An Interprocedural Framework for Placement of Asynchronous I/O Operations *

Gagan Agrawal and Anurag Acharya and Joel Saltz
UMIACS and Department of Computer Science
University of Maryland
College Park, MD 20742
(301)-405-2756
{gagan,acha,saltz}@cs.umd.edu

Abstract

Overlapping memory accesses with computations is a standard technique for improving performance on modern architectures, which have deep memory hierarchies. In this paper, we present a compiler technique for overlapping accesses to secondary memory (disks) with computation. We have developed an Interprocedural Balanced Code Placement (IBCP) framework, which performs analysis on arbitrary recursive procedures and arbitrary control flow and replaces synchronous I/O operations with a balanced pair of asynchronous operations. We demonstrate how this analysis is useful for applications which perform frequent and large accesses to secondary memory, including applications which snapshot or checkpoint their computations or out-of-core applications.

1 Introduction

Modern architectures have large number of memory hierarchies. Processors have one or two levels of cache, followed by primary memory (RAM), secondary memory (disks) and tertiary memory. The cost of data access increases rapidly with the depth of access. Achieving and sustaining good performance in presence of deep memory hierarchies is a very important problem and has received significant attention in the last few years.

Several research projects have worked on code transformations to improve locality [6, 31, 32]. Most of the programs still spend considerable amount of their time in accessing data from memory at a deeper level in the hierarchy. The overhead of deep memory accesses can be reduced by using asynchronous operations overlapped with computations. Substantial work has been done on compiler analysis for overlapping memory accesses with computations, mostly in the context of reducing overheads of cache stalls [8, 23].

Several classes of applications involve large and frequent accesses to data stored in the secondary memory. The need for reading or writing data in secondary memory can arise for

*This work was supported by NSF under grant No. ASC 9213821, by ONR under contract No. N00014-93-1-0158, by ARPA under the Scalable I/O project (Caltech Subcontract 9503) and by NASA/ARPA under contract No. NAG-1-1485. The authors assume all responsibility for the contents of the paper.

several reasons: 1) the program may operate on data structures which do not fit into the primary memory of the processor (*out-of-core* programs), 2) the state of the computation may be output (for future analysis) several times during the execution of the program, or 3) the main data structures may be frequently checkpointed so as to be able to restart in presence of failures. For programs that use a large fraction of the primary memory, large write requests would overflow the file cache and would cause the program to be stalled till the data is written into the disks. Because of the relatively low disk bandwidth, writing into disks can take a large amount of time. If such I/O requests are frequent, the performance of the program will be severely degraded. This degradation can be avoided if the disk accesses can be overlapped with the computation. Several current operating systems provide asynchronous I/O operations for this reason. We believe that compiler analysis can automate the usage of asynchronous I/O operations, thus hiding the architectural complexities from the application programmers.

To be able to overlap disk accesses with computation, it is important for the compiler to analyze code across procedure boundaries. In contrast, the existing compiler analysis for overlapping cache stalls or communication has considered analysis within a single procedure and usually does not consider moving operations outside conditionals or in presence of arbitrary flow of control.

In this paper, we present an Interprocedural Balanced Code Placement (IBCP) framework for overlapping large latency operations with computation. This framework is able to deal with arbitrary control flow as well as arbitrary recursive procedures. Each synchronous large latency operation is replaced by a balanced pair of asynchronous operations. The asynchronous operations are placed to achieve overlap with the computation, while maintaining correctness and safety. In general, our analysis can be used for any large latency operation, including disk accesses, frame buffer writes, network I/O, co-processor operations and remote memory accesses.

In this paper, we have focussed on the use of this analysis for the placement of disk accesses. We have implemented a Fortran source to source transformation tool, which performs the IBCP analysis. This tool is based upon the Parascope/D System Fortran front end.

We have used two applications for demonstrating the efficacy of our scheme: `dsmc-3d`, a particle simulation code which periodically outputs its state and `satellite`, a satellite data processing template which repeatedly modifies an out-of-core image. Our results show that use of compiler placed asynchronous operations can reduce the I/O overhead by 30%-70% and improve the overall performance by 15% - 20%. Performing interprocedural analysis for placement was critical for getting better performance in both these applications, almost no overlap would have been possible if the analysis was restricted intraprocedurally.

The rest of the paper is organized as follows. In Section 2, we further examine the classes of applications which have large I/O requirements. In Section 3, we state the requirements for our code placement framework and motivate our analysis. In Section 4, we present our interprocedural analysis. We present experimental results in Section 5. We briefly compare our work with related work in Section 6 and conclude in Section 7.

2 I/O Intensive Applications

In this section, we discuss how the need for large and frequent I/O accesses arises in several classes of scientific applications. There are at least three different scenarios in which the application may need to access secondary memory frequently and therefore, can potentially become I/O bound.

Snapshoting. By snapshoting, we mean the frequent copying of the state of the computation to disks. There are two main reasons for snapshoting: the computation has a time-dependent solution (as opposed to a steady state solution) or the user may want to visualize the progress of the computation. In the former case, a series of program states are written to disks for post-processing. In the latter case, the state of the computation is periodically piped into an appropriate visualization tool. Examples of applications that require snapshoting include Direct Simulation Monte Carlo (DSMC) [5], simulation of a flame sweeping through a volume [27], airplane wake simulations [19], etc. The amount of data generated per time-step is often large, e.g., if one million particle are simulated in a particular run of the three-dimensional `dsmc-3d` code, then each snapshot is 24 MB (3 double precision numbers per particle).

Checkpointing. Checkpointing is often required for long running applications which can get interrupted for a variety of reasons [12]. Checkpointing is also used for parametric studies, i.e., modifying some of the checkpointed values and restarting the computation [11].

Out-of-Core Computations. Several scientific and engineering computations operate on large data structures which do not fit into the main memory. Such codes need to access the secondary memory very frequently. Examples include several sensor data processing codes which perform out-of-core reductions on images which are several hundred MB in size. Restructuring the computation can often reduce the frequency and increase the granularity of secondary memory access [1, 6], however, such codes can still spend significant amount of their time in I/O operations.

3 Problem Definition

In this section, we define the interprocedural balanced code placement problem and motivate our analysis framework.

As we stated in Section 2, scientific applications may make frequent and large I/O requests for a variety of reasons. In the applications which checkpoint or snapshot the progress of the computation, the programmer may explicitly insert synchronous write operations. In the out-of-core applications, an initial phase of the compilation may determine when disk operations need to be made and correspondingly may insert synchronous read or write operations [6, 26].

For our purpose, we consider original program text (or a compiler processed representation) which has explicit calls to read or write operations. We denote by Op, any such synchronous operation, which the compiler may want to replace by a corresponding asynchronous or split-phase operation. In general, this operation OP has a number of parameters (usually specifying the buffer which is to be written out or filled in and their dimensions). The operation can result in a modification to some of these parameters (e.g., the contents of the array may be read in) and/or modification to the contents of a disk. We assume that this operation does not result in any other change of state.

To overlap this operation with computation, the compiler will have to place corresponding START_OP and END_OP operations in the program text. A START_OP operation starts an asynchronous I/O operation and the corresponding END_OP ensures that the operation has completed.

The goal of the analysis is to place START_OP as early as possible and END_OP as late as possible, i.e., allow as much overlap of the computation with disk operations as possible without violating the following guarantees: as follows:

- *Balance and Safety.* Consider any control flow path from the beginning of the program to the end of the program. Suppose this path contains n ($n \geq 0$) occurrences of particular operation OP (which reads or writes a buffer). After the compiler inserts asynchronous operations, this path will include exactly n occurrences of START_OP and exactly n occurrences of END_OP.

- *Correctness.* The effect of the asynchronous operation will the same as the effect of the corresponding synchronous operation i.e., the modifications to the parameters of the operation and the contents of the disks will be the same as the synchronous operation would have resulted in. Further, any access to any variable in the program, will have the same result as it would have with synchronous operations.

Note that the first requirement mentioned above has two implications:

- The occurrences of START_OP and END_OP will match each other irrespective of the control flow paths taken during the execution of the program. This is also known as the balanced code placement property [16].

- In any execution of the program, there will exactly be the same number of asynchronous operations as there were synchronous operations, i.e., under no circumstances, the new placement will increase the number of I/O operations on any path. This is known as the *safety* property [22].

Note that we do not consider redundancy elimination as part of our analysis i.e., there is always exactly one asynchronous operation placed for each synchronous operation in the original program text. In case of checkpointing or snapshoting, the application programmer expects to see the same number of outputs as he/she had inserted initially. In the case of out-of-core programs, the initial phase of the compilation can possibly generate redundant requests. In this case, some redundancy elimination analysis like Interprocedural Partial Redundancy Elimination (IPRE) [4] can be applied before applying the analysis we present in this paper.

4 Interprocedural Balanced Code Placement Framework

In the previous section, we described the requirements of our interprocedural code placement framework. In this section, we present Interprocedural Balanced Code Placement (IBCP) framework, which performs such placement. We first describe the full program representation we use, then we present the interprocedural analysis on our full program representation. Finally, we present the intraprocedural phase of our analysis.

Data flow analysis has been a key method for performing various optimization in the program text, without imposing any restrictions on control-flow [20]. Various intraprocedural code placement frameworks like Partial Redundancy Elimination [22] and Give-N-Take [16] perform data flow analysis on Control Flow Graph (CFG) of a single procedure.

A number of different program representations have been used for various interprocedural data flow problems. The most commonly used program representation is a Call Graph [14]. A call graph is a directed multi graph, which has a single node for each procedure in the program. A directed edge from node i to node j means that procedure i calls procedure j. Call Graph is a very concise representation and no information is available in a call graph about flow of control between different call sites within a single procedure. On the other extreme, Myer's SuperGraph [24] is a very detailed representation. SuperGraph is constructed by linking control flow graphs of procedures by inserting edges from call site in the caller to start node in callee. The total number of nodes in SuperGraph can get very large and consequently the solution may take a long time to converge. We have developed a new full program representation which preserves information about the call sites but is more concise than a SuperGraph.

4.1 Program Representation

We now define the full program representation we use for the purpose of our analysis. We assume that a variable is either global to the entire program or is local to a single procedure. We further assume that all parameters are passed by reference. We do not consider the possibility of aliasing in our discussion.

We define a basic block to consist of consecutive statements in the program text without any intervening procedure calls or return statements, and no branching except at the beginning and at the end. A procedure can then be partitioned into a set of basic blocks, a set of call statements and a set of return statements. Each call statement is a *call site* of the procedure invoked there.

In our program presentation, the main idea is to construct *blocks of code* within each procedure. A block of code comprises of basic blocks which do not have any call statement between them. In the directed graph we define below, each edge e corresponds to a block of code $B(e)$. The nodes of the graph specify the control-flow relationships between the blocks of code.

```
Program Foo
  Arrays A, B
  d = ....
  Call Q(c)                ... cs 1
  Do i = 1, 100
    if cond then
      Call R(A,d)          ... cs 2
      Call Q(c)            ... cs 3
    else
      A = ...
    endif
    Call S(B,d)            ... cs 4
    Call P(A,d)            ... cs 5
    Call P(B,d)            ... cs 6
  Enddo
End

Procedure P(x,y)
  WRITE_OP(x,y)
  ..other computations ..
End
Procedure Q(z)
  z = ...z...
End
Procedure R(x,y)
  x = ...x...
  ..other computations ..
End
Procedure S(x,y)
  x = ...x...
  ..other computations ..
End
```

Figure 1: An example program: call sites are labelled

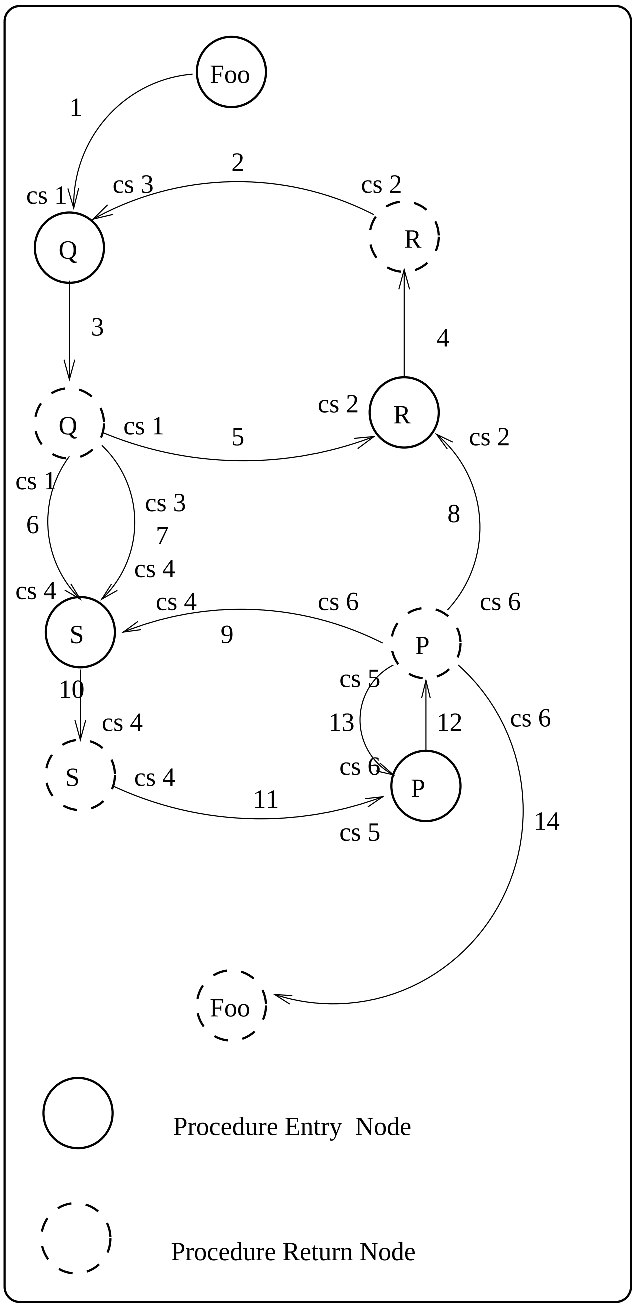

Figure 2: *FPR* for program in the left. Edge numbers and call sites at which edges start/end (whenever applicable) are marked in the Figure.

Full Program Representation: (*FPR*) is a directed multigraph $G = (V, E)$, where the set of nodes V consists of an entry node and a return node for each procedure in the program. For procedure i, the entry node is denoted by s_i and the return node is denoted by r_i. Edges are inserted in the following cases:

1. Procedures i and j are invoked by procedure k at call sites cs_1 and cs_2 respectively and there is a path in the CFG of k from cs_1 to cs_2 which does not include any other call statements. Edge (r_i, s_j) exists in this case. The call site cs_1 is associated with the start of this edge and the call site cs_2 is associated with its end. The block of code $B(e)$ consists of basic blocks of procedure k which may be visited in any control flow path p from cs_1 to cs_2, such that the path p does not include any other call statements.

2. Procedure i invokes procedure j at call site cs and there is a path in the CFG of i from the entry node of procedure i to cs which does not include any other call statements. In this case, edge (s_i, s_j) exists. The call site cs is associated with the end of this edge. The block of code $B(e)$ consists of basic blocks of procedure i which may be visited in any control flow path p from start of i to cs, such that the path p does not include any other call statement.

3. Procedure j invokes procedure i at call site cs and there is a path in the CFG of j from call site cs to a return statement within procedure j which does not include any other call statements. In this case, edge (r_i, r_j) exists. The call site cs is associated with the start of the this edge. The block of code $B(e)$ consists of basic blocks of procedure j which may be visited in any control flow path p from cs to a return statement of j, such that the path p does not include any call statements.

4. In a procedure i, there is a possible flow of control from entry node to a return statement, without any call statements. In this case, edge (s_i, r_i) exists. The block of code $B(e)$ consists of basic blocks of procedure i which may be visited in any control flow path p from start of i to a return statement in i, such that the path p does not include any call statements.

An example program and its *FPR* are shown in Figures 1 and 2 respectively.

A block of code is the primary unit of placement in our analysis, i.e. we initially consider placement only at the beginning and end of a block of code. Note that a basic block in a block of code may or may not be visited along a given control flow path from source to sink of the edge. Also, a basic block may belong to several blocks of code. This is taken into account during intraprocedural analysis done for determining final local placement, which we discuss in Section 4.6.

The following information is pre-computed and assumed to be available during our analysis phase. For each edge in *FPR*, we compute the list of variables modified in the corresponding block of code. Similarly, we also compute the list of variables referred to in each block of code. These sets are used by the TRANS_e functions defined later. For each procedure, we also compute the list of variables modified and referred to by the procedure or any of the procedures invoked by this procedure. In the absence of aliasing, this information can easily be computed by flow-insensitive interprocedural analysis in time linear to the size of call graph of the program [10]. This information is used by the FSUM_{cs} functions defined later.

4.2 Candidates for Placement

We now introduce the format of the read and write operations that we assume. We are only interested in the parameters which can effect the placement of the operation and not in the architecture dependent parameters which come in the exact format of the statements. A read or write operation, for our purposes, has the format

$$\text{O}\textsc{p}(Arrayname, s1, s2....)$$

where, *Arrayname* is the name of the array which is to be read in or written out. $s1, s2, ...$, are various *size specifiers*, they specify the size of the array being input/output or the section of the array which is to be input/output.

The operation OP results in either a modification to the contents of the array *Arrayname* (in case of reads) or a modification of the contents of an attached disk (in case of writes). The parameters $s1, s2$, ..., or any other program variables are not modified.

For the purpose of our discussion, we refer to any synchronous operation, which the compiler may want to replace by asynchronous operation, as a *candidate* for placement. We refer to the parameters of a candidate as its *influencers*.

4.3 Interprocedural Analysis

We now present the IBCP scheme we have developed. We use the terms *edge* and *block of code* interchangeably in this section.

Our method is based upon the notions of *availability* and *anticipability*. Availability of a candidate C at any point p in the program means that C is currently placed on all of the paths leading to p and if C were to be placed at this point, C will have the same result as the result of the last occurrence on any of the paths. Anticipability of a candidate C at a point p in the program means that C is currently placed on all the paths leading from p, and if C were to be placed at p, C will have the same result as the result of the first occurrence on any of the paths. We use anticipability for determining placement of START_OP and availability for determining placement of END_OP. Both anticipability and availability are computed for the beginning and the end of each block of code (edge) in the FPR. For a candidate $\mathcal{C}$, anticipability at the beginning of a block of code e is denoted by $\text{ANTIN}_{\mathcal{C}}(e)$ and anticipability at the end of the block of code is denoted by $\text{ANTOUT}_{\mathcal{C}}(e)$. Similarly, availability at the beginning of the block of code is denoted by $\text{AVIN}_{\mathcal{C}}(e)$ and availability at the end of the block of code is denoted by $\text{AVOUT}_{\mathcal{C}}(e)$.

We now list the major problems that need to be addressed in our analysis:

1. A procedure containing a candidate (or a procedure invoking such a procedure) may be invoked at multiple call sites, possibly with different sets of actual parameters. (e.g. in the

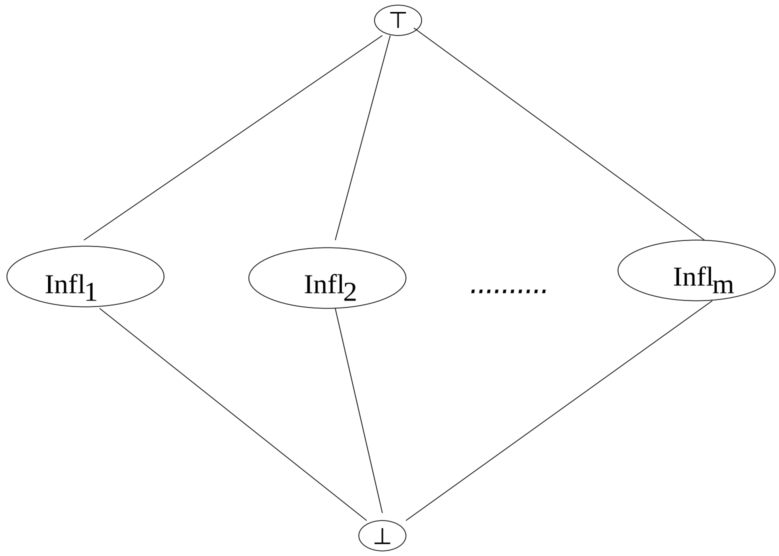

Figure 3: Lattice used in the data flow problems

code shown in Figure 1, procedure P is invoked at two call sites with different parameters). While considering placement of the candidate outside the procedure it is originally placed, asynchronous operations must be placed corresponding to each invocation of the procedure in which the candidate was originally placed.

2. For placement of a candidate in a procedure, all influencers of the candidate must be visible inside that procedure, i.e. each of them is either a global variable, a formal parameter or a local variable. (e.g. in the code shown in Figure 1, no placement will be possible inside procedure Q).

3. If a procedure is invoked at several call sites in the program, our program representation shows paths from edges ending at a call site calling this procedure to the edges starting at other call sites for this procedure. (e.g. in Figure 2, there is a path from edge 1 to edge 3 to edge 7. Edge 1 ends at call site $cs1$ whereas edge 7 starts at callsite $cs3$.) These paths are never taken and the data flow analysis must not lose accuracy because of such paths in the graph.

Consider again the example given in Figure 2. We denote the synchronous write operation in Procedure P as candidate $\mathcal{C}$. If we decide to do a placement outside procedure P, we really need to consider two different candidates, one each corresponding to the calls to the procedure P at call sites $cs5$ and $cs6$. We will refer to these two candidates as $\mathcal{C}\_cs5$ and $\mathcal{C}\_cs6$ respectively. Note that if the original occurrence of a candidate is in a nested sequence of procedure calls, and if each of these procedures is called at several call sites, we can potentially have an exponential number of candidates to consider.

4.3.1 Lattice for Data Flow Problems

If a candidate is available or anticipable at a certain point in the program, it is always with a certain set of influencers, which will be used in placing it (i.e. if it is decided that the candidate is to be placed at this location). For this reason, we use a three-level lattice for the data flow problems. The lattice is shown in Figure 3. Each middle element in the lattice refers to a list of influencers, i.e. $Infl_i = < v_1, v_2, \ldots, v_n >$. We define the following functions on this lattice: $\vee$ and $\wedge$ are standard binary join and meet operators. For ease in presenting our data flow equations, we use $\bigvee$ and $\bigwedge$ as confluence join and meet operators i.e. for computing join and meet respectively over a set of elements. $\neg$ is a unary operator which returns $\perp$ when applied to a list of influencers or to $\top$ and returns $\top$ when applied to $\perp$.

4.3.2 Terminology

We use the following terms to specify the data flow equations in this paper. We had defined our program representation earlier in Section 4.1. In our Full Program Representation (*FPR*), the entry node corresponding to the main procedure is referred to as BEGIN node and the return node corresponding to the main is referred to as the END node.

The set of procedure entry nodes is represented by $\mathcal{E}$ and the set of procedure return nodes is represented by $\mathcal{R}$. Consider an edge $e = (v, w)$. The source of e (i.e. the node v) is referred to as $So(e)$ and the sink of e (i.e. the node w) is referred to as $Si(e)$. We use $pred(e)$ to refer to the set of edges whose sink is v. We denote by $succ(e)$ the set of edges whose source is w.

If the sink of the edge e is a procedure entry node, then the call site associated with the end of edge e is denoted by $Si'(e)$. We use $succ'(e)$ to refer to the set of edges which have the call site $Si'(e)$ associated at their start. Alternatively, if the source of the edge e is a procedure return node, then the call site associated with the start of edge e is denoted by $So'(e)$. We refer by $pred'(e)$ the set of edges which are associated with the call site $So'(e)$ at their end.

Consider any edge e whose source is a procedure entry node. The set $cobeg(e)$ comprises of edges whose source is the same as the source of edge e. If an edge e has a procedure return node as the source and if cs is the call site associated with the start of the edge e, then the set $cobeg(e)$ comprises of the edges which have the call site cs associated at their start.

Next, consider any edge e whose sink is a procedure return node. The set $coend(e)$ comprises of the edges whose sink is the same as the sink of the edge e. If an edge e has a procedure entry node as the source and if cs is the call site associated with the end of the edge e, then the set $coend(e)$ comprises of edges which have the call site cs associated at their end.

The sets $pred(e)$, $pred'(e)$, $succ(e)$, $succ'(e)$, $cobeg(e)$ and $coend(e)$ for edges in the Graph shown in Figure 1 are shown in Figure 4.

At any call site cs, the set of actual parameters passed is ap_{cs} and the j^{th} actual parameter is $ap_{cs}(j)$. The set of formal parameters of the procedure invoked at the call site cs is fp_{cs}. (Clearly, this set is the same for all call sites which invoke this procedure). The j^{th} formal parameter is denoted by $fp_{cs}(j)$. The set of global variables in the program is gv.

e	$pred(e)$	$pred'(e)$	$succ(e)$	$succ'(e)$	$cobeg(e)$	$coend(e)$
1	-	-	3	5,6	1	1
2	4	5,8	3	7	2	2
3	1,2	-	5,6,7	-	3	3
4	5,8	-	2	-	4	4
5	3	1	4	2	5,6	5,8
6	3	1	10	11	5,6	6,7,9
7	3	2	10	11	7	6,7,9
8	12	13	4	2	8,9,14	5,8
9	12	13	10	11	8,9,14	6,7,9
10	6,7,9	-	11	-	10	10
11	10	6,7,9	12	13	11	11
12	11,13	-	8,9,13,14	-	12	12
13	12	11	12	8,9,14	13	13
14	12	13	-	-	8,9,14	14

Figure 4: $pred(e)$, $pred'(e)$, $succ(e)$ and $succ'(e)$ sets for Graph in Figure 2

4.4 Placement of START_OP

We determine the placement of START_OP by computing anticipability of the candidates. The key idea is to place the candidates at the earliest points where they are still anticipable. This allows for maximum overlap of the operation with the computation without violating the safety, correctness or balance guarantees.

The equations for computing anticapability and determining placement are shown in in the Figure 6. All unknowns are initialized with $\top$. This state means that the candidate may be anticipable, but we do not yet know what the list of influencers will be, if it is anticipable. The bottom element in the lattice means that the candidate is not anticipable.

Initially, the local data flow property ANTLOC(i) of the edges in the graph is determined. (For a block of code, ANTLOC means that there is an occurrence of this candidate inside the block which can be moved to the beginning of the block.) Let $\mathcal{C}$ be the candidate for placement. Consider any block of code (edge) i in which the original occurrence of the operation is placed. From the beginning of this block of code, if the occurrence of the candidate is not enclosed in any conditional or loop, and none of its influencers are modified or referred to then, we mark

$$\mathrm{ANTLOC}_{\mathcal{C}}(i) \;=\; Infl_c$$

Here, $Infl_c$ is the list of parameters of the call to the operation. Only if $\mathrm{ANTLOC}_{\mathcal{C}}(i)$ is $Infl_c$ for all blocks of code i in which the original occurrence of this operation occurs, we can consider moving the START_OP earlier than the original location.

Two sets of functions FTRANS1 and FSUM1 summarize the mod/ref information, and are used for propagating data flow information. If we are considering early placement of a write operation, $\mathrm{FTRANS1}_e[< Arrayname, s1, s2, .. >]$, for an edge e, returns the same list if none of the influencers in the list is modified in the block of code associated with this edge. If any

$$\mathrm{T1}(v_i) = \begin{cases} v_i & \text{if } v_i \in gv \\ fp_{cs}(j) & \text{if } v_i = ap_{cs}(j) \end{cases}$$

$$\mathrm{RNM1}_{cs}[< v_1, \ldots, v_n >] = \begin{cases} \perp & \text{if } \exists i,\ (v_i \notin gv) \wedge (\forall j\,, v_i \neq ap_{cs}(j)) \\ < \mathrm{T1}(v_1), \ldots, \mathrm{T1}(v_n) > & \text{otherwise} \end{cases} \quad (1)$$

$$\mathrm{T2}(v_i) = \begin{cases} v_i & \text{if } v_i \in gv \\ ap_{cs}(j) & \text{if } v_i = fp_{cs}(j) \end{cases}$$

$$\mathrm{RNM2}_{cs}[< v_1, \ldots, v_n >] = \begin{cases} \perp & \text{if } \exists i,\ (v_i \notin gv) \wedge (\forall j\,, v_i \neq fp_{cs}(j)) \\ < \mathrm{T2}(v_1), \ldots, \mathrm{T2}(v_n) > & \text{otherwise} \end{cases} \quad (2)$$

Figure 5: Renaming functions

of these influencers is modified, this function returns $\perp$. If we are considering early placement of a read operation, $\mathrm{FTRANS1}_e[< Arrayname, s1, s2, ..]$ for an edge e, returns the same list if $Arrayname$ is not referred to and the other influencers s1,s2,.. are not modified. $\mathrm{FTRANS1}_e[\top]$ and $\mathrm{FTRANS1}_e[\perp]$ are defined to be $\top$ and $\perp$ respectively.

For a call site cs which invokes procedure p, $\mathrm{FSUM1}_{cs}[< Arrayname, s1, s2, .. >]$ returns the same list if none of the influencers in the list is modified by the procedure p (or by a procedure invoked by p). Otherwise $\perp$ is returned. For a read operation, $\mathrm{FSUM1}_{cs}[< Arrayname, s1, s2, .. >]$ checks if $Arrayname$ is not referred to and the other influencers s1,s2,.. are not modified. $\mathrm{FSUM1}_{cs}[\top]$ always returns $\top$ and $\mathrm{FSUM1}_{cs}[\perp]$ always returns $\perp$.

$\mathrm{OCR}_{\mathcal{C}}(cs)$ determine if the procedure p (or any procedure invoked by p) includes any occurrence of the candidate $\mathcal{C}$. (Clearly, this will be the same for all call sites which call procedure p). Whenever there is no scope for ambiguity, we drop the subscript $\mathcal{C}$. $\mathrm{OCR}(cs)$ returns $\top$ or true when there is an occurrence of the candidate in the procedure p and $\perp$ (or false) when there is no occurrence of the candidate at procedure p.

For renaming of formal parameters at call sites, we define two functions $\mathrm{RNM1}_{cs}$ and $\mathrm{RNM2}_{cs}$ (see Figure 5). Suppose a candidate is anticipable at a call site cs with a list of influencers $Infl_i$. The function $\mathrm{RNM1}_{cs}$ determines if this candidate can be anticipable inside the procedure invoked at cs, and if so, with what list of influencers. If any of influencers is neither a global variable nor an actual parameter at cs, $\mathrm{RNM1}_{cs}$ returns $\perp$, otherwise, each actual parameter in the list is replaced by corresponding formal parameter. $\mathrm{RNM1}_{cs}[\top]$ and $\mathrm{RNM1}_{cs}[\perp]$ are defined to be $\top$ and $\perp$ respectively. Suppose a candidate is anticipable at the beginning of a procedure and let cs be one of the call sites which invoke this procedure. $\mathrm{RNM2}_{cs}$ determines if this candidate will be anticipable at the entry of the edges which end at call site cs. If any of the influencers of the candidate inside the procedure is neither a global variable, nor a formal parameter, then $\mathrm{RNM2}_{cs}$ returns $\perp$. Otherwise, each formal parameter is replaced by the actual parameter at call site cs.

The equations for propagation of anticipability (Figure 6) can be explained as follows. Con-

$$\text{ANTOUT}_{\mathcal{C}}(e) = \begin{cases} \perp & \text{if } Si(e) \text{ is END node} \\ \bigwedge_{s \in succ(e)} (\, \text{RNM2}_{So'(s)}[\, \text{ANTIN}_{\mathcal{C}}(s)\,]\,) & \text{if } Si(e) \in \mathcal{R} \\ \text{FSUM1}_{Si'(e)}[\, \bigwedge_{s' \in succ'(e)} \text{ANTIN}_{\mathcal{C}}(s')\,] & \text{if } (Si(e) \in \mathcal{E}) \wedge (\, \neg \text{OCR}_{\mathcal{C}}(Si'(e))\,) \end{cases} \quad (3)$$

$$\text{ANTOUT}_{\mathcal{C}\_Si'(e)}(e) = \text{RNM1}_{Si'(e)}[\, \bigwedge_{s \in succ(e)} \text{ANTIN}_{\mathcal{C}}(s)\,] \quad \text{if } (Si(e) \in \mathcal{E}) \wedge (\, \text{OCR}_{\mathcal{C}}(Si'(e))\,) \quad (4)$$

$$\text{ANTIN}_{\mathcal{C}}(e) = \begin{cases} \perp & \text{if } So(e) \text{ is BEGIN node} \\ \text{ANTLOC}_{\mathcal{C}}(e) \vee & \\ \quad (\bigwedge_{b \in cobeg(e)} \text{FTRANS1}_b[\text{ANTOUT}_{\mathcal{C}}(b)]\,) & \text{otherwise} \end{cases} \quad (5)$$

$$\text{INS\_IN}_{\mathcal{C}}(e) = \text{ANTOUT}_{\mathcal{C}}(e) \wedge (\neg \text{ANTIN}_{\mathcal{C}}(e)) \quad (6)$$

$$\text{INS\_BEG}_{\mathcal{C}}(e) = \begin{cases} \text{ANTIN}_{\mathcal{C}}(e) \wedge & \\ (\bigwedge_{p \in pred(e)} \neg \text{ANTOUT}_{\mathcal{C}\_Si'(p)}(p)) & \text{if } (So(e) \in \mathcal{E}) \wedge (\, \text{OCR}_{\mathcal{C}}(Si'(p))\,) \\ \text{ANTIN}_{\mathcal{C}}(e) \wedge (\bigwedge_{p \in pred(e)} \neg \text{ANTOUT}_{\mathcal{C}}(p)) & \\ \quad \wedge (\bigwedge_{p' \in pred'(e)} \neg \text{ANTOUT}_{\mathcal{C}}(p')) & \text{if } So(e) \in \mathcal{R} \end{cases} \quad (7)$$

Figure 6: Data Flow Equations for Placement of START_OP

sider an edge e whose sink is a procedure return node. A candidate will be anticipable at the end of this edge e if the following holds: This candidate should be anticipable at the beginning of any edge s which starts at this procedure return node (i.e. $s \in succ(e)$), and furthermore, after renaming (i.e. applying $\text{RNM1}_{So'(s)}$), the list of influencers with which the candidate is anticipable should be the same for all such edges (see equation 3).

If an edge e has a procedure entry node $Si(e)$ as the sink, e is associated with call site $Si'(e)$ at its end. The set $succ(e)$ comprises of edges whose source is node $Si(e)$ and the set $succ'(e)$ comprises of edges which are associated with the call site $Si'(e)$ at their start. Note that even if the candidate is anticipable at the beginning of all the edges s' ($s' \in succ'(e)$) and none of the influencers is modified inside the procedure, the candidate may not be anticipable inside the procedure. This can happen for two reasons, all influencers of the candidate may not be visible inside the procedure, or the procedure may be invoked at multiple call sites and the candidate may not be anticipable at other call sites.

For maintaining the accuracy of our analysis, we consider two different cases. Let the call site $Si'(e)$ call procedure p. If the candidate's original occurrence is inside the invocation of the procedure p (or any procedure invoked by the procedure p), then the value of ANTOUT(e) is determined by ANTOUT(s), for $s \in succ(e)$. Also, note that if the procedure p is called at more than one call site (i.e. it is called at call site(s) besides $Si'(e)$), we need to consider different candidates corresponding to the different call sites at which p is called. So, if the candidate $\mathcal{C}$ can be moved outside procedure p, we denote it as the candidate $\mathcal{C}\_Si'(e)$(equation 4).

If the procedure p does not include the original occurrence of the candidate $\mathcal{C}$, then the value of ANTOUT(e) is determined by ANTOUT(s'), for $s' \in succ'(e)$(equation 3). The function

FSUM1 checks the required mod/ref condition on the procedure invoked at call site $Si'(e)$.

Equation 5 determines anticipability of a candidate at the beginning of an edge or block of code. For solving the equations 3 through 5, all unknowns are initialized with $\top$ and largest solution satisfying the equations is chosen.

Equations 6 and 7 determine how placement decisions are made using the anticipability terms. For an edge e, the term INS_BEG(e) means that placement of a candidate needs to be done at the beginning of the edge e. The term INS_IN means that placement is to be done inside edge e, and further intraprocedural analysis will be required for determining the exact placement. The equations can be explained as follows: Consider an block of code (edge) e, such that, for a particular candidate, ANTOUT(e) is true but ANTIN(e) is $\perp$. i.e. the candidate is anticipable at the end of this block of code but is not anticipable at the beginning of this block of code. Note that this can happen for two reasons: First, the block of code e may not be transparent with respect to this candidate or the candidate is not anticipable at the end of a block of code b, for $b \in cobeg(e)$. In either of these two cases, INS_IN(e) is marked true (see equation 6). If INS_IN(e) is true, this means that candidate will be placed inside the block of code e, however, further analysis (described in Section 4.6) will be required for determining the final placement.

Now, consider an edge e whose source is a procedure entry node, i.e. $So(e) \in \mathcal{E}$. Suppose that ANTIN(e) is not $\perp$. Consider any edge p, $p \in pred(e)$ and let ANTIN(p) be $\perp$. It can then be shown that ANTIN(p) will be $\perp$ for all the edges p, where $p \in pred(e)$. In this case, candidate is placed at the beginning of the procedure in which the block of code e exists. No further analysis is required for determining the placement.

Next, consider the case when the source of the edge e is a procedure return node i.e. $So(e) \in \mathcal{R}$. Consider an edge p such that $p \in pred(e)$. Even if ANTIN(e) is true and ANTOUT(p) is $\perp$, no placement may be required. This is because, ANTOUT(p') may be true for $p' \in pred'(e)$. It can be easily shown that ANTOUT(p') will be identical for all the edges p', for $p' \in pred'(e)$, and further, if ANTOUT(p) is $\perp$ for a $p \in pred(e)$, then ANTOUT(p) is $\perp$ for all $p \in pred(e)$. To determine the placement at the beginning of this edge e, we check if both $(\bigwedge_{p \in pred(e)} \neg \mathrm{ANTOUT}_{\mathcal{C}}(p))$ and $(\bigwedge_{p' \in pred'(e)} \neg \mathrm{ANTOUT}_{\mathcal{C}}(p'))$ return $\top$. If the condition is met, the candidate is placed just after the call statement at $So'(e)$. Again, no further analysis is required.

For the example program shown in Figure 1, the solution of data flow properties for determining placement of START_OP is shown in Figure 7.

4.5 Placement of END_OP

The analysis done for determining placement of END_OP is analogous to the analysis done for determining START_OP. Instead of using the anticipability terms, we use availability terms. The equations for propagating availability and for determining the placement is given in Figure 8.

Initially, the local data flow property COMP(i) of the edges in the graph is determined. (For a block of code, COMP means that there is an occurrence of this candidate inside the block which can be moved to the end of the block.) Let $\mathcal{C}$ be the candidate for placement. Consider

Edge	ANTOUT			ANTIN			INS_IN			INS_BEG		
	$\mathcal{C}$	$\mathcal{C}\_cs5$	$\mathcal{C}\_cs6$	$\mathcal{C}$	$\mathcal{C}\_cs5$	$\mathcal{C}\_cs6$	$\mathcal{C}$	$\mathcal{C}\_cs5$	$\mathcal{C}\_cs6$	$\mathcal{C}$	$\mathcal{C}\_cs5$	$\mathcal{C}\_cs6$
1	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
2	-	$< A, d >$	$\perp$	-	$< A, d >$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
3	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
4	-	$< x, y >$	$\perp$	-	$\perp$	$\perp$	-	$< x, y >$	$\perp$	-	$\perp$	$\perp$
5	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
6	-			-			-			-		
7	-	$< A, d >$	$\perp$	-	$\perp$	$\perp$	-	$< A, d >$	$\perp$	-	$\perp$	$\perp$
8	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
9	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
10	-	$\perp$	$< x, y >$	-	$\perp$	$\perp$	-	$\perp$	$< x, y >$	-	$\perp$	$\perp$
11	-	$< A, d >$	$< B, d >$	-	$< A, d >$	$< A, d >$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
12	$\perp$	-	-	$< x, y >$	-	-	$\perp$	-	-	$\perp$	-	-
13	-	$\perp$	$< B, d >$	-	$\perp$	$< B, d >$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
14	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$

Figure 7: Solution of Data Flow Properties for the Graph (for placement of START_OP)

any block of code (edge) i in which the original occurrence of the operation is placed. If the candidate is not enclosed in any conditional or loop, and none of its influencers are modified or referred to from its occurrence to the end of the block of code, then we mark

$$\mathrm{COMP}_{\mathcal{C}}(i) = Infl_c$$

Here, $Infl_c$ is the list of parameters of the call to the operation. Only if $\mathrm{COMP}_{\mathcal{C}}(i)$ is $Infl_c$ for all blocks of code i in which the original occurrence of this operation occurred, we can consider moving the END_OP after the existing location.

Two sets of functions FTRANS2 and FSUM2 summarize the mod/ref information, and are used for propagating data flow information. If we are considering placement of a write operation, $\mathrm{FTRANS2}_e[< Arrayname, s1, s2, .. >]$, for an edge e, returns the same list if $Arrayname$ is not modified in the block of code associated with this edge. Otherwise, this function returns $\perp$. If we are considering placement of a read operation, $\mathrm{FTRANS2}_e[< Arrayname, s1, s2, .. >]$ of an edge e returns the same list if $Arrayname$ is not referred to nor modified. $\mathrm{FTRANS2}_e[\top]$ and $\mathrm{FTRANS2}_e[\perp]$ are defined to be $\top$ and $\perp$ respectively.

For a call site cs which invokes procedure p, $\mathrm{FSUM2}_{cs}[< Arrayname, s1, s2, .. >]$ returns the same list if $Arrayname$ is not modified by the procedure p (or by a procedure invoked by p) (for a write operation). Otherwise $\perp$ is returned. For a read operation, $\mathrm{FSUM2}_{cs}[< Arrayname, s1, s2, .. >]$ checks if $Arrayname$ is not referred to or modified. $\mathrm{FSUM2}_{cs}[\top]$ returns $\top$ and $\mathrm{FSUM2}_{cs}[\perp]$ returns $\perp$.

Equations 8, 9 and 10 determine availability at the beginning and end of each block of code. Equations 11 and 12 determine final placement using the availability information. These equations are analogous to the equations for determining anticipability and computing placement of START_OP, and we do not explain them here.

$$\text{AVIN}_{\mathcal{C}}(e) \; = \; \begin{cases} \perp & \text{if } So(e) \text{ is BEGIN node} \\ \bigwedge_{p \in pred(e)} (\, \text{RNM1}_{Si'(p)} [\, \text{AVOUT}_{\mathcal{C}}(p) \,] \,) & \text{if } So(e) \in \mathcal{E} \\ \text{FSUM2}_{So'(e)}[\, \bigwedge_{p' \in pred'(e)} \text{AVOUT}_{\mathcal{C}}(p') \,] & \text{if } (So(e) \in \mathcal{R}) \wedge (\, \neg\, \text{OCR}_{\mathcal{C}}(So'(e)) \,) \end{cases} \quad (8)$$

$$\text{AVIN}_{\mathcal{C}\_So'(e)}(e) \; = \; \text{RNM2}_{So'(e)}[\, \bigwedge_{p \in pred(e)} \text{AVOUT}_{\mathcal{C}}(p) \,] \qquad \text{if } (So(e) \in \mathcal{R}) \wedge (\, \text{OCR}_{\mathcal{C}}\,(So'(e)) \,) \quad (9)$$

$$\text{AVOUT}_{\mathcal{C}}(e) \; = \; \begin{cases} \perp & \text{if } Si(e) \text{ is END node} \\ \text{COMP}_{\mathcal{C}}(e) \vee & \\ \quad (\, \bigwedge_{b \in coend(e)} \text{FTRANS2}_b[\text{AVIN}_{\mathcal{C}}(b)] \,) & \text{otherwise} \end{cases} \quad (10)$$

$$\text{INS\_IN}_{\mathcal{C}}(e) \; = \; \text{AVIN}_{\mathcal{C}}(e) \;\wedge\; (\neg \text{AVOUT}_{\mathcal{C}}(e)) \quad (11)$$

$$\text{INS\_END}_{\mathcal{C}}(e) \; = \; \begin{cases} \text{AVOUT}_{\mathcal{C}}(e) \wedge & \\ (\bigwedge_{s \in succ(e)} \neg \text{AVIN}_{\mathcal{C}\_So'(s)}(s)) & \text{if } (Si(e) \in \mathcal{R}) \wedge (\, \text{OCR}_{\mathcal{C}}(So'(s)) \,) \\ \text{AVOUT}_{\mathcal{C}}(e) \;\wedge\; (\bigwedge_{s \in succ(e)} \neg \text{AVIN}_{\mathcal{C}}(s)) & \\ \quad \wedge\, (\bigwedge_{s' \in succ'(e)} \neg \text{AVIN}_{\mathcal{C}}(s')) & \text{if } Si(e) \in \mathcal{E} \end{cases} \quad (12)$$

Figure 8: Data Flow Equations for Placement of END_OP

For the example program shown in Figure 1, the solution of data flow properties for determining placement of END_OP is shown in Figure 9.

4.6 Final Intraprocedural Analysis

As we stated earlier, intraprocedural analysis is required in the blocks of code where $\text{INS\_IN}_{\mathcal{C}}$ is marked to be true. We briefly sketch this analysis in this subsection. We describe our analysis in the context of placement of START_OP only; the analysis for the placement of END_OP is analogous.

Consider a procedure $\mathcal{P}$, which has at least one block of code e, such that INS_IN(e) is set to a list of influencers, $Infl$. Data flow analysis is done on the CFG of the procedure $\mathcal{P}$. At the end of the procedure and at all the call sites, initialization of properties is done using the values computed during the interprocedural analysis on the FPR of the full program.

For determining the final placement of START_OP, we compute anticipability at the beginning and end of each basic block in $\mathcal{P}$. For a basic block b, ANTICIN(b) denotes the anticipability of the candidate at the beginning of the basic block and ANTICOUT(b) denotes anticipability at the end of the basic block. If a block of code e ends at a call site cs, then and if b is the basic block just before this call site, then ANTICOUT(b) is set to be the same as ANTOUT(e). Similarly, if a block of code e ends at the return of procedure $\mathcal{P}$, then ANTICOUT(b) is set as ANTOUT(e), where b is the last basic block in the CFG of procedure $\mathcal{P}$.

After these initializations, the anticipability is propagated upwards in a simple fashion. For a basic block b, then ANTICOUT(b) is the meet of ANTICOUT(s), where s is any successor of b in the CFG. ANTICIN(b) is set to a list of influencers, only if ANTICIN(b) is the same list of

Edge	AVIN			AVOUT			INS_IN			INS_END		
	$\mathcal{C}$	$\mathcal{C}\_cs5$	$\mathcal{C}\_cs6$	$\mathcal{C}$	$\mathcal{C}\_cs5$	$\mathcal{C}\_cs6$	$\mathcal{C}$	$\mathcal{C}\_cs5$	$\mathcal{C}\_cs6$	$\mathcal{C}$	$\mathcal{C}\_cs5$	$\mathcal{C}\_cs6$
1	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
2	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
3	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
4	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
5	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
6	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
7	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
8	-	$< A, d >$	$< B, d >$	-	$\perp$	$\perp$	-	$< A, d >$	$< B, d >$	-	$\perp$	$\perp$
9	-	$< A, d >$	$< B, d >$	-	$\perp$	$\perp$	-	$< A, d >$	$< B, d >$	-	$\perp$	$\perp$
10	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
11	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
12	$\perp$	-	-	$< x, y >$	-	-	$\perp$	-	-	$\perp$	-	-
13	-	$< A, d >$	$\perp$	-	$< A, d >$	$\perp$	-	$\perp$	$\perp$	-	$\perp$	$\perp$
14	-	$< A, d >$	$< B, d >$	-	$< A, d >$	$< B, d >$	-	$< A, d >$	$< B, d >$	-	$\perp$	$\perp$

Figure 9: Solution of Data Flow Properties for the Graph (for placement of END_OP)

influencers and if the basic block is transparent with respect to this list of influencers. We do not give formal equations for intraprocedural analysis here.

The final placement inside procedure is decided as follows: If ANTICOUT(b) is a list of influencers and if ANTICIN(b) is $\perp$, then placement of the candidate is done after the last statement in the basic block b. Alternatively, let s be a successor of b in the CFG. If ANTICIN(s) is a list of influencers and if ANTICOUT(b) is $\perp$, then the final placement is done in a new basic block, inserted between b and s.

For the example program shown in Figure 1, the optimized program is shown in Figure 10.

5 Experimental Results

We have implemented a source to source Fortran translation tool based on our interprocedural balanced code placement technique. This tool is based upon the Parascope/D System front end. We have evaluated the efficacy of our technique by determining the performance improvement achieved by using this tool on two I/O intensive applications. The first application simulates a physical process and generates a snapshot to capture the evolution of the process. The second application is a template for satellite sensor data processing programs. These programs process large images that do not fit into the physical memory of a processor.

All our experiments were done on an IBM RS/6000 running AIX 3.2.5, with 64 MB of primary memory and one 2.2 GB IBM Starfire 7200 SCSI disk. The Starfire 7200 is rated at a maximum bandwidth of 8 MB/s; the maximum measured application-level file I/O bandwidth is 7.5 MB/s [1]. For all our experiments, we measured end-to-end execution time including a final `fsync()` to ensure that all data has been written to disk. This allowed us to include the cost of operating system operations. It also allowed us to measure end-to-end improvements in the execution time.

```
Program Foo                                    Procedure P(x,y)
    Arrays A, B                                    ..other computations ..
    d = ....                                   End
    Call Q(c)                                  Procedure Q(z)
    Do i = 1, 100                                  z = ...z...
        if cond then                           End
            Call R(A,d)                        Procedure R(x,y)
            Call Q(c)                              x = ...x...
        else                                       START_OP(x,y)
            A = ...                                ..other computations ..
            START_OP(A,d)                      End
        endif
        Call S(B,d)                            Procedure S(x,y)
        Call P(A,d)                                x = ...x...
        Call P(B,d)                                START_OP(x,y)
        END_OP(A,d)                                ..other computations ..
        END_OP(B,d)                            End
    Enddo
End
```

Figure 10: Optimized Version of Program

5.1 Direct Simulation Monte Carlo

Direction Simulation Monte Carlo is a well known technique for studying the interaction of particles in cells [5]. The program we used in our experiments, `dsmc-3d` was originally developed by Richard Wilmoth at NASA Langley [21]. To study the evolution of the process being simulated, it is useful to snapshot the position of all the particles after every time-step. This program is parameterized by the number of particles which governs the memory and I/O requirements of the program. It is a useful benchmark for our experiments as it allows us to study its behavior as the memory and I/O requirements varied.

The code input to the compiler used a synchronous write operation to generate a snapshot at the end of every time-step. Our compiler replaced this operation with appropriate asynchronous write operations. To determine the overall cost of I/O (both synchronous and asynchronous), we used a version of the program which did not generate snapshots.

Figure 11 presents results for four `dsmc-3d` data sets – with 388K, 486K, 584K and 682K particles (the output per iteration being 9.3 MB, 11.7 MB, 14.0 MB and 16.4 MB respectively). The memory requirements of the code (without I/O) were 34.5 MB, 40 MB, 45.5 MB and 51 MB respectively at these configurations.

Results are presented for the first five iterations. The third column shows the execution time of the code input to the compiler (with synchronous writes); the fourth column shows the execution time of the code generated by the compiler (with asynchronous writes). The fifth column shows the execution time of the version of the program that generates no snapshots.

No. of Particles.	Output/Iter. (MB)	With sync. output (sec)	With async. output (sec)	No output (sec)	Reduction I/O time	Reduction overall time
388K	9.3	33.78	31.35	30.19	67.7%	7.2%
486K	11.7	49.49	42.24	38.68	67.1%	15.6%
584K	14.0	85.28	68.36	49.62	47.5%	19.5%
682K	16.4	108.51	90.87	70.53	46.4%	16.8%

Figure 11: Performance of Compiler Placed Asynchronous I/O, Five iterations of `dsmc-3d`

The sixth and seventh columns show the percentage reduction in the total *time ascribed to I/O* and end-to-end execution time. The total *time ascribed to I/O* for either version of the program was computed by a difference of its execution time and the execution time of the no-snapshots version running on the same configuration.

For small configurations, both the memory requirements of the application and the size of output are small and fit together into the physical memory. In these cases, the write-behind facility provided by the operating system file cache works well and a synchronous write waits only as long as it takes to copy the data into the file cache. Use of asynchronous writes can reduce the I/O overhead by up to 60%, but the difference in the overall performance of the application is small.

For larger configurations, both the memory and I/O requirements higher. In these cases, a synchronous write operation has to wait till some pages are written to disk. In such cases, I/O can take up to 40% of the execution time of the program. As the results in Figure 11 show, the use of compiler placed asynchronous I/O can reduce the I/O waiting time by up to 67% and the end-to-end execution time of the program by up to nearly 20%.

Note that the percent reduction in the time ascribed to I/O drops as the number of particles simulated increases. We believe that this is because of the contention for the memory between the data pages and the file pages. If the data pages which are likely to be used in subsequent iterations are written back to the disk, the number of page faults increases. For the 682K data set (with synchronous I/O), the number of page faults resulting in I/O was 1401, whereas, for the 486K data set, this number was only 60. The contention for the main memory was, in our opinion, the main reason for the relatively low I/O rate observed for large configurations. For very large configurations, the competition between the data pages and the file pages leads to thrashing. For example, when the number of particles was increased to 731K, the execution time for the version with synchronous writes was 163 seconds, the time ascribed to I/O was 89 seconds. The use of asynchronous operation did not improve the performance as the time ascribed to I/O was dominated by paging activity.

Because of the structure of the code, no overlap in the I/O and computation would have been possible without interprocedural analysis. By hand analysis of the code, we determined that placement of the asynchronous operations could not have been any better that what was

Output/Iter. (MB)	With synch. write (sec)	With async. write (sec)	Reduction I/O time	Reduction overall time
6	114.8	98.9	40.8%	13.3%
8	123.9	106.1	37.1%	14.4%
10	130.2	111.0	35.3%	14.7%
12	147.9	128.2	27.4%	13.3%
14	173.5	148.6	25.2%	14.3%

Figure 12: Performance of Compiler Placed Asynchronous I/O, `satellite` template

achieved by our compiler. In all experiments with the compiler generated code, the time spent in `aio_complete()`[1] was close to zero. This indicates that the analysis was able to completely overlap the write with computation and the performance of snapshot generation for `dsmc-3d` cannot be further improved.

5.2 Template for Satellite Data Processing Programs

This Fortran template, which we shall refer to as `satellite`, was constructed to emulate the computational and I/O characteristics of a large number of satellite data processing programs. These programs take sensor data from a sequence of days and generate a single composite multi-band image of world. The sensor data is processed in chunks of about 500 KB. For composition, these programs maintain an out-of-core intermediate version of the image. After the data in each chunk is processed, each data value is mapped into the intermediate image and is compared with the corresponding pixel. If the new value is "better", it is copied into the pixel. To implement this *out-of-core max-reduction* efficiently, a bounding region of the image containing the pixels to be updated, is read, modified in-core and written back to disk.

This template was loosely based on `pathfinder`, the AVHRR program from the NASA Goddard Distributed Active Archive Center [1]. It has a similar organization, the same memory requirement and processes its input in same size (500 KB) chunks. In the template, the computation for every chunk is fixed at 7.6 seconds [2]; the size of the image region to updated by the read-modify-write operation is assumed to be the same for all chunks and is a parameter of the template.

The code input to the compiler used synchronous operations for all I/O – an input read at the beginning of the process-a-chunk loop as well as the read and write for the out-of-core max-reduction at the end of this loop. Our compiler replaced the synchronous write for the out-of-core computation with appropriate asynchronous write operations. The synchronous reads were not changed.

[1] The POSIX routine used to wait for completion of an asynchronous I/O operation.

[2] This number is derived from a set of `pathfinder` runs.

Figure 12 presents results for five `satellite` configurations with the region sizes of 6MB, 8MB, 10MB, 12MB and 14MB. Results are presented for the first ten chunks. The second column shows the execution time of the code input to the compiler (with synchronous writes); the third column shows the execution time of the code generated by the compiler (with asynchronous writes). The fourth and fifth columns show the percentage reduction in total time ascribed to I/O and the end to end execution time respectively. The total time ascribed to I/O for either version of the program was computed by subtracting 75.9 seconds (the execution time for the no-intermediate-IO version of the program) from its end-to-end execution time. The memory requirement of the program (without I/O) remained constant at nearly 45 MB for different configuration.

The improvement in end-to-end execution time is consistently around 14% for all configurations. The time ascribed to I/O grows roughly proportionally with the size of region to be updated. For the synchronous version, from 38.9 seconds for the 6 MB case to 97.6 seconds for the 14 MB case. On the other hand, the percentage reduction in the time ascribed to I/O drops as the size of region to be updated is increased.

Again, for this code, no overlap in the I/O and computation would have been possible without interprocedural analysis. Also, by hand analysis of the code, we determined that placement of the asynchronous operations could not have been any better that what was achieved by our compiler.

6 Discussion and Related Work

We now state the relationship of our work with the existing work in the areas of I/O, cache performance improvement, data flow analysis and interprocedural analysis. We also mention limitations of our current approach and the future directions we plan to take.

Several researchers have developed compiler techniques for overlapping cache stalls with computation (also known as software prefetching). This includes the work of Mowry *et al.* as part of the SUIF system [23] and Callahan *et al.* at Rice University [8]. The amount of time required for cache misses is usually of the order of 10 cycles; much smaller than the disk latency, which of the order of 100,000 cycles. Therefore, for cache prefetching one only needs to consider overlap within a single loop, as they do. In our case, we need to look at computation across procedure boundaries to allow for significant overlap.

Future architecture trends show that micro-processors will have large Level 2 (L2) caches, and the miss penalty for L2 caches will of the order of 500 cycles [28]. In such scenarios, it will be profitable to perform analysis across procedure boundaries to prefetch data into the L2 cache. We believe that our analysis can be extended to perform prefetches for L2 caches as well.

In separate work, we have worked on Interprocedural Partial Redundancy Elimination (IPRE) [2, 4] and other placement optimizations for distributed memory compilation [3]. The analysis required for balanced code placement is significantly different from the analysis in IPRE for at least two reasons: First, IBCP analysis needs to ensure that there is exactly one occurrence of asynchronous operation corresponding to each occurrence of the synchronous operation. IPRE, on the other hand, tries to reduce the number of occurrences of the candidates in the optimized code. Secondly, IPRE analysis does not need to consider placement of START_OP and END_OP separately. The program representation used in this paper was first introduced in the context of IPRE. Use of the same representation for IBCP framework establishes that this representation has wide applicability.

Hanxleden and Kennedy have developed a general framework for communication placement, which includes performing early placement of sends and late placement of receives [16]. This framework can also be used for placement of read and write operations, however, it is restricted to analysis and placement within a single procedure. Gornish *et al.* present methods for prefetching in shared memory multiprocessors [13]. Their techniques are also applicable for placing read operations early within a single procedure. However, their techniques apply only when the procedure has a simple loop structure. In contrast, our method does not impose any restrictions on the shapes of call graph and CFGs of the procedures. Several other projects have performed interprocedural optimizations for parallelism and for dealing with memory hierarchies. FIAT has been proposed as a general framework for interprocedural analysis [15], but largely targets flow-insensitive problems.

Compiler optimizations for improving I/O accesses have been addressed by at least two projects. The PASSION compiler (based upon Syracuse F90D system) performs loop transformations for improving locality in out-of-core applications [29]. Similar optimizations have also been performed as part of the Fortran D compilation system's support for out-of-core applications [26]. Neither of these groups have proposed any general techniques for placement of asynchronous operations or any interprocedural optimizations. Hand-compilation experiments have been presented to show performance gains from using asynchronous I/O [6].

Significant amount of work has been done on runtime libraries for optimizing Parallel I/O. PASSION library at Syracuse University is one such library for optimizing I/O accesses and uses the *two phase I/O* technique [7, 9]. Kotz has developed a similar technique, *disk-directed I/O* for performing collective I/O operations [18]. Other projects have focussed on benchmarking I/O intensive applications, including Crandall *et. al* at Illinois [11] and Acharya *et. al* at Maryland [1].

An important limitation of our current work has been to consider each array as a single entity, i.e., modification or reference to any element of an array is considered as mod/ref to the entire array. We plan to augment our analysis with array section analysis [17] to improve its accuracy. Consider, for example, a loop iterating over the elements of an array, which is to be snapshot later. In such a case, array section analysis will allow us to output the parts of the array which have been modified, without waiting for the other elements to be modified. One potential advantage of this kind of analysis will be to break up a large I/O operation into several smaller I/O operations, presumably of the size which will fit into the file caches. This can allow better performance even if the operating system or the I/O library does not allow asynchronous operations.

7 Conclusions

This paper has two main contributions. The first is a general interprocedural framework for replacing large latency synchronous applications with split-phase operations and overlapping them with computation. This scheme is applicable to arbitrary recursive procedures and arbitrary control flow within each procedure.

The second contribution of this paper is to evaluate the efficacy of this scheme for scientific applications which perform large and frequent write operations. We have implemented a Fortran source to source transformation tool which performs the analysis. Experiments with our tool on two applications has shown that large overlap of the write operations could be achieved through flow-sensitive interprocedural analysis. In both these applications, almost no overlap would have been possible if the analysis was restricted within single procedures.

We compared the performance of the version of the code performing synchronous write operations with the version performing compiler placed asynchronous operations. Use of compiler placed asynchronous operations could reduce the I/O overhead of these applications by 30%-70% and the overall performance of the code by up to 20%. Performance gains are most significant when large write operations are made by codes which are using a large fraction of the primary memory of the program.

Acknowledgements

We have implemented our source to source compiler using the Parascope/ D System Fortran front end. We gratefully acknowledge our debt to its implementers. We would like thank Bongki Moon for the `dsmc-3d` program. We would also like to thank Tonjua Hines and Steve Kempler from the Earth Science Data & Information Systems Project (ESDIS) at NASA Goddard Space Flight Center for invaluable discussions about NASA's satellite data processing requirements, and for helping us gain access to NASA programs.

References

[1] Anurag Acharya, Mustafa Uysal, Robert Bennett, Assaf Mendelson, Mike Beynon, Jeff Hollingsworth, Joel Saltz, and Alan Sussman. Tuning the Performance of I/O Intensive Parallel Applications. Submitted to IOPADS'96, October 1995.

[2] Gagan Agrawal and Joel Saltz. Interprocedural communication optimizations for distributed memory compilation. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, pages 283–299, August 1994. Also available as University of Maryland Technical Report CS-TR-3264.

[3] Gagan Agrawal and Joel Saltz. Interprocedural compilation of irregular applications for distributed memory machines. In *Proceedings Supercomputing '95*. IEEE Computer Society Press, December 1995. To appear. Also available as University of Maryland Technical Report CS-TR-3447.

[4] Gagan Agrawal, Joel Saltz, and Raja Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 258–269. ACM Press, June 1995. ACM SIGPLAN Notices, Vol. 30, No. 6. Also available as University of Maryland Technical Report CS-TR-3446 and UMIACS-TR-95-42.

[5] Graeme A. Bird. *Molecular Gas Dynamics and the Direct Simulation of Gas Flows.* Clarendon Press, Oxford, 1994.

[6] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[7] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings Supercomputing '93*, pages 452–461. IEEE Computer Society Press, November 1993.

[8] D. Callahan, Ken Kennedy, and A. Porterfield. Software prefetching. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991.

[9] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: Parallel and scalable software for input-output. Technical Report SCCS-636, NPAC, September 1994. Also available as CRPC Report CRPC-TR94483.

[10] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988.

[11] P. E. Crandall, R. A. Aydt, A. C. Chien, and D. A. Reed. Input/Output characteristics of Scalable Parallel Applications. In *Proceedings Supercomputing '95*, December 1995. To appear.

[12] N. Galbreath, W. Gropp, and D. Levine. Applications-driven Parallel I/O. In *Proceedings Supercomputing '93*, pages 462–471, November 1993.

[13] E. H. Gornish, E. D. Granston, and A. V. Veindenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of International Conference on SuperComputing*, pages 354–368, July 1990.

[14] Mary Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, October 1990.

[15] Mary Hall, John Mellor Crummey, Alan Carle, and Rene G Rodriguez. FIAT: A framework for interprocedural analysis and transformations. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 522–545. Springer-Verlag, August 1993.

[16] Reinhard von Hanxleden and Ken Kennedy. Give-n-take – a balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 107–120. ACM Press, June 1994. ACM SIGPLAN Notices, Vol. 29, No. 6.

[17] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[18] David Kotz. Disk-directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Department of Computer Science, Dartmouth College, July 1994.

[19] Kwan-Liu Ma and Z.C. Zheng. 3D visualization of unsteady 2D airplane wake vortices. In *Proceedings of Visualization'94*, pages 124–31, Oct 1994.

[20] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[21] B. Moon and J. Saltz. Adaptive runtime support for direct simulation Monte Carlo methods on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 176–183. IEEE Computer Society Press, May 1994.

[22] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[23] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73. ACM Press, October 1992.

[24] E. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth ACM Symposium on the Principles of Programming Languages*, pages 219–230, January 1981.

[25] R. Nance, R. Wilmoth, B. Moon, H. Hassan, and J. Saltz. Parallel DSMC solution of three-dimensional flow over a finite flat plate. In *Proceedings of the 6th AIAA/ASME Joint Thermophysics and Heat Transfer Conference*, Colorado Springs, CO, June 1994.

[26] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118. IEEE Computer Society Press, February 1995.

[27] G. Patnaik, K. Kailasnath, and E.S. Oran. Effect of gravity on flame instabilities in premixed gases. *AIAA Journal*, 29(12):2141–8, Dec 1991.

[28] Mendel Rosenblum, Ed Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *Proceedings of Symposium on Operating System Principles*, 1995. To appear.

[29] Rajeev Thakur, Rajesh Bordawekar, and Alok Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. In *Proceedings of the IPPS'94 Second Annual Workshop on Input/Output in Parallel Computer Systems*, pages 54–72, April 1994. Also appears in ACM Computer Architecture News, Vol. 22, No. 4, September 1994.

[30] R.G. Wilmoth. Application of a parallel direct simulation monte carlo method to hypersonic rarefied flows. *AIAA Journal*, 30(10):2447–52, Oct 1992.

[31] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44. ACM Press, June 1991.

[32] Michael Wolfe. Data dependence and program restructuring. *Journal of Supercomputing*, 4(4):321–344, January 1991.