Iterative Collective Loop Fusion

T.J. Ashby[1] and M.F.P. O'Boyle[1]

Institute for Computer Systems Architecture,
University of Edinburgh, Scotland, UK
T.Ashby@ed.ac.uk, mob@inf.ed.ac.uk
http://www.icsa.inf.ed.ac.uk/compilers/

Abstract. Naive code generation from high-level languages that encourage modularity can give rise to large numbers of simple loops for array-based programs. Collective loop fusion and array contraction can be used on such codes to improve temporal locality and performance. The problem is typically formalised using a *loop dependence graph* (LDG), with solutions denoted by *fusion partitions*. Much previous work has concentrated on approaches to the abstract formulation. We present our technique called *iterative collective loop fusion* based on empirically evaluating different transformations, and show how it can provide speedups over existing approaches of up to 1.38. We also give results showing that applying such techniques to high-level languages can provide speedups of up to 2.45 over the original code, and outperforms an equivalent code in Fortran.

1 Introduction

Advanced programming languages that encourage modularity can give rise to programs with many loops, poor temporal locality and many temporary array variables when used to write scientific codes. The original motivation for this work came from the desire to optimise a package of iterative linear solvers with exactly these characteristics, written in Aldor[1], a high level mixed functional/imperative programming language for numerical and symbolic computer algebra. Such codes can benefit greatly from a systematic approach to loop fusion and array contraction. However, previous approaches to collective loop fusion have chosen transformations based on models rather than real performance. Our proposed solution to the problem is *iterative collective loop fusion*, which brings the techniques of iterative compilation to collective loop fusion.

The rest of the paper is organised as follows: Section 2 introduces the basic formalism with examples, Section 3 discusses previous work and motivates the development of our technique, Section 4 describes the technique in detail, Section 5 provides experimental results and Section 6 concludes and offers some ideas for future development.

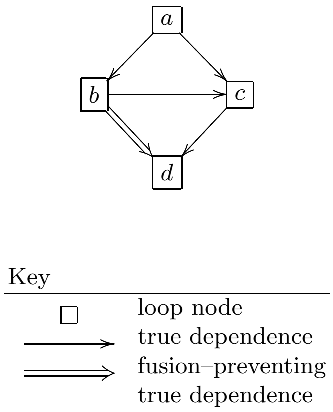
2 Formulation and Example

A *loop dependence graph* (LDG) describes a *program section* that consists of basic blocks and perfectly nested loops with no additional branching for which data dependencies are known. Figure 1 gives an example with pseudocode for four loops and the corresponding loop dependence graph. Nodes in the graph represent the loops of the program section, and a directed edge exists between two nodes if the target is data dependent on the source. The lack of branching in the program section ensures that its LDG is acyclic.

The LDG is used to reason about loop fusion for the program section that it represents. A *dependency path* (or just path) in the LDG is a set of edges describing a path from a source node to a destination node through the graph following the directed edges. Two loops are *conformable* if their headers are the same. The nodes representing two conformable loops are possible candidates to be directly fused if they connected by paths of length one and all the distance vectors from the source to the target are non-negative, such as loops *a* and *b* in Figure 1, or if they are not connected by a path. In the former case such an edge is defined as *collapsible*. A dependency path is collapsible if all its edges are collapsible, and non-collapsible otherwise.

```
(loop a)        for i in 1..n do
                    a[i] := ...
                done

(loop b)        for i in 1..n do
                    reduction := reduction + a[i]
                    b[i] := ...
                done

(basic block)   alpha := reduction

(loop c)        for i in 1..n do
                    c[i] := a[i] + b[i]
                done

(loop d)        for i in 1..n do
                    d[i] := alpha * c[i]
                done
```

i)



ii)

Fig. 1. An example LDG. i) Pseudocode for the original program section, with four loops and one basic block. Only array *d* is live out of the program section (i.e. read at some later point), so all the other arrays can potentially be completely contracted. The loops are all conformable, and all distance vectors are 0, except for the loop-carried dependence in the second loop for a reduction variable, the dependence of the basic block on said reduction variable, and the dependence of the fourth loop on the basic block. ii) The corresponding loop dependence graph. Nodes in the graph are labelled with the name of the array that they write to.

2.1 Collective Loop Fusion and Fusion Partitions

A fusion partition is a partitioning of the nodes of an LDG into disjoint sets (*partitions* or *clusters*) where the nodes in each set will be fused together to produce the final transformed code. A fusion partition itself can be represented by a graph where nodes are clusters, and there is an edge between cluster nodes for every edge that exists between the loop nodes that belong to the respective partitions in the LDG. See Figure 2 for two fusion partitions of the LDG in Figure 1. The *size* of a fusion partition is the number of non-empty partitions it has (empty partitions are not allowed). For a fusion partition to be legal, it must be possible to fuse together all the nodes within a given partition, and the graph of the fusion partition must be acyclic. The first condition is satisfied by the absence of non-collapsible edges within the cluster. A given LDG has a lower bound on the size of its legal fusion partitions determined by the dependency path with the most fusion–preventing edges in it – for example, the minimum size fusion partition for the LDG in Figure 1 is two.

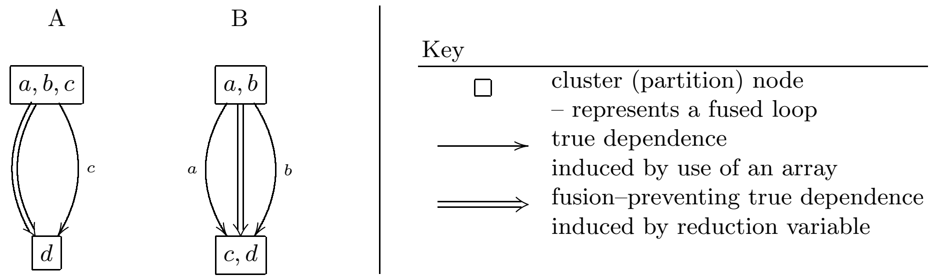

A
B
a, b, c
d
c
a, b
c, d
a
b
Key
cluster (partition) node
– represents a fused loop
true dependence
induced by use of an array
fusion–preventing true dependence
induced by reduction variable

Fig. 2. The graphs of two possible fusion partitions of the LDG from Figure 1. Nodes in the graph (clusters) are labelled with the letters representing the loop nodes within that cluster. Both fusion partitions are the same size (2), but permit different amounts of array contraction – partition A allows two arrays to be contracted (a and b), whereas B allows only one (c). This corresponds (inversely) to the inter-cluster array dependency edges in the graphs of the fusion partitions, which are labelled with the non-contracted array they correspond to – one for partition A and two for B.

2.2 Array Contraction

For a given array, (complete) contraction will be legal after partitioning if all the dependencies associated with it appear in the same cluster, and they all have distance zero. Applying array contraction to two fusion partitions of the same size on a given LDG can give different contraction amounts. This can be seen by the number of edges in Figure 2, and also the replacement of arrays with scalar variables in Figure 3, which represents the end product of the transformations represented in Figure 2. Conversely, different size partitions with the same amount of contraction are also possible.

A fusion partition can be labelled with a pair of numbers that denote the size of the fusion partition and the amount of array contraction that it permits. For

```
for i in 1..n do
    a := ...
    reduction := reduction + a
    b := ...
    c[i] := a + b
done

alpha := reduction

for i in 1..n do
    d[i] := alpha * c[i]
done
```

A

```
for i in 1..n do
    a[i] := ...
    reduction := reduction + a
    b[i] := ...
done

alpha := reduction

for i in 1..n do
    c := a[i] + b[i]
    d[i] := alpha * c
done
```

B

Fig. 3. Pseudocode representing the two fusion partitions A and B from Figure 2. Array contraction has been applied e.g. arrays *a* and *b* in the first loop of partition A have been reduced to scalar variables

some LDGs there will be multiple fusion partitions with the same (*contraction amount*, *partition size*) label.

3 Previous Work

3.1 Standard Model Based Approach

It is usual to associate a *cost function* with an LDG that ranks the possible transformations that can be applied to it. The simplest example of this is preferring more fusion over less (e.g. [2] in the context of typed loop fusion), with all fusion partitions of the same size being equal. A more sophisticated (and more common) approach is to add to the LDG a set of edges and associated weights that model the expected benefit of fusing the loops that they connect.

There have been numerous minor variations on the second approach. Some examples include transformations specifically for array contraction [3], [4], and a technique which minimises memory usage and simultaneously improves locality whilst limiting the size of any fused loop that is produced (i.e. avoiding "over fusing") [5]. One adaptation replaced edges in the cost graph with hyper edges to better capture re-use between array operands being read [6]. There have also been several composite approaches, such as a technique that prevents the creation of loops with parallelisation–preventing loop–carried dependencies [2], and a related approach that uses adjustable weights which can be altered to favour fusion for parallelism or fusion for locality [7].

The abstract formulation of various problems has been shown to be at least NP-hard [6], [8]. Consequently, most work on loop fusion is based on heuristic algorithms to find some approximation to the optimum answer for the model.

Approaches have included various greedy algorithms [4], [9], and algorithms based on max-flow min-cut heuristics [3], [6], [5].

There are two major weaknesses in previous model based fusion/contraction work. The first is the use in some approaches of overly simple search strategies to find some approximation to the solution of the idealised NP-hard problem (e.g. greedy search). As pointed out in [10], the majority of LDGs encountered in realistic programs will be small, and hence there is no real reason to emphasise the efficiency of the search so much at the cost of the quality of the approximation. The second problem is that although all the approaches discussed above target slightly different optimisations, it can be assumed that their ultimate goal is to get the best performance for a given LDG, but no authors have adequately explored the differences between their idealised problem and the implementation details of actual hardware.

For example, for a given LDG there may be many fusion partitions all ranked equal according to some abstract cost function (e.g. all with the same amount of contraction). However, for any method in the literature there is not usually any indication of how any particular one is chosen, or any indication of how the actual quality of the equally ranked LDGs varies in practice. Another illustration is the lack of any indication as to how fusion for locality and fusion for contraction may conflict, how the trade-off should be managed to get the best performance, and crucially how this may vary depending on the form of the loops and the actual processor architecture under consideration.

3.2 Iterative Optimisation

Current implementations of computer architectures contain a wide variety of complex structures, and consequently they are very difficult to model accurately – for one example of this see [11]. To combat this, the approach of iterative optimisation treats the goal of finding good transformations as a search problem, with the cost function as the empirical cost of executing the program that results from a candidate transformation.

Almost all previous approaches to iterative optimisation deal with trivial search spaces that are the Cartesian product of some number of options (e.g. array padding and tiling and unrolling factors for a loop [12]), where all choices are legal. A notable exception to this is [13]. Our work similarly deals with search spaces that are themselves nontrivial to generate (see Section 4.1). Also, loop fusion is rarely included in iterative optimisation work, with [13], [14] being two largely isolated examples. In the first of these papers loop fusion is implicitly included in the action of generated space-time mappings, but appears to be applied in an ad hoc fashion with no mention of choosing fusion partitions etc (in fact, fusion is almost not mentioned at all) – the primary focus of the paper is on finding parallelisation transformations with good performance. In the second, a small experiment on four loops with no fusion–preventing dependencies finds that fusing all loops together gives the best reduction in energy use, but the main emphasis is on tiling and unrolling. Again, there is no mention of fusion partitions. In both papers there is no mention at all of array contraction.

4 Iterative Collective Loop Fusion

The choice of fusion partition on an LDG usually involves a trade-off in locality for different pairs of references, and so the best choice depends on how the locality characteristics of the program interact with the architecture on which it is being run. These include considerations such as issue width, clock rate, cache size, miss penalty and bandwidth limits. Hence, choosing a good fusion partition with respect to temporal locality is architecture dependent and far from trivial, which is why we employ search.

To perform iterative loop fusion exhaustively we simply require a method of enumerating all the legal fusion partitions for a given LDG, and the means to empirically test their run-times. The size of the search space, that is the number of legal fusion partitions, almost always makes testing each point in it unfeasible, so there must be some method of selecting a subset of the search space to test. This is a standard problem in iterative compilation. An extra complication though is the generation of the search space of legal transformations itself, which is discussed below.

4.1 Generating Legal Fusion Partitions

Although clusters within a fusion partition are not distinguished, it is useful to label them with identification (ID) numbers to reason about the enumeration of the fusion partitions for an LDG. Clusters are numbered from 1 to n giving a total ordering on the loops produced from a fusion partition.

The naive approach to generating fusion partitions of size n is to assign each node to a partition i with $1 \leq i \leq n$. The vast majority of these configurations will be illegal though, so a large number will have to be generated and tested to find each legal point. An alternative is to find some algorithmic way of enumerating only legal fusion partitions. The approach in this paper is based on node numbering, which is described below, followed by the enumeration algorithm.

Node numbering and range finding. Given a loop dependence graph, a target size of fusion partition, and a set of nodes with pre-assigned partition numbers, the forward node numbering procedure provides a test to determine the lower bound on the ID number of the partition to which any given (unassigned) node may belong.

Two directly connected nodes joined by at least one fusion–preventing edge must belong to different partitions. Consequently, given any path from a source to a sink, the nodes along the path can be numbered to show the earliest partition that they may belong to (as determined by this path) by grouping the nodes into sets separated by fusion–preventing edges and numbering the sets (and their elements) along the path consecutively. If a set contains a pre-assigned node with a value different from the parent set, then the set is split into two with the second set starting with the pre-assigned node and labelled with its value. Numbering along the path continues as before counting upward from the new value.

```
NumberNodesForwards(preassigned, LDG)
Description: Labels each unassigned node in the LDG with the
earliest partition that it may belong to.
Input: { LDG, a loop dependence graph
       { preassigned, a set of (node, partitionID) pairs
Output: An integer label for each node as a set of (node,
partitionID) pairs
(1)   $sources := \{(v, partitionID = 1) \mid$
          $v \in$ Sources(LDG) $\setminus$ Nodes(preassigned) $\}$
(2)   $labelled := preassigned \cup sources$
(3)   $unlabelled := \{v \mid v \in$ Nodes(LDG) $\setminus$ Nodes(labelled) $\}$
(4)   repeat
(5)     choose $v \in unlabelled$ s.t. Parents($v$) $\cap\ unlabelled = \emptyset$
(6)     $rank_v := 0$
(7)     foreach $p \in$ Parents($v$)
(8)       if $\exists e \in$ Joins($v$, $p$) s.t. FusionPreventing?(e)
(9)         $rank_{v,p} := 1+$ Rank($p, labelled$)
(10)      else
(11)        $rank_{v,p} :=$ Rank($p, labelled$)
(12)    $rank_v :=$ Maximum( $\{ rank_{v,p} \}$ )
(13)    $labelled := labelled \cup \{(v, rank_v)\}$
(14)    $unlabelled := unlabelled \setminus \{v\}$
(15)  until $unlabelled = \emptyset$
(16)  return $labelled \setminus preassigned$
```

Fig. 4. Forward node numbering algorithm

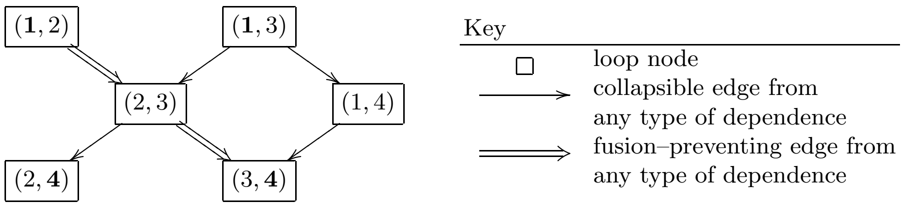

(1, 2)
(1, 3)
(2, 3)
(1, 4)
(2, 4)
(3, 4)
Key
loop node
collapsible edge from any type of dependence
fusion–preventing edge from any type of dependence

Fig. 5. An example showing the results produced by Ranges() when calculating possible partitionings into four clusters for a graph containing both collapsible and fusion–preventing edges. Each node is labelled with a (*minimum partition number*, *maximum partition number*) tuple, with numbers in bold indicating that the value results from the node being either a source or a sink in the graph.

If this procedure is repeated for all paths through the graph with each node being assigned the maximum value over all paths, then the final label P_{min} will denote the earliest possible partition that the node may belong to in this LDG with these pre-assigned nodes. A pseudocode for the algorithm is provided in Figure 4. The description makes use of several simple utility functions:

- Nodes(): returns the set of vertices from an aggregate data structure (either an LDG or a set of $(node, partitionID)$ pairs).
- Sources(): returns the set of root (source) vertices in the LDG.
- Parents(): returns the parents of a vertex (in the current LDG).
- Joins(): returns the set of edges that joins two vertices (in the current LDG).
- FusionPreventing?(): returns a Boolean depending on whether the edge is collapsible or not.
- Rank(): returns from a set of $(node, partitionID)$ pairs the partition ID (integer) of a given vertex.
- Maximum(): returns the maximum from a set of integers.

The algorithm does not actually enumerate all the paths through the LDG. Instead it successively selects nodes from the unassigned set only after all their parents have been processed.[1]

Given a maximum number of partitions, the same numbering can be repeated in reverse working from sinks to sources. This gives NumberNodesBackwards(), the result of which denotes the latest possible partition that a node may belong to, P_{max}. Taken together, the two procedures provide the range of partition IDs to which any unassigned node v may belong $P_{v,min} \leq ID_v \leq P_{v,max}$, and also the size of the range for that node $P_{v,max} - P_{v,min} + 1$. Any node with a range of sizes less than or equal to zero indicates that no legal fusion partitions of this size exist for this LDG. This information is provided by the Ranges() function, which essentially just calls NumberNodesForwards() and NumberNodesBackwards().

An example of the results produced by applying the Ranges() function to an example problem is given in Figure 5. The labelling of the graph shows for each node the earliest (minimum number) and latest (maximum number) cluster that it may belong to for the case of four partitions. Note that this is not the minimum number of partitions possible for this LDG.

Enumeration algorithm. The enumeration algorithm generates the fusion partitions of a given size for an LDG. It starts by finding the ranges of the nodes in the LDG, then choosing a $(node, range)$ pair. For the chosen node, the algorithm chooses a value in its range, treats the $(node, value)$ pair as a pre-assigned node, and recursively calls itself. At each step, the search is pruned if any partition will remain empty. For subsequent calls, a different value from the range of the last assigned node is chosen, until the range has been covered indicating that this recursive step is complete. Note that the ranges of unassigned nodes may change before each recursive function call, and that any unassigned node can be selected for enumeration within a call.

The enumeration algorithm is given in Figure 6. As well as the recursive call, it uses two other functions; Ranges(), explained above, and FusionPartition(),

[1] A similar algorithm to NumberNodesForwards() , without the notion of accommodating preassigned nodes, can be found in an early paper on the subject [3]. However, the authors do not apply the same technique in reverse, as described here, and do not attempt to enumerate different fusion partitions.

```
EnumerateFusionPartitions($LDG$, $size$, $fixed$)
Description: Enumerates the fusion partitions of an LDG
Input: $\begin{cases} LDG\text{, a loop dependence graph} \\ size\text{, the required size of fusion partition} \\ fixed\text{, a set of } (node, partitionID) \text{ pairs} \end{cases}$
Output: the set of fusion partitions of size $size$ in $LDG$
(1)   if Nodes($LDG$) \ Nodes($fixed$) = $\emptyset$ then return FusionPartition($fixed$)
(2)   $fps := \emptyset$
(3)   $ranges :=$ Ranges($LDG$, $size$, $fixed$)
(4)   if $\forall p \in \{1, \ldots, size\}\ \exists(v,p) \in fixed \bigvee \exists(v, r_{min}, r_{max}) \in ranges$ such that
      $(r_{min} \leq p \leq r_{max})$
(5)       choose $(v, rank_{v,min}, rank_{v,max})$ from $ranges$
(6)       for $i := rank_{v,min}$ to $rank_{v,max}$
(7)           $newFixed := fixed \cup \{(v,i)\}$
(8)           $fps := fps \cup$ EnumerateFusionPartitions($LDG$, $size$, $newFixed$)
(9)   return $fps$
```

Fig. 6. Fusion partition enumeration algorithm

which makes a fusion partition data structure from a list of (*node*, *partitionID*) pairs. In the current implementation there is no special criterion for choosing nodes to fix (they are taken in whatever order they are provided in by the function that calculates the ranges) or values from their ranges (currently they are taken sequentially, from bottom to top by the loop on line 6).

4.2 Search Heuristics and Search Space Reduction

Although generating legal fusion partitions is relatively cheap, the total number of them means that generating and storing all of them (i.e. the search space) before choosing points to tests would take far too much time and space (see Table 1). Consequently, there needs to be some way of selecting a region of the search space to generate. The choice of this region is governed by the characteristics of the points we hope to find, and therefore determined by the search heuristics themselves:

1. More array contraction is likely to be better.
2. A smaller size fusion partition (i.e. less clusters) is likely to better.

Both heuristics stem from the goal of improving memory performance. The heuristics are not independent. Given some initial LDG, it is necessary to fuse some loops (i.e. choose a fusion partition) to uncover any more array contraction. Note that the smallest partition size may not contain the fusion partition with the most contracted arrays. However, for a non-pathological LDG derived from a typical program, more fusion and more contraction are likely to be related. This last assumption allows us to use the second heuristic to guide the generation of points in the space with the assumption that they will include (the majority of) the good points as determined by the first heuristic.

```
GenerateTestCases($LDG$, $maxCandidates$, $minPartitions$, $maxPartitions$)
Description: Enumerates the fusion partitions of an LDG
Input: { $LDG$, a loop dependence graph
       { $maxCandidates$, the maximum number of fusion partitions to generate
       { $minPartitions$, the minimum size of fusion partition to generate
       { $maxPartitions$, the maximum size of fusion partition to generate
Output: fusion partitions of $LDG$
(1)   $candidates := \emptyset$
(2)   for $i := minPartitions$ to $maxPartitions$
(3)       $fps :=$ EnumerateFusionPartitions($LDG$, $i$, $\emptyset$)
(4)       $total := fps \cup candidates$
(5)       $candidates :=$ SelectBest($maxCandidates$, $total$)
(6)   return $candidates$
```

Fig. 7. Test case generation algorithm

Using the enumerating procedure, the overall algorithm for generating cases is given in Figure 7. The algorithm starts at small fusion partition sizes, and with each successive iteration the size of fusion partitions that are considered increases by one. Note that the amount of search space to generate (i.e. fusion partition size range) and the number of points to try are arguments supplied by the user. The function SelectBest() orders the set *total* based on the search heuristics (e.g. contraction, then partition size, then first come-first served) and then cuts it down to the first *maxCandidates* elements.

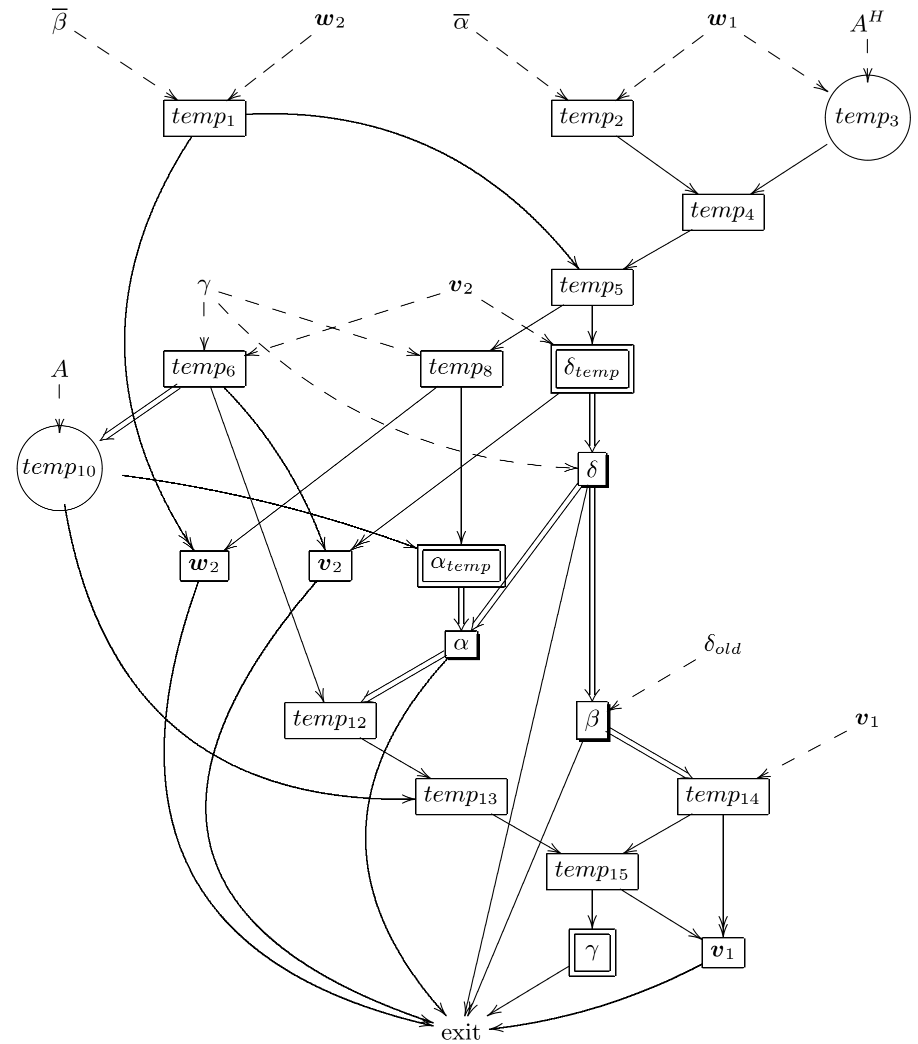
4.3 Code Generation

The only requirements on the code generated from the fusion partition of an LDG is that dependencies between partitions are respected in the final ordering of the loops generated from them, and similarly that the dependencies within a partition are respected in the ordering of the bodies from the original loops to form the body of the partition. The first requirement is automatically satisfied by ordering the loops according to the partition label sequence, and the second can be satisfied by a simple topological sort. Basic blocks are placed in the code between loops as early as is legal.

5 Experiments

5.1 Example LDG

The example is derived from the general step of a two-sided Krylov space algorithm, the fundamental component of several sparse linear solver and eigenvalue approximation algorithms [17], code for which is given in [15]. It is applied to a 3/4 dimensional simple stencil problem. A decorated version of the associated LDG is presented in Figure 8. All loops are conformable, and all edge weights are equal. Loop nodes are labeled with the variable they write to – scalars are denoted with greek letters ($\alpha, \beta \dots$), arrays with lower case letters or numbered



Key

	data live-in to or live-out of LDG region		
□	*simple node* – a data parallel operation on vectors such as addition, subtraction etc.	- - ->	input dependence (present for illustration)
▣	*reduction node* – the computation of the inner product of two vectors	→	true dependence
◯	*operator application node* – the application of a nearest neighbour averaging stencil in three or four dimensions	↠	anti dependence
❏	*basic block*	⇒	fusion–preventing true dependence

Fig. 8. LDG from two-sided Krylov space update

temporaries (w_2, $temp_9$ ...), and data for the stencil with capitals (A ...). Data that is live-in and dependencies to an `exit` node for data that is live-out are added for illustration purposes.

5.2 Enumeration of Fusion Partitions and Compilation Times

Table 1 gives the four smallest fusion partition sizes (column 1), the number of legal fusion partitions for that size (column 2), along with the time taken to generate them (column 3). The results are further broken down to show how many partitions of that size exist with a given amount of maximum array contraction (columns 4 – 8). For example, of the 80 fusion partitions of size 3, 24 have a maximum of 9 array contractions. Our heuristics prioritise partitions with the most contraction – i.e. the column marked 10, which has a total of eight partitions from different sizes. In fact, the best result was always produced by one of these fusion partitions (although not necessarily the smallest). The table shows that there are relatively few fusion partitions with the best characteristics according to our heuristics, and so restricting empirical testing to these preferred candidates would be cheap.

The time to find a solution depends on the time spent generating the search space and testing points, both of which are under user control. Consequently compile times are determined by how much search a user is willing to do to characterise the space. The optimum points can be found for our example by testing only eight points each time, (i.e. column four from Table 5.2) but this may not be enough in all cases. A characterisation of the search spaces for multiple benchmarks on different architectures is currently in progress. Compilation times are, however, expected to be relatively long – the approach is targeted at long-running scientific/embedded applications where the investment will pay off.

Table 1. Number of legal fusion partitions (FPs) of certain sizes, the time taken to generate them and how many partitions with a given amount of array contraction exist for that size

FP size	no. legal	time to enumerate	no. FPs with n contracted arrays				
	FPs	(in minutes)	10	9	8	7	6
3	80	1	2	24	39	13	2
4	3557	1	4	174	960	1395	792
5	63801	4	2	366	4974	17066	22362
6	633799	57	0	307	10350	71951	178862

5.3 Comparison Against Existing Fusion Techniques

Method. The first set of iterative search experiments compare our search technique against two algorithms representative of those in the literature that target array contraction, a greedy [9] and a max-flow min-cut [3] algorithm, as well as the original untransformed code (i.e. without any fusion/contraction). In all cases simple Aldor code is generated from the fusion partitions by our prototype tool, followed by compilation to C code using the Aldor compiler version

Table 2. Times in seconds for best search, control methods and untransformed code

stencil	machine	size	best search	greedy	max-flow min-cut	original
3D	Pentium III	50	136.8	186.3	163.6	329.6
	Pentium 4	70	55.3	64.2	76.6	122.3
4D	Pentium III	18	118.6	141.3	148.5	291.1
	Pentium 4	24	59.7	69.7	79.2	126.9

Table 3. Times for linear solve on 3D stencil (search vs. Fortran)

machine	size	best search	Fortran
Pentium III	30	43.4	64.1
	50	209.1	303.7
Pentium4	30	5.26	7.20
	50	24.5	33.6
	70	71.6	95.6

1.01 with aggressive inlining settings. This C code is compiled using the Intel C compiler (`icc`) version 8.0 to run on either a 1 GHz Pentium III (Coppermine) or a 2.6 GHz Pentium 4 (Northwood). Flags for `icc` were set to target the specific processor (`-xK/N`), perform all but the most aggressive optimisations (`-O2`) and instrument the code for profiling. Timings were generated by executing a program that calls the main function 1000 times, to give stable results.

Results. A comparison of the results produced for the first set of experiments by our search method, the control techniques and the original code is given in Table 2.[2] Our technique provides speedups of up to 2.45 over the original code, and up to 1.36 and 1.38 over a greedy and a max-flow min-cut algorithms respectively. The speedup over the original code shows that there are important gains to be had from this kind of technique, and the speedup over the other methods shows that search is necessary to get the full potential benefit of the transformations.

5.4 Comparison Against Fortran

Method. The second set of experiments provide some broad comparison of the performance of Aldor code transformed with our technique against a standard Fortran 77 package containing an equivalent algorithm, QMRpack [16]. This was compiled using the Intel Fortran compiler version 8.0 (`ifc`) with the same flags as for `icc`, but also with cross-file inlining and the highest level of optimisation (`-O3`) to enable high-level transformations such as loop fusion. QMRpack had to be modified slightly to make the two codes more similar, by adding a stencil

[2] For further results, a more in-depth analysis and a discussion of how the best solution changes with respect to the problem and the architecture in question, please see [15].

and removing some conditionals that skip steps based on floating point error tolerances and may have prevented transformations such as loop fusion. Additionally, the Aldor code had to be augmented with some extra code to make it into a full QMR solver.

Results. Results for the second set of experiments are presented in Table 3. The transformed version outperforms the Fortran version with the relative performance gain being ≈ 1.46 on the Pentium III and ≈ 1.35 on the Pentium 4. These results show that using an advanced language does not necessarily mean sacrificing performance compared to lower-level languages.

6 Conclusion and Future Work

Iterative collective loop fusion applies heuristically guided search to select the best candidate from several fusion partition sizes and contraction amounts, and provides important performance benefits over the alternative techniques with speedups of up to 1.38. The overall approach of applying such a technique to a high-level language that is inherently very modular is promising, with performance improvements over navely generated code of up to 2.45, combining elegance of expression with performance more usually associated with traditional imperative languages.

The two most important extensions to this work will be to gather further results using more machines and LDGs derived from other codes, and to investigate how loop fusion and array contraction interact with subsequent single loop optimisations such as loop unrolling or software pipelining. In addition, investigating how to formulate the loop fusion/array contraction problem for other abstract frameworks such as the polytope model would be interesting.

References

1. Watt, S.M. Aldor Users Guide. `http://www.aldor.org`
2. Kennedy, K., McKinley, K.S. Typed Fusion with Applications to Parallel and Sequential Code Generation. Techreport TR93-208. Rice University Dept. of Computer Science (1993)
3. Gao, G.R., Olsen, R., Sarkar, V., Thekkath, R. Collective Loop Fusion for Array Contraction. Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing. Springer-Verlag (1992) 281–295
4. Lewis, E.C., Lin, C., Snyder, L. The implementation and evaluation of fusion and contraction in array languages. In: PLDI '98. Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. ACM Press (1998) 50–59
5. Song, Y., Xu, R., Wang, C., Li, Z. Data locality enhancement by memory reduction. In: ICS '01. Proceedings of the 15th international conference on Supercomputing. ACM Press (2001) 50–64
6. Ding, C., Kennedy, K. The Memory Bandwidth Bottleneck and its Amelioration by a Compiler. In: IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing. IEEE Computer Society (2000) 181–

7. Singhai, S., McKinley, K.S. A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. In: The Computer Journal **40, 6** (340–355) 1997
8. Darte, A. On the Complexity of Loop Fusion. In: PACT '99. Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society (1999) 149–
9. Kennedy, K. Fast greedy weighted fusion. In: ICS '00. ACM Press (2000) 131–140
10. Megiddo, N., Sarkar, V. Optimal weighted loop fusion for parallel programs. In: SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures. ACM Press (1997) 282–291
11. Parello, D., Temam, O., Verdun, J-M. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance: matrix-multiply revisited. In: Supercomputing '02. IEEE Computer Society Press (2002) 1–11
12. Kisuki, T., Knijnenburg, P.M.W., O'Boyle, M.F.P. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In: PACT '00. IEEE Computer Society (2000) 237–
13. Nisbet, A.P. GAPS: Iterative Feedback Directed Parallelisation Using Genetic Algorithms. In: Proceedings of Workshop on Profile and Feedback-Directed Compilation at PACT98, Paris, France
14. Gheorghita, S.V., Corporaal, H., Basten, T. Iterative Compilation for Energy Reduction. Journal of Embedded Computing (To appear in 2005)
15. Ashby, T.J. Design and Optimisation of Scientific Programs in a Categorical Language. PhD Thesis, University of Edinburgh (2005)
16. Freund, R., Nachtigal, N. QMRpack. `http://www.netlib.org/linalg/qmr/`
17. Greenbaum, A. Iterative methods for solving linear systems. Society for Industrial and Applied Mathematics (1997)