express parallel computations on the mesh in terms of iterations over particular sets. A program containing calls to

the OP2 API is then compiled using source-to-source transformation tools, before finally being compiled to the target architecture using a vendor-specific compiler. In this way, the source-to-source translator becomes crucial to the optimisation process because it knows which parallel computations are to be performed on the mesh (via calls to the OP2 API) and the architecture of choice.

Using OP2 a single application program (written using the OP2 API) can be transformed to a range of diverse architectures, including multi- and many-core systems. The optimisation challenge is to deliver near optimal performance for each single architecture, and in turn achieve performance portability. For the target applications, loop optimisations are the key to achieve this goal.

This paper investigates the optimisation opportunities applicable to programs written in OP2. With respect to the optimisations, we first show how much of the program can be sliced [18] due to assumptions in the OP2 programming model, thereby making whole-program analysis feasible. Using the whole-program Control Flow Graph (CFG) we then show how to apply loop fusion used to accelerate the parallel loops over the mesh.

The general optimisation problem that we investigate in this paper can be formulated as the compile-time fusion of two loops using non-affine array accesses such as the following ones:

```
! loop over edges
do i = 1, numberOfEdges
 A[n(i,1)] = kernel1 (B[m(i,1), B[m(i,2)], C[i])
enddo

! loop over cells
do j = 1, numberOfCells
 D[j] = kernel2 (B[p(j,1), B[p(j,2)], A[j])
enddo
```

In this example A, B, C, and D are arrays defined for a mesh set, e.g. there is a tuple of elements in the A array for each cell in the mesh. With n, m, and p we denote *mappings* which relate mesh sets between themselves. For instance, n maps an edge identifier to a cell identifier (the second argument of n is used to select one of the cells linked to an edge) and it is used to access A when iterating over edges. Finally, *kernel1* and *kernel2* are user-defined kernel functions, defined for the generic mesh set element (either an edge or a cell).

Obviously, this general loop fusion case cannot be achieved at compile-time. A fundamental piece of information is unknown until run-time, i.e. array accesses derived from the mesh. In this paper we precisely characterise which simpler sub-cases can and cannot be subject to compile time loop fusion. More specifically, the paper concludes that:

- If two loops are iterating over the same iteration space, and one of them is not using indirections (i.e. non-affine array accesses), then compile-time loop fusion is possible. This can be done without the knowledge of the specific details of the meshes on which the program executes, i.e. in a *mesh independent* manner.
- In all other cases, loop fusion can only be performed at run-time, when mesh information (i.e. non-affine accesses) is known. We will address such cases in future work.

The actual application of the first case depends on performance considerations, which are based on: (i) increased data locality, in case the two loops access same datasets; (ii) to a lower extent, reduced global synchronisations.

The described loop fusion case is a recurring one in large-scale applications. A fundamental trade-off in performance must be studied to understand if loop fusion is actually delivering a performance improvement. In the specific case of OP2 (but equivalent solutions must be applied to any parallel implementation of loops with non-affine array accesses) a loop using indirections is implemented using colouring to control the parallelism. This guarantees that no race conditions can happen during the execution, as two iterations incrementing a common memory location are scheduled serially. Colouring implies that parallelism is reduced from the iteration space size to the average number of iterations of the same colour. In contrast, the implementation of a loop without indirections (henceforth called "direct loop") can use a parallelism degree as high as the size of the iteration set. If two such loops are fused, the resulting fused loop uses indirections, and the implementation is based on colouring. There is hence a decrease in the maximum achievable parallelism degree when comparing the two loops against the single fused one. This must be balanced by the benefits gained in the fusion. In this paper we show that in a two representative examples improvement after loop fusion is obtained.

Another fundamental step to achieve compile-time loop fusion is given by the ability of the compiler to define the relation between two loops, in terms of abstract concepts as iteration spaces, indirections used to access datasets, and type of operations. This is obtained in OP2 through the use of access descriptors, which precisely characterise dependence. Using this abstraction, the compiler is fully capable of deriving if two loops can be fused without further analysis.

We validate our approach by applying loop fusion to a classical computational fluid dynamics application, called *Airfoil*. This is a standard benchmark which we use to characterise the performance of a section of the significantly more complex HYDRA case. Therefore, any performance improvement obtained for the Airfoil application will be directly mapped in specific sections of HYDRA. Our experiments show the performance gains that we obtain on two main-stream architectures including an Nvidia M2050 GPU, and an Intel Xeon X5650 "Nehalem" multicore, with 12 2-way hyperthreaded cores.

The contributions of this paper are:

- We show how program slicing can be defined for OP2 programs, to identify loop optimisation opportunities.
- We show how OP2 access descriptors permit the compiler to analyse if loop fusion can be applied, and what benefits can be obtained.
- We precisely identify in which cases loop fusion can be applied without the knowledge of the mesh, and which other cases require instead that knowledge.
- We characterise experimentally the performance benefits due to loop fusion in two representative examples.

The remainder of the paper is organised as follows: Section 2 places OP2 into context by reviewing related work. Section 3 then presents the OP2 programming layer, outlines the assumptions which are critical to the optimisations

proposed, and gives an overview of our compiler infrastructure supporting OP2. Following that, Section 4 formalises the program model utilised by our optimisation framework, and it presents how whole-program analysis and slicing and loop optimisations are applied to the program model. Section 5 focuses on loop fusion, motivating its theoretical performance gain in the OP2 implementation, and describing its feasibility by analysing different loop types. Section 6 evaluates our approach before conclusions are finally drawn in Section 7.

2. RELATED WORK

OP2 is the second iteration of OPlus (Oxford Parallel Library for Unstructured Solvers) [6, 5]. OPlus provided an abstraction framework for performing unstructured mesh based computations across a distributed-memory cluster, using a traditional MPI library approach. It is currently used as the underlying parallelisation library for HYDRA [15, 9] a CFD application used in turbomachinery design at Rolls-Royce plc. OP2 builds upon the features provided by its predecessor but develops an "active" library approach with code generation to exploit parallelism on heterogeneous multi-core/many-core architectures.

Although OPlus pre-dates it, both OPlus and OP2 can be viewed as an instantiation of the AEcute (access-execute descriptor) [13] programming model that separates the specification of a computational kernel with its parallel iteration space, from a declarative specification of how each iteration accesses its data. The decoupled Access/Execute specification in turn creates the opportunity to apply powerful optimisations targeting the underlying hardware.

A number of related research projects have implemented similar programming frameworks. The most comparable of these is LISZT [7], which is a domain-specific language based on the SCALA language targeting unstructured mesh applications. Unlike OP2, LISZT attempts to synthesise dependence information from the program, by exposing programmers a stricter programming model. Another notable framework is described in [14], which shares with OP2 the library approach but it is based on a C++ framework.

3. THE OP2 MODEL OF COMPUTATION

From the programmer's point of view, OP2 is an Application Programming Interface (API) enabling the following to be declared:

1. The sets constituting the topology of the mesh. These normally include its vertices, edges, and polygons, but might incorporate various subsets, such as boundary edges.

2. How distinct sets relate to each other via mapping functions. A typical example is the mapping from an edge to the two vertices to which it is incident.

3. The data associated with elements in the sets, e.g. vertex co-ordinates or edge weights.

4. How a specified function can be applied *in parallel* to every element of a particular set.

We demonstrate OP2 through the C++ code shown in Listing 3[1], which will serve as a running example throughout the paper. In this example, we omit the user kernel code, and we focus on the OP2 details useful to show loop fusion feasibility in the next sections. The sample application is Airfoil, a non-linear 2D inviscid airfoil code that uses an unstructured mesh. It is a very simple application, but acts as a forerunner for testing the OP2 library in our compiler framework due to its strong similarities to HYDRA, the CFD application used at Rolls Royce plc. for the simulation of jet engines.

At the top of Listing 3 is one of the user-supplied kernels, `save_soln`, contained in the Airfoil application. It has two formal parameters, namely `p_q` and `p_qold`, which are both arrays of double precision floating-point values. In the function `main`, three sets are defined (`cells`, `edges`, and `bedges`) such that their cardinalities are set to 2, 100, and 10, respectively. Next, data contained in the local variable declarations `q` and `qold` are associated with the set `cells` through `op_decl_dat`, which specifies that there is a vector of four elements associated to each set element. That is, the first element of `cells` has initial values $1.0, 2.0, 3.0, 4.0$, while the second element has initial values $5.0, 6.0, 7.0, 8.0$. The declaration of an `op_map` is used to define a mapping between two sets. For instance, `pcell` is declared as a mapping between cells and nodes, identifying which nodes (i.e. their identifiers) are at the border of each cell, and thus also how cells are connected between themselves through nodes. This mapping information is obtained from an input array passed to the related declaration call, and it defines the topology of the unstructured mesh. The use of indirect dataset addressing also classifies these applications as *irregular*.

We need to clarify at this point the difference between a loop arising from a programming-language construct, e.g. due to *for* statements, and a loop arising from the OP2 programming layer. We shall refer to the former as a loop and the latter as an OP2-loop.

The remainder of the code contains a doubly-nested loop with OP2-loops (`op_par_loop`) that specify parallel iteration over the declared sets. For example, the first call to `op_par_loop` states that the `save_soln` function is to be applied to each element of the set `cells`. The next six actual arguments in the `op_par_loop` call indicate the OP2 data expected by the kernel function and provide information pertaining to their access. These are the so-called *access descriptors*.

Since `save_soln` has two formal parameters, two OP2 data types, `p_q` and `p_qold`, are supplied. This means that there are three actual arguments in the OP2-loop per kernel formal parameter, which we refer to as an OP2 argument group. The first OP2 argument group (`p_q, -1,OP_ID, OP_READ`) states that the OP2 data type `p_q` is *directly* accessed and is read within the kernel function. Direct access is expressed by `OP_ID` and arises from the fact that the sets over which `save_soln` iterates, and to which `p_q` is associated, are cells; in other words, no mapping is required. In this case, we say that the OP2-loop is *direct*, whereas an OP2-loop containing mappings is *indirect*. The final argument in this grouping, `OP_READ`, signifies read access and arises from the fact that `p_q` in `save_soln` is an r-value. The second OP2 argument group (`p_qold, -1, OP_ID, OP_WRITE`) expresses similar semantics, except `p_qold` is written, i.e. `p_qold` is an l-value in `save_soln`. Note, therefore, how the access patterns of an OP2-loop are explicitly expressed, thereby

[1]Unnecessary OP2 declarations are omitted in the code to make reading easier and confined to essential information.

```
int main (void) {
 // Declare sets
 op_set cells  = op_decl_set (2);
 op_set edges  = op_decl_set (100);
 op_set bedges = op_decl_set (10);
 op_set nodes = op_decl_set (100);

//Declare maps
 op_map pcell = op_decl_map (cells, nodes, 4,
      map_arr);

 // Declare data
 double * q = {1.0, 2.0, 3.0, 4.0,
                5.0, 6.0, 7.0, 8.0};
 op_dat p_q = op_decl_dat (cells, 4, q);

 double * qold = {10.0, 20.0, 30.0, 40.0,
                  50.0, 60.0, 70.0, 80.0};
 op_dat p_qold = op_decl_dat (cells, 4, qold);

 double *x = {..}
 op_dat p_x = op_decl_dat (nodes, 2, x);

// ...other declarations...

 for (int i = 0; i < 1000; ++i) {

  op_par_loop(save_soln, cells,
    op_arg_dat(p_q,     -1, OP_ID,  OP_READ),
    op_arg_dat(p_qold, -1, OP_ID, OP_WRITE));

  for (int j = 0; j < 2; ++j) {
     op_par_loop(adt_calc,cells,
       op_arg_dat(p_x,    0,pcell, OP_READ),
       op_arg_dat(p_x,    1,pcell, OP_READ),
       op_arg_dat(p_x,    2,pcell, OP_READ),
       op_arg_dat(p_x,    3,pcell, OP_READ),
       op_arg_dat(p_q,   -1,OP_ID, OP_READ),
       op_arg_dat(p_adt,-1,OP_ID,OP_WRITE));

      op_par_loop(res_calc,edges,
        op_arg_dat(p_x,     0,pedge, OP_READ),
        op_arg_dat(p_x,     1,pedge, OP_READ),
        op_arg_dat(p_q,     0,pecell,OP_READ),
        op_arg_dat(p_q,     1,pecell,OP_READ),
        op_arg_dat(p_adt,   0,pecell,OP_READ),
        op_arg_dat(p_adt,   1,pecell,OP_READ),
        op_arg_dat(p_res,   0,pecell, OP_INC),
        op_arg_dat(p_res,   1,pecell, OP_INC));

     op_par_loop(bres_calc,bedges,
       op_arg_dat(p_x,      0,pbedge, OP_READ),
       op_arg_dat(p_x,      1,pbedge, OP_READ),
       op_arg_dat(p_q,      0,pbecell,OP_READ),
       op_arg_dat(p_adt,    0,pbecell,OP_READ),
       op_arg_dat(p_res,    0,pbecell, OP_INC),
       op_arg_dat(p_bound,-1,OP_ID,   OP_READ));

     rms = 0.0;

     op_par_loop(update,cells,
       op_arg_dat(p_qold,-1,OP_ID, OP_READ),
       op_arg_dat(p_q,    -1,OP_ID,OP_WRITE),
       op_arg_dat(p_res, -1,OP_ID,    OP_RW),
       op_arg_dat(p_adt, -1,OP_ID, OP_READ),
       op_arg_gbl(&rms, OP_INC));} \\inner loop
  } \\outer loop
  return 0;
}
```

Figure 1: Airfoil application implemented in OP2.

allowing the OP2 compiler to precisely characterise the dependencies between OP2-loops without complex data-flow analyses.

The second OP2-loop iterates over cells and it includes as first parameter an indirectly accessed `op_dat`. The related argument line indicates that: the `op_dat p_x` is accessed; as `p_x` is associated with the node set, the mapping (*pcell*) is used to translates a cell identifier (i.e. an iteration index) to one of the 4 corresponding node identifiers. The second argument of the `op_arg` function is used to specify which one of the four nodes connected to the current cell (on which the user kernel is applied at each loop iteration) is to be consider as actual parameter. In this loop we pass all four nodes information related to each cell to the kernel. As this `adt_calc` loop includes one or more indirectly accessed `op_dat` arguments, it is then classified as *indirect*.

Finally, consider the `res_calc` OP2-loop, which iterates over edges, and accesses the `p_res` dataset through an indirection (from edges to cells) by incrementing it (OP_INC). This means that two iterations of the same loop can in principle increment the same data at the same time (i.e. in parallel, as required by OP2-loop semantics). To solve this parallelism control issue, OP2 takes different strategies, each optimised for a different target architecture. For the sake of loop fusion discussed in this paper, we only consider the case of GPUs and multicores. For these architectures a colouring technique is used, where different colours are assigned to possible conflicting iterations, and then execution parallelises iterations with the same colour, but it sequentialises those with different colours. This — OP_INC over indirect argument — is the only way in which the user can rely on OP2 to avoid potential race conditions. If the user declares a OP_WRITE access to an indirect argument, then it is the user's task to guarantee that no two iterations can modify the same data, through a proper selection of the mapping data.

3.1 OP2 Assumptions

OP2 makes several important assumptions:

- For any indirect OP2-loop, there is only one level of indirection in retrieving the data.
- The order in which elements of a set are processed *in an OP2-loop* does not affect the final result, thereby providing the compiler and run-time support great flexibility in ordering the computations.
- How the data are accessed in OP2-loops is correctly given by the programmer.
- The data associated with sets are not written outside of the parallel iteration space, that is, in the sequential parts of code. In essence, once the data pass into OP2 territory, further modifications to the data can only occur opaquely through the OP2 API. As we observe in Section 5, this property facilitates aggressive program slicing and whole-program analysis.

3.2 OP2 Compiler Infrastructure

Producing a binary for a specific hardware from an OP2 program requires several compilation steps. The first of these utilises source-to-source transformation to transpose the OP2-loops into a parallel implementation compliant with the architecture of choice, e.g. CUDA. The second step then simply compiles the generated code using a vendor-specific compiler.

In order to realise the source-to-source translation, our OP2 compiler, illustrated in Figure 2, performs the following steps:

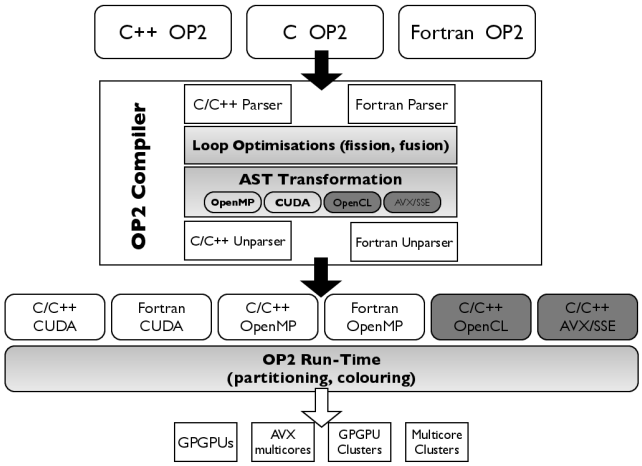

Figure 2: Overview of OP2 Compiler Framework.

- Parses the source code, using the parsers provided by ROSE [1], to obtain the Abstract Syntax Tree (AST) that forms the basis of all subsequent analyses. Currently we support Fortran 77-2003 and C/C++.
- Optimisations of OP2-loops using the AST and the techniques outlined in Section 5.
- Re-writing of the AST to generate functions implementing the parallel computation and replacing calls to OP2-loops with calls to the generated functions. For instance, if CUDA is the target programming model, this step involves generation of the host stub and the CUDA kernel functions. Each OP2-loop call is subsequently replaced by a call to the appropriate host stub.
- Re-writing of the kernel functions supplied by the user. In the case of CUDA, for example, this is needed to label the kernel as a device subroutine.
- Unparsing of the AST to output the generated code using the ROSE unparsers.

Our compiler currently supports generation of CUDA and OpenMP for both Fortran and C/C++, while OpenCL and SSE/AVX backends are in development for C/C++.

Note that the final step in Figure 2 includes linking the generated code against an OP2 library; this provides runtime support to partition the iteration sets (to parallelise) and to colour partitions (to avoid race conditions). Both Fortran and C/C++ generated programs utilise the same implementation of the colouring and partitioning algorithms, which are written in C. As shown in [3], this incurs a small ($\sim$5%)) performance cost for Fortran generated programs due to interoperating with C.

4. PROGRAM MODEL AND SLICING

The previous section gave an overview of OP2 from the programmatic point of view. Here we formalise the model extracted from the analysis stage of our compiler, which serves as the basis of the optimisation process.

An OP2 program consists of several subprograms each of which is characterised by its CFG:

DEFINITION 1. *The CFG $C = \langle V_C, E_C, \mathtt{s}, \mathtt{t}\rangle$ of a subprogram is a directed graph such that:*

- *V_C are vertices representing basic blocks, which are maximal sequences of statements, such that there is only one entry point and a single exit point.*
- *$\mathtt{s}, \mathtt{t} \in V_C$ are distinguished (dummy) vertices, respectively, such that* `s` *has no predecessors (the entry vertex) and* `t` *has no successors (the exit vertex).*
- *E_C models the branches and fall-throughs between basic blocks. Or, every entry point to the function has* `s` *as a predecessor and every exit point of the function has* `t` *as a successor.*

This paper assumes that the CFGs are constructed from the AST after parsing of the source code during the compilation process. For this reason, OP2 parallel loop calls merely appear as unique basic blocks in the CFG.

The CFG of the OP2 program appearing in Figure 3 is provided to the left of Figure 3. The outer loop with a bound of 1000 is represented by vertex L_1 and exits to vertex E_1, its outer scope. Similarly, the inner loop with bound 2 is represented by vertex L_2 and exits to vertex E_2.

The optimisations proposed in this paper assume that *natural loops* in a CFG can be identified, which are loops with a single entry point, called the header. Transitions from within a loop to the header are called *loop-back edges*. The nesting relationship between loops can then be represented hierarchically through a Loop-Nesting Tree (LNT):

DEFINITION 2. *For a CFG $C = \langle V_C, E_C \cup \{\mathtt{t} \to \mathtt{s}\}, \mathtt{s}, \mathtt{t}\rangle$, its LNT $T_L^C = \langle V_{T_L^C} = V_C, \mathcal{H}, E_{T_L^C}, r\rangle$ is a tree with the following properties[2]:*

- *$\mathcal{H} \subseteq V_C$ is the set of internal vertices representing the headers identified in C.*
- *$V_C \setminus \mathcal{H}$ is the set of leaves.*
- *$E_{T_L^C} = \{(header(v), v) | header(v) \in \mathcal{H}, v \in V_C \setminus \mathcal{H}\}$ where $header(v)$ is the representative header of the innermost enclosing loop in which v is contained.*

For example, in the CFG of Figure 3, there are two loop-back edges, `update` $\to L_2$ and $E_2 \to L_1$. The LNT is depicted to the right in Figure 3 where internal vertices L_1 and L_2 represent the respective loops. This figure also shows that: L_2 is nested in L_1; the innermost enclosing loop of `save_soln` is L_1; the innermost enclosing loop of `adt_calc`, `res_calc`, `bres_calc`, `update` is L_2.

The final program model that our optimisation framework needs is the call graph, which represents inter-procedural relations between subprograms is needed:

DEFINITION 3. *A call graph $\mathcal{C} = \langle V_\mathcal{C}, E_\mathcal{C}, \mathcal{B}, \omega\rangle$ of a program is a digraph in which:*

- *$V_\mathcal{C}$ is the set of vertices representing subprograms.*

[2]We explicitly add the edge $\mathtt{t} \to \mathtt{s}$ to the CFG C to ensure that it becomes strongly connected. Therefore, all vertices in C are enclosed in a loop and the nesting relationship between loops can be captured in a tree as opposed to a disjoint union of trees, i.e. a forest.

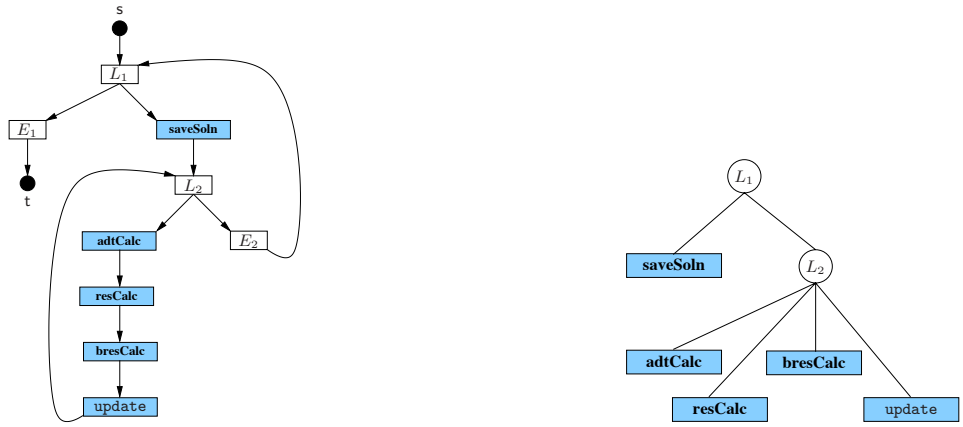

Figure 3: Example CFG and LNT.

- *$E_\mathcal{C}$ is the set of edges representing the calling relation between subprograms. In particular, for any $(u, v) \in E_\mathcal{C}$, subprogram u calls subprogram v.*
- *$\mathcal{B}$ is the set of basic blocks in the program that transfer control flow to a different subprogram, i.e. the call sites.*
- *$\omega : E_\mathcal{C} \mapsto (2^\mathcal{B} \setminus \emptyset)$ is a function mapping a call to the call sites leading to that call.*

In this paper, we assume that all indirect function call destinations have been resolved and that there is no recursion so that the call graph is acyclic.

4.1 Slicing to Enable Whole-Program Analysis

Whole-program analysis is a technique that provides visibility to the *entire* CFG of the program. It is particularly desirable because it opens up optimisation opportunities not available to the typical module-wide analyses of modern compilers. However, the size of a whole-program CFG grows exponentially due to the need to duplicate CFG from callees into callers at every call site; that is, for a callee CFG $C = \langle V_C, E_C, \mathtt{s}, \mathtt{t} \rangle$, each inline increases the size of the caller CFG C' by $(|V_C| + |E_C|)^n$, where n is the number of of calls from C' to C.

In practice, the number of vertices and edges inlined can be eased considerably by *slicing* away portions of CFGs which do not apply to the analysis. This is particularly relevant to OP2 as it presumes that the data it effectively owns are not modified in the sequential parts of the code (see Section 3). For this reason, aggressive program slicing can be applied and whole-program analysis enabled. In HYDRA, for instance, there are a mere 700 OP2-loops, vastly reducing the complexity of the whole-program CFG.

The slice of interest in our case is the OP2-loops and the decisions on which their execution is dependent. The latter parts are needed to maintain the basic shape of control flow, such as loops in which OP2-loops are contained or the conditional controlling entry into a specific OP2-loop. This is needed in the subsequent optimisation phase to reason about which loops can safely be fused or split, without changing the semantics of the code.

BUILD-OP2-CFG($\mathcal{C}, C_1, \ldots, C_{|V_\mathcal{C}|}$)

```
1  foreach v ∈ V_C in reverse post-order do
2    Compute the control-dependence graph of CFG C_v
3    Slice C_v with respect to the OP2-loops
4    foreach s ∈ succ(v) do
5      foreach c ∈ ω(v, s) do
6        Inline C_s into C_v at c
```

Figure 4: Algorithm to construct the whole-program CFG.

Figure 4 gives the algorithm to produce the whole-program CFG given the call graph $\mathcal{C}$ and the CFGs $C_1, \ldots C_{|V_\mathcal{C}|}$) of the program. It moves up the call graph in a bottom-up fashion (Lines 1- 6). For each vertex in the call graph v, we compute the control-dependence relation of its CFG C_v (Line 2), for which there are known algorithms [4]. This allows us to slice the program accordingly with respect to the OP2-loops (Line 3). Every callee s of v is then analysed (Line 4), and at each call site c leading to a call of s (Line 5), we inline the CFG C_s (Line 6).

4.2 Loop unrolling

The biggest hurdle in fusing loops is discovering opportunities in the code, as a user typically writes OP2-loops in an optimised way; that is, a programmer usually spots when two sequentially-composed OP2-loops can be merged as they iterate over the same set. For instance, the code in Listing 3 does not contain any obvious fusion opportunities.

OP2-loop fusion is enabled inside our compiler by unrolling specific loops. In particular, we ignore loops with early exits, as these complicate the unrolling process due to code duplication, and those for which the bound cannot be determined at compile time, since it is impossible to determine the unroll factor. Note that there are known techniques to determine the upper bounds of loops [2, 12, 8]. Armed with this information, the LNT of the whole-program CFG is then traversed in a bottom-up fashion and, as each loop is analysed, we attempt to fully unroll the body of the loop.

We then search in the body of the unrolled loop for maximal sequences of basic blocks that contain OP2-loops over the same set, and fuse depending on the chosen backend ar-

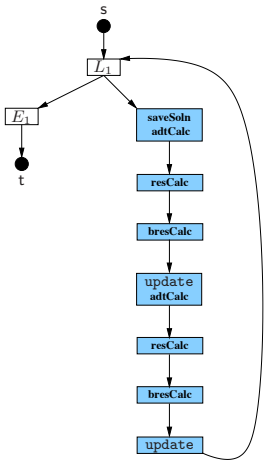

Figure 5: The CFG of Figure 3 after Unrolling and Fusion Transformations.

chitecture. For example, it is not always beneficial to fuse in the case of CUDA, because each thread can only assume the availability of a limited number of registers, otherwise spillage into the L1 cache occurs. Observe that, when complete unrolling becomes too costly due to a large loop bound, we instead limit the unroll factor to a small positive number.

Let us re-consider the (whole-program) CFG and its LNT in Figure 3. Using our technique, loop L_2 is the first encountered during the bottom-up traversal of the LNT; its body is then completely unrolled because its bound is two. This immediately exposes two obvious OP2-loop fusion opportunities which are not feasible without unrolling: `save_soln` and `adt_calc` because they both iterate over the set `cells` (see Listing 3); similarly, `update` and `adt_calc`. The CFG with L_2 unrolled and the OP2-loops fused is presented in Figure 5. Note that completely unrolling loop L_1 is too costly as its upper bound is one thousand; however, unrolling it once would expose another opportunity since, in the CFG of Figure 5, `update` is the last OP2-loop executed, while the fused OP2-loop `save_soln/adt_calc` is the first OP2-loop executed, and both iterate over `cells`.

5. OP2-LOOP FUSION

The program analysis presented in the previous section enables us to spot loop optimisation opportunities, specifically related to OP2-loops fusion. In this section we discuss the benefits deriving from applying loop fusion in the case of the GPU compiler back-end, and the restrictions that we have so far discovered to its application in our programming model. For brevity, we omit a full discussion related to the OpenMP back-end of our compiler. However, the same implementation scheme, based on a so-called *scratchpad* memory (see below), is used also on multicores, to minimise cache misses. Loop fusion is thus valid also in this case.

5.1 Motivation

The CUDA implementation of an OP2-loop proceeds in the following manner. First, data declared by the user to be owned by OP2 (through `op_decl_dat`) is transferred at elaboration time into global device memory through the OP2 run-time support. Second, when control reaches an OP2-loop in the program (generated by our OP2 compiler), a *host stub* is called. In the host stub, the grid dimensions, the thread-block dimensions, and the size of dynamically allocated shared memory are set for the CUDA kernel launch. Third, the CUDA kernel is called, which itself consists of three principal steps. In this implementation, the mesh is partitioned and an instance of CUDA kernel is applied to each partition. The kernel steps are:

1. Data needed by the user-supplied kernel is staged in from global device memory into shared memory. In doing so, the compiler coalesces in shared memory the data scattered in device memory, this last being a typical feature of unstructured mesh applications. This means that the shared memory implements a scratchpad memory for the partition execution.
2. The user-supplied kernel is called with the data resident in shared memory.
3. Data is staged out back to global device memory. Evidently, this allows subsequent calls to the OP2-loop to observe changes to the data.

Staging data in and out of the global device memory is a costly activity. In the general case these costs cannot be avoided, since there can be arbitrary control flow between different invocations of an OP2-loop. However, when there are at least two sequentially-composed OP2-loops iterating over the same set which share some data, the overhead can be reduced through loop fusion, potentially leading to significant performance benefits. In addition, the current OP2 CUDA and OpenMP implementation requires a full thread synchronisation between loops (i.e. OP2-loops). Such a synchronisation can be avoided when loop fusion is applied.

Fusing OP2-loops is generally more straightforward than fusing (ordinary) loops since the dependencies between data are explicitly represented in the OP2 programming layer by means of access descriptors. The fusion element, therefore, equates to concatenating several user-supplied kernel function bodies and unioning their formal parameters to create a single, monolithic function. Later we show an example to clarify the fusion process.

5.2 Feasibility of Loop Fusion in OP2

Loop fusion is a well-known technique always applied by optimising compilers to regular applications. However, for irregular applications, using indirect data accesses instantiated at run-time as it is the case of unstructured meshes, this technique is difficult to apply automatically. In this paper we consider *syntactic* loop fusion, where the compiler is able to transform the input program fusing loops. This is possible only in some simple albeit effective cases, as we show in the experiments section, thanks to the use of OP2 access descriptors. In fact, these permit us to analyse straightforwardly dependencies between successive OP2-loops, and to automatically decide if loop fusion is feasible and possibly efficient.

Let's consider that our whole program analysis has spotted that two successive loops can be potentially fused. A first analysis for loop fusion feasibility considers the iteration sets of each loop:

- If the two loops iterate over different sets, we do not apply loop fusion. This is mainly due to the impossibility of easily relating iterations defined over two different spaces.

- If the two loops iterate over the same sets, then loop fusion might be applied, depending on the analysis of the access descriptors.

We further analyse the second case, when the two OP2-loops iterate over the same set. A further loop fusion feasibility classification can be done by considering if two OP2-loops use indirect datasets accesses (i.e. `op_maps`):

- If the two OP2-loops are direct (no indirect accesses) then loop fusion is feasible. As there are no indirect accesses, an iteration i of the second loop can only depends on the values produced by the same iteration i of the previous loop. The resulting fused loop applies, for each iteration, successively the first and second loops' kernels.

- Consider the case in which the two OP2-loops are indirect. If these have a read-after-write dependency (the first loop using OP_INC) on a dataset indirectly accesses, then it is possible than iteration i from the second loop can depend on two iterations (e.g. j and k) of the first loop. As a consequence, no simple scheduling without the knowledge of the precise iteration set sizes and mapping information can be applied at compile-time, hence no loop fusion can be defined at compile-time. If no data sharing is present, then there is no actual performance gain (see below) derives from loop fusion, even if it is feasible.

- Finally, consider the cases when a direct OP2-loop is followed by an indirect loop, or viceversa. The condition of the previous case cannot hold, as the direct loops cannot access datasets indirectly, and as a consequence a one-to-one dependency between loop iterations is defined. Notice that data sharing can only happen through directly accessed datasets. Loop fusion is hence feasible for these cases, and the resulting single fused loop will have indirect datasets accesses (i.e. it is an indirect loop).

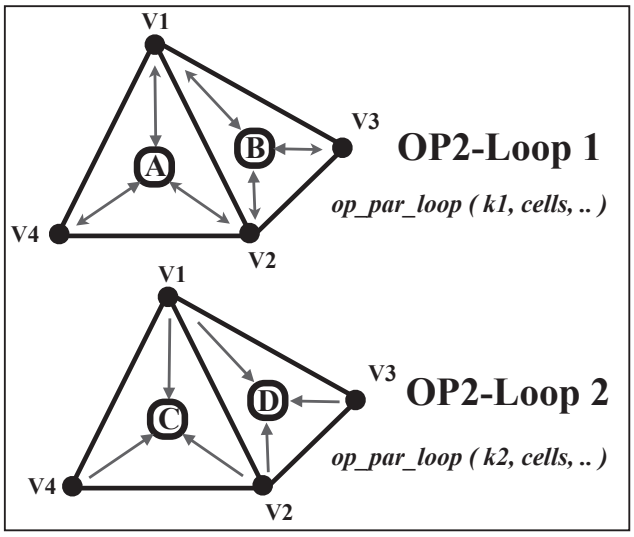

Figure 6: Example of infeasibility of loop fusion, when two indirect loops, iterating over cells, access the same dataset defined over nodes using some mapping.

Figure 6 show a visual representation of the second case. Two successive OP2-loops iterate over cells, the first incrementing data associated to nodes, and the second reading it. In the figure, we show the details of two iterations (i.e. two cells) for each loop. The shared data on nodes is related to nodes **V1** and **V2**. As iteration **A** and **B** modify these nodes, then both iterations **C** and **D** depends on them, i.e. on the results that they produce. Therefore, no loop fusion of iterations can be performed without the knowledge of which nodes connect which cells (i.e. the mesh).

The first case (two direct loops) always delivers a performance improvement if there is data sharing between the loops and the scratchpad memory is efficiently used. The efficiency of the third case depends on the trade-off between the improved data sharing between the loops, and the reduced parallelism degree achievable for the direct loop involved, as the fused loop will be an indirect one, inevitably using parallelism control techniques (e.g. colouring in the considered architectures).

By applying this analysis to the Airfoil application with the inner loop unrolled (see Section 4), we are able to apply loop fusion to the following OP2-loop pairs: `save_soln` and `adt_calc`; `update` and `adt_calc`. These pairs both iterate over cells, and they include a direct and indirect loop (third case described above).

When loop fusion cannot be defined at compile-time, in a syntactic manner, then run-time loop fusion can be pursued, by properly synthesising fusing code in the compiler. A support for run-time loop fusion can be based on the building of a task graph of the iterations of the two loops, where dependencies between iterations are derived from OP2 maps. This kind of technique can be expressed by using similar frameworks enabling the application of other techniques, like those underlying sparse tiling, as described in [17]. In future work we will introduce this modelling framework in our compiler to achieve run-time loop fusion.

6. EVALUATION

To evaluate the effectiveness of loop fusion we applied it "manually" to the unrolled version of the Airfoil program. We generated three versions of the Airfoil:

- The original version, denoted as *original*, as presented in the previous sections.

- A version with a single loop fusion of the `save_soln` and `adt_calc` loops, denoted with *single fusion*. This version is achieved by un-rolling the inner loop, as showed in Section 4.

- A version extending the previous one with a further fusion of the `update` and `adt_calc`, again feasible due to the loop unrolling of the inner loop. This version is called *double fusion*.

We then translated the related Airfoil program into CUDA and OpenMP using our OP2 compiler, and executed them respectively on an Nvidia M2050 GPU, and on a multicore node supporting two Intel Xeon X5650 "Nehalem" processors, each including 6 2-way hyperthreaded cores. Configuration parameters for the CUDA architecture are the size of the unstructured mesh partition, which has a direct impact on the size of shared memory/cache needed for the execution of each partition, and the number of threads in a block

(called *block size*), executing in parallel the iterations in a same partition (see [3] for implementation details). For the multicore architecture we have again the partition size as a parameter, and the number of threads used in the execution of the OP2-loops, which we instantiate to 4, 8, 12 and 16. Notice that the last value is larger than the actual maximum parallelism degree supported by the multicore. This last cases does not map each thread to a physical core, but the hyperthreading support is used.

The following experiments are related to an Airfoil program using double precision floating point values as datasets, applied to an unstructured mesh including approximately 1.5 million edges, and 700 hundreds thousands nodes and cells. The actual working version of the Airfoil used in these experiments is written in Fortran. It is compiled by our OP2 compiler to the Nvidia GPU, and then compiled again to its executable form using both the PGI CUDA Fortran compiler and the Nvidia CUDA compiler (nvcc). Options passed to this last compilation step are "-fast -O2". For the multicore implementation, we used the Intel Fortran and C compiler, using the following options: "-O3 -parallel"

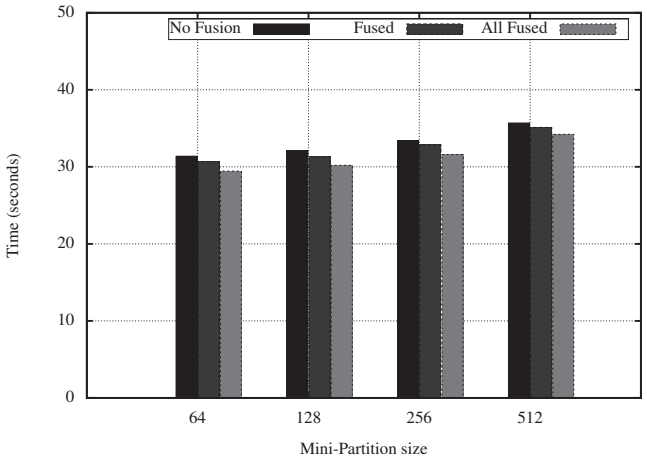

Figure 7: Results of three Airfoil versions in comparison. For each group, the original version has the leftmost darker line; the single fusion one the medium dark middle line; the double fusion version the rightmost lighter line. On the X-axis the partition size is displayed, and on the Y-axis the total execution time.

Figure 7 shows the results of the execution of respectively the *original* (darker line), *single fusion* (medium dark line), and *double fusion* (lighter line) versions on the GPU. The figures have on the X-axis the partition size, which in these experiments is equal to the CUDA thread block size. In the Y-axis the execution times in seconds is shown, while different bands for the same X-axis value denote the three different versions of the airfoil. Results indicate that the incremental loop fusion application delivers increasing performance, i.e. smaller execution times, for the three different configurations. In particular, the best configuration (i.e. partition size equal to 64) results in a 6.124% improvement for the double fusion version and 2.19% for the single fusion one.

This performance gain is small compared to the total execution time. This is mainly due to the fact that a loop that is not subject to optimisation, i.e. the `res_calc` loop, is the most expensive one in this program. Loop optimisations applied to other more lightweight kernels affect only slightly the total performance. However, this result also means that loop fusion delivers a performance improvement even on lightweight OP2-loops.

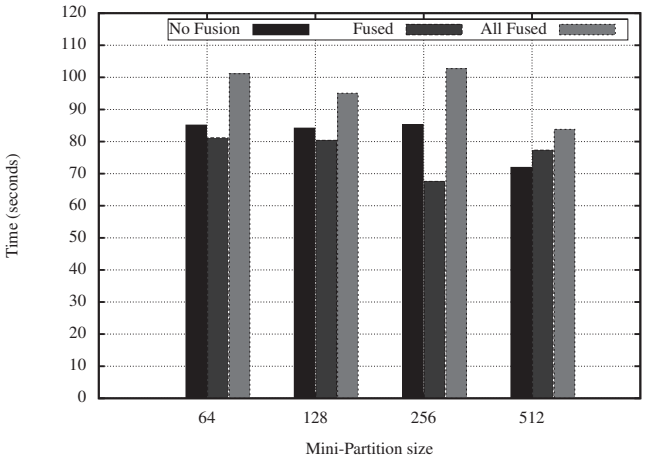

Figure 8: Results of three Airfoil versions in comparison for 12 OpenMP threads. For each group, the original version has the leftmost darker line; the single fusion one the medium dark middle line; the double fusion version the rightmost lighter line. On the X-axis the partition size is displayed, and on the Y-axis the total execution time

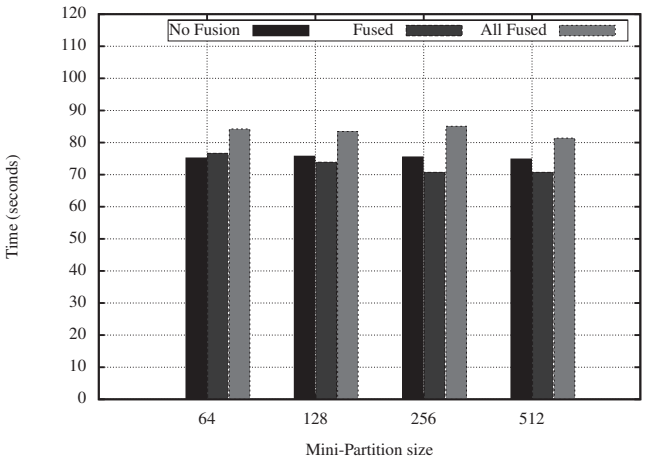

Figure 9: Results of three Airfoil versions in comparison for 16 OpenMP threads. For each group, the original version has the leftmost darker line; the single fusion one the medium dark middle line; the double fusion version the rightmost lighter line. On the X-axis the partition size is displayed, and on the Y-axis the total execution time

Figure 8 and 9 show the results of the experiments on the multicore processor for 12 and 16 OpenMP threads. Again, the X-axis models increasing partition sizes and the Y-axis the execution time in seconds. In this paper we only report results for parallelism degrees giving the best performance. The different columns for the same X-axis value are related to one of the three versions of the Airfoil application, as for CUDA experiments.

In general, it can be noticed that a performance improvement is obtained by the single fusion version, with a peak performance improvement of 20.76%. However, the double fusion version is showing generally worse performance. This is mainly given by a cache pollution effect: as in this case the shared op_dat between the two second fused loops is not staged into cache, then fusion benefits are only based on the ability of increasing data locality, against the cache pollution effects.

7. CONCLUSION

OP2 is a programming library which enables parallel iteration over elements of unstructured meshes to be expressed without being tied to a particular implementation. This paper proposes using whole-program analysis to optimise these applications, which is made feasible by aggressively slicing the sequential parts of the code. The whole-program control flow graph then enables loops in the OP2 layer to be fused or split according to the best choice for a particular architecture.

Since a programmer typically writes OP2-loops in an optimal way w.r.t. sequential composition, we showed how loop unrolling can increase the opportunities for fusion. To assess the benefits and practicality of these optimisations, we analysed the Airfoil application, a representative program heavily used in the computational fluid dynamics domain. Our results showed a small performance improvement for GPUs, and a greater one for multicore, due to a better use of respectively shared and cache memory. This suggests us that loop fusion represents generally a performance improvement. Extensive studies of more complex forms of run-time loop fusion to cover the unfeasible cases showed in this paper will deliver additional and higher optimisations to unstructured mesh applications.

8. REFERENCES

[1] The ROSE compiler. http://wwww.rosecompiler.org/.

[2] M. Bartlett, I. Bate, and D. Kazakov. Guaranteed loop bound identification from program traces for wcet. In *Proceedings of the* 15^{th} *Real-Time Technology and Applications Symposium (RTAS'09)*, April 2009.

[3] C. Bertolli, A. Betts, G. Mudalige, M. B. Giles, and P. H.J. Kelly. Design and performance of the OP2 library for unstructured mesh applications. In *Euro-Par 2001 Parallel Processing Workshops*, LNCS. Springer, 2011.

[4] G. Bilardi and K. Pingali. A framework for generalized control dependence. *SIGPLAN Not.*, 31, May 1996.

[5] D.A. Burgess, P.I. Crumpton, and M.B. Giles. A parallel framework for unstructured grid solvers. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 97–106, 1994.

[6] P.I. Crumpton and M.B. Giles. *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers*, chapter Multigrid aircraft computations using the OPlus parallel library, pages 339–346. 1996.

[7] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[8] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proceedings of the* 7^{th} *Int'l. Workshop on Worst Case Execution Time (WCET) Analysis*, July 2007.

[9] M. B. Giles, M. C. Duta, J. D. Muller, and N. A. Pierce. Algorithm developments for discrete adjoint methods. *AIAA Journal*, 42(2):198–205, 2003.

[10] M.B. Giles, G.R. Mudalige, Z. Sharif, G. Markall, and P. H.J. Kelly. Performance analysis and optimisation of the OP2 framework on many-core architectures. *The Computer Journal*, 2011.

[11] M.B. Giles, G.R. Mudalige, Z. Sharif, G. Markall, and P. H.J. Kelly. Performance analysis of the OP2 framework on many-core architectures. *SIGMETRICS Perform. Eval. Rev.*, 38(4):9–15, March 2011.

[12] C. A. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loops iterations. *Real-Time Systems*, 18(2-3):129–156, May 2000.

[13] Lee W. Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H.J. Kelly. Deriving efficient data movement from decoupled access/execute specifications. In *Proceedings of the 4th International. Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, 2009.

[14] J. S. Meredith, R. Sisneros, D. Pugmire, and S. Ahern. A distributed data-parallel framework for analysis and visualization algorithm development. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 11–19, New York, NY, USA, 2012. ACM.

[15] P. Moinier, J. D. Muller, and M. B. Giles. Edge-based multigrid and preconditioning for hybrid grids. *AIAA Journal*, 40(10):1954–1960, 2002.

[16] http://www.oerc.ox.ac.uk/research/op2.

[17] M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[18] Mark Weiser. Program slicing. In *Proceedings of the 5th Int'l. conference on Software engineering*, ICSE '81, 1981.