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 the applications which perform frequent and large accesses to secondary memory, including the applications which snapshot or checkpoint their computations or the out-of-core applications.

1 Introduction

Modern architectures have deep memory hierarchies. 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 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 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 (*snapshoting*), 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.

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 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

*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.

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 state the requirements for our code placement framework and motivate our analysis. In Section 3, we present our interprocedural analysis. We present experimental results in Section 4. We conclude in Section 5.

2 Problem Definition

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.

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:

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 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) [3] can be applied before applying the analysis we present in this paper.

Our method is broadly applicable for the placement of both the read and write operations. However, performing an early start of a read operation involves additional memory usage. In such cases, our analysis needs to be used in conjunction with additional analysis for estimating resource constraints. This is especially important for the read operations in the out-of-core applications, which are constrained by the limits of the primary memory [5].

We are not aware of any other work on overlapping operations with computation interprocedurally. In previous work, Hanxleden and Kennedy have developed a general framework for placement of asynchronous operations and redundancy elimination [6], but this framework is restricted to performing analysis within a single procedure. Compiler optimizations for improving I/O accesses have been addressed by at least two projects. PASSION compiler (based upon Syracuse F90D system) performs loop transformations for improving locality in out-of-core applications [5]. Similar optimizations have also been performed as part of the Fortran D compilation system's support for out-of-core applications [8]. None of these efforts presented any general techniques for placement of asynchronous operations or any interprocedural optimizations.

3 Interprocedural Balanced Code Placement Framework

3.1 Program Representation

We have developed a new full program representation which extends the call graph by preserving information about the call sites but is more concise than the Myers's SuperGraph [7]. Full details of this program representation are available elsewhere [2, 3].

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.

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 between the nodes depending upon the flow of control. Corressponding to each such edge, there is a *block of code*. A block of code comprises of basic blocks which may be visited during flow of control from the source to the sink of the corressponding edge.

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

3.2 Candidates for Placement

For the purpose of our discussion, we refer to any synchronous operation, which the compiler may want to

```
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

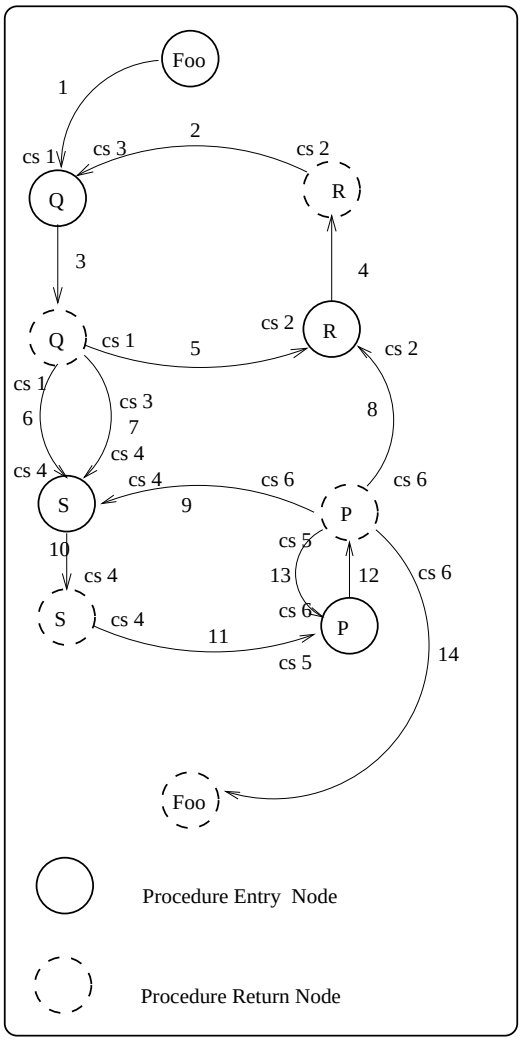

Figure 2: *FPR* for program in the left. Edge numbers and call sites at which edges start/end (whenever applicable) are marked in the Figure.

replace by asynchronous operation, as a *candidate* for placement. We refer to the parameters of a candidate as its *influencers*.

A candidate (read or write operation), for our purposes, has the format

$$\text{OP}(s1, s2, ...)$$

where, $s1, s2, ...$, are various parameters which specify the array or the buffer to be output/input and the sizes/sections of the array which are to be input/output.

3.3 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 $\text{ANTOUT}_{\mathcal{C}}(e)$.

In this paper, we will only present the details of the analysis for the placement of START_OP (Section 3.4). Analysis for the placement of END_OP is analogous and the details are presented elsewhere [2].

We now list the two 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 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. 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. Such paths are also referred to as *invalid* or *unrealizable* paths.

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.

3.3.1 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 top-most element is denoted by $\top$ and the botton element is denoted by $\bot$. 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 $\bot$ when applied to a list of influencers or to $\top$ and returns $\top$ when applied to $\bot$.

3.3.2 Terminology

We use the following terms to specify the data flow equations in this paper. We had defined our program representation earlier in Section 3.1. 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 $cs_end(e)$. We use $succ'(e)$ to refer to the set of edges which have the call site $cs_end(e)$ 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 $cs_beg(e)$. We refer by $pred'(e)$ the set of edges which are associated with the call site $cs_beg(e)$ 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 3.

3.4 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 4. 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

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

Here, $Infl_c$ is the list of parameters of the call to the operation. Only if $\text{ANTLOC}_{\mathcal{C}}(i)$ is $Infl_c$ for

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

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 functions, BIND and BIND^{-1} rename the influencers at the call sites. BIND converts the formal parameters into actual parameters and BIND^{-1} converts the actual parameters into formal parameters.

Two sets of functions TRANS and SUM summarize the mod/ref information, and are used for propagating data flow information. If we are considering early placement of a write operation, TRANS[$< s1, s2, .. >$], 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 we are considering early placement of a read operation, TRANS[$< s1, s2, .. >$] for an edge e, returns the same list if none of the influencers are referred to or modified.

For a procedure call p, SUM[$< s1, s2, .. >$] 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). For a read operation, SUM[$< s1, s2, .. >$] checks if none of influencers are referred to or modified.

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 $\mathcal{C}$. 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.

The equations for propagation of anticipability (Figure 4) can be explained as follows. Consider 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 BIND^{-1} function), the list of influencers with which the candidate is anticipable should be the same for all such edges (see equation 1).

If an edge e has a procedure entry node $Si(e)$ as the sink, e is associated with call site $cs_end(e)$ 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 $cs_end(e)$ 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 $cs_end(e)$ 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 $cs_end(e)$), 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 $\mathcal{C}_(cs_end(e))$(equation 2).

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 1). The function SUM checks the required mod/ref condition on the procedure invoked at call site $cs_end(e)$.

Equations 4 and 5 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. This intraprocedural analysis is done on

$$\mathrm{ANTOUT}_{\mathcal{C}}(e) = \begin{cases} \bot & \text{if } Si(e) \text{ is END node} \\ \bigwedge_{s \in succ(e)} (\,\mathrm{BIND}^{-1}[\,\mathrm{ANTIN}_{\mathcal{C}}(s)]\,) & \text{if } Si(e) \in \mathcal{R} \\ \mathrm{SUM}[\bigwedge_{s' \in succ'(e)} \mathrm{ANTIN}_{\mathcal{C}}(s')\,] & \text{if } (Si(e) \in \mathcal{E}) \wedge (\neg \mathrm{OCR}_{\mathcal{C}}(cs_end(e))) \end{cases} \quad (1)$$

$$\mathrm{ANTOUT}_{\mathcal{C}_(cs_end(e))}(e) = \mathrm{BIND}[\,\bigwedge_{s \in succ(e)} \mathrm{ANTIN}_{\mathcal{C}}(s)\,] \quad \text{if } (Si(e) \in \mathcal{E}) \wedge (\,\mathrm{OCR}_{\mathcal{C}}(cs_end(e))) \quad (2)$$

$$\mathrm{ANTIN}_{\mathcal{C}}(e) = \begin{cases} \bot & \text{if } So(e) \text{ is BEGIN node} \\ \mathrm{ANTLOC}_{\mathcal{C}}(e) \vee & \\ \quad (\bigwedge_{b \in cobeg(e)} \mathrm{TRANS}_b[\mathrm{ANTOUT}_{\mathcal{C}}(b)]) & \text{otherwise} \end{cases} \quad (3)$$

$$\mathrm{INS_IN}_{\mathcal{C}}(e) = \mathrm{ANTOUT}_{\mathcal{C}}(e) \wedge (\neg \mathrm{ANTIN}_{\mathcal{C}}(e)) \quad (4)$$

$$\mathrm{INS_BEG}_{\mathcal{C}}(e) = \begin{cases} \mathrm{ANTIN}_{\mathcal{C}}(e) \wedge & \\ (\bigwedge_{p \in pred(e)} \neg \mathrm{ANTOUT}_{\mathcal{C}_(cs_end(p))}(p)) & \text{if } (So(e) \in \mathcal{E}) \wedge (\,\mathrm{OCR}_{\mathcal{C}}(cs_end(p))) \\ \mathrm{ANTIN}_{\mathcal{C}}(e) \wedge (\bigwedge_{p \in pred(e)} \neg \mathrm{ANTOUT}_{\mathcal{C}}(p)) & \\ \quad \wedge (\bigwedge_{p' \in pred'(e)} \neg \mathrm{ANTOUT}_{\mathcal{C}}(p')) & \text{if } So(e) \in \mathcal{R} \end{cases} \quad (5)$$

Figure 4: Data Flow Equations for Placement of START_OP

the Control Flow Graph (CFG) of a procedure, and is analogous to the interprocedural analysis presented here. The details of this analysis are omitted from this paper [2].

4 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 periodically 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.

4.1 Direct Simulation Monte Carlo

Direction Simulation Monte Carlo is a well known technique for studying the interaction of particles in cells [4]. The program we used in our experiments, `dsmc-3d`, was originally developed by Richard Wilmoth at NASA Langley. 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.

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.

Figure 5 presents results for four `dsmc-3d` data sets – with 388K, 486K, 584K and 682K particles (the output per iteration is 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 for 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. 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 appli-

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

cation 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 5 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 primary 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 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.

4.2 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.

Figure 6 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

[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.

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

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.

5 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.

Our experimental results have been restricted to single processor applications only. However, experiences with I/O on parallel applications have shown that accesses to local disks are very important [1]. Our analysis is directly applicable to performing placement of accesses to local disk on parallel applications.

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. To appear in IOPADS'96, May 1996.

[2] Gagan Agrawal, Anurag Acharya, and Joel Saltz. An interprocedural framework for placement of asynchronous i/o operations. Technical Report CS-TR-3563, UMD, December 1995.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.