# A Software Architecture for Massively Parallel Input-Output

Peter Brezany[†] Thomas Mück[‡], and Erich Schikuta[‡]

Technical Report No. TR 96-3
Institute for Software Technology and Parallel Systems, University of Vienna.

[†]Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstr. 22
A-1092 Vienna, Austria
Email: brezany@par.univie.ac.at

[‡]Department of Data Engineering
University of Vienna
Rathausstr. 19/4
A-1010 Vienna, Austria
Email: [mueck, schiki]@ifs.univie.ac.at

*Abstract*— **For an increasing number of data intensive scientific applications, parallel I/O concepts are a major performance issue. Tackling this issue, we provide an outline of an input/output system designed for highly efficient, scalable and conveniently usable parallel I/O on distributed memory systems. The main focus of this paper is the parallel I/O runtime system support provided for software-generated programs produced by High Performance FORTRAN compilers. Specifically, our design is presented in the context of the Vienna Fortran Compilation System.**

Contents



## I. Introduction

The main issue for I/O subsystems in supercomputing environments is to feed arrays of computing processors with huge amounts of raw data (nowadays typically in the Tbyte range) in such a way that neither the processor execution time nor the program turnaround time increase significantly.

On the hardware side, disk arrays (RAID), HIPPI interfaces, data vaults, etc. are the enabling technology approaches towards high-performance Input/Output systems. However, on the software side, current I/O subsystems suffer from a lack of a high-level programming support. This issue (including language, compiler and runtime system aspects) is currently a hot research topic at a variety of scientific and industrial institutions.

In this paper, we investigate the I/O problem from a runtime system support perspective. We focus on the design of an advanced parallel I/O support, called VIPIOS (VIenna Parallel I/O System), to be targeted by language compilers supporting the same programming model like High Performance Fortran (HPF) [9]. The VIPIOS design is partly influenced by the concepts of parallel database technology.

Section II introduces a formal model, which forms a theoretical framework on which the VIPIOS design is based. Here, we assume that the reader is familiar with the basic concepts of HPF. The VIPIOS architecture and the interface to a particular HPF compilation system, i.e., the Vienna Fortran Compilation System [2], are addressed in Section III which is the core of the paper. Section IV discusses related work and Section V concludes with a brief summary.

## II. Data Mapping Model

The model introduced in this section describes the mapping of the problem specific data space starting from the application program data structures down to

The work described in this paper is being carried out as part of the research project "Language, Compiler and Advanced Data Structure Support for Parallel I/O Operations" supported by the Austrian Research Foundation (FWF Grant P11006-MAT).

the physical layout on disk across several intermediate representation levels.
Parallelization of a program $\mathcal{Q}$ is guided by a mapping of the *data space* of $\mathcal{Q}$ to the processors and disks. The data space $\mathcal{A}$ is the set of declared arrays of $\mathcal{Q}$.

Definition 1 *Index Domain*
*Let $A \in \mathcal{A}$ denote an arbitrary array. The* index domain *of A is denoted by $\mathsf{I}^A$. The* extent *of A, i.e. the number of array elements is denoted by $ext^A$.*

The index domain of an array is usually determined through its declaration.
In a lot of languages, only array elements or whole arrays may be referenced. On the other hand, some languages like Fortran 90 provide features for computing subarrays. A subarray can be either characterized as an array element or an array section.

Definition 2 *Index Domain of an Array Section*[1]
*Let us consider an array A of rank n with index domain $\mathsf{I}^A = [l_1 : u_1, \dots, l_n : u_n]$ where the form $[l_i : u_i]$ denotes the sequence of numbers $(l_i, l_i+1, ..., u_i)$. The index domain of an array section $A' = A(ss_1, \dots, ss_n)$ is of rank $n'$, $1 \leq n' \leq n$, iff $n'$ subscripts $ss_i$ are section subscripts (subscript triplet or vector subscript) and the remaining subscripts are scalars. For example the index domain of the Fortran 90 array section $A(11:100:2,3)$ is given by $[1:45]$.*
*The* extended index domain *of the array section $A'$, denoted by $\bar{\mathsf{I}}^{A'}$, is given by $\bar{\mathsf{I}}^{A'} = [ss'_1, \dots, ss'_n]$ where $ss'_i = ss_i$, if $ss_i$ is a section subscript, and $ss'_i := [c_i : c_i]$ if $ss_i$ is a scalar subscript that is denoted by $c_i$. For example the extended index domain of the array section A(11:100:2,3) is given by [11:100:2,3:3].*

### A. Mapping Data Arrays to Computing Processors

Data distribution is modelled as a mapping of array elements to (non-empty) subsets of processors. The notational conventions introduced for arrays above can also be used for *processor arrays*, i.e., $\mathsf{I}^H$ denotes the index domain of a processor array $H$.
A *distribution* of an array maps each array element to one or more processors which become the *owners* of the element, and, in this capacity, store the element in their local memory. Distributions are modelled by functions between the associated index domains.

Definition 3 *Distributions*
*Let $A \in \mathcal{A}$, and assume that H is a processor array. A distribution of the array A with respect to H is defined by the the mapping:*

$$\mu_H^A : \mathsf{I}^A \rightarrow \mathcal{P}(\mathsf{I}^H) - \{\phi\}, \text{ where } \mathcal{P}(\mathsf{I}^H) \text{ denotes the power set of } \mathsf{I}^H$$

### B. Logical storage model

Definition 4 *File*
*A file $F_L$ consists of a sequence of logically connected information units, which we call records r.*

Definition 5 *Record*
*A record r consists of a fixed number m of typed data values (attributes).*

$$r = (A_1, A_2, ...A_m)$$

One record can store, for example, one float number or a complete data structure.

Definition 6 *File View*
*The sequence of data elements which results from a write operation of an array A to a file $F_L$ and is assumed by subsequent read operations is described by the function:*

$$\alpha^A : \mathsf{I}^A \rightarrow [1 : ext^A]; \text{ (the standard Fortran record ordering may be applied)}$$

### C. Physical storage model

Definition 7 *Disk*
*A disk d is the physical storage medium of interest. It is organized into storage units of fixed size (buckets). Let*

$$D = \{d_1, ..., d_N\}$$

*denote the set of all available disks.*

Definition 8 *Bucket*
*A bucket is a physical, contiguous sequence of bytes of a fixed size. It is the smallest unit, which the operating system of the underlying machine can transfer at once between the external storage medium and the main memory of a processor. Let $B_j$ denote the set of all buckets of a disk $d_j$. The set B of all buckets available for storing users' data is defined by the union of all buckets $B_j$ of all disks. Therefore,*

$$B = \bigcup B_j, \text{ where } 1 \leq j \leq |D| \text{ and } |D| \text{ denotes the number of disks}$$

Definition 9 *Physical file*
*A physical file $F_P$ is defined by a fixed number of physical blocks or buckets, which are stored on the external storage medium.*

Definition 10 *Declustering function*
*A declustering function $\delta$ defines the record distribu-*

[1] We have borrowed this definition from [1].

*tion scheme for D, according to one or more attributes and/or any other distribution criteria,*

$$\delta : R \to D, \text{ where } R \text{ is the set of records and } D \text{ the set of disks.}$$

*In other words, the declustering function $\delta$ partitions the set of records R into disjoint subsets $R_1$, $R_2$, ... $R_{|D|}$, which are assigned to disks.*

Definition 11 *Organization function*
*The organization function $o_d$ gives the buckets $B_d$ of the disk d, where a specific record is stored.*

$$o_d : R \to B_d, \text{ where } B_d \text{ are the buckets of disk } d$$

*Each disk has its own organization function. Therefore a family of organization functions exists, i.e. $o_1, o_2, ...o_{|D|}$.*

Definition 12 *Array area*
*An array area $E^A$ denotes the set of file records that store the elements of an array A or array section A'.*

It is now possible to redefine the declustering and organization functions for array areas.

Definition 13 *Areal declustering function*
*An areal declustering function $\Delta$ defines the set of disk drives D, which is spanned by a given area $E^A$. In other words, with $\mathcal{E}$ denoting the set of all possible areas of an array A*

$$\Delta : \mathcal{E} \to \mathcal{P}(D) - \{\phi\}$$

Definition 14 *Areal organization function*
*The areal organization function $O_d$ defines the set of buckets of disk d, where the records covered by the given area $E^A$ are stored, i.e.*

$$O_d : \mathcal{E} \to \mathcal{P}(B_d) - \{\phi\}, \text{ where } B_d \text{ is the set of buckets of disk } d$$

*Every disk has its own area organization function. Therefore a family of area organization functions exists, i.e. $O_1$, $O_2$, ... $O_{|D|}$.*

Definition 15 *Query mapping function*
*The query mapping function q gives the set of buckets B over all disks defined by a given area, i.e.*

$$q : \mathcal{E} \to \mathcal{P}(B), \text{ where } B \text{ is the union of buckets over all disks}$$

*The query mapping function can be defined by a combination of the areal declustering $\Delta$ and the areal organization function O, i.e.*

$$q(E^A) = \bigcup O_i(E^A), \text{ where } i \in \Delta(E^A)$$

## III. Design of the Parallel I/O Runtime Support

The I/O runtime system VIPIOS introduced in this section provides support for parallel access to files for read/write operations, optimization of data-layout on disks, redistribution of data stored on disks, communication of out-of-core[2] (OOC) data and many optimizations including data prefetching from disks based on the access pattern knowledge extracted from the program by the compiler or provided by a user specification. In the first implementation, VIPIOS will provide a mass storage support for VFCS [2].

### A. Design Objectives

The goals of VIPIOS can be summarize as follows.

1. Efficiency. The objective of runtime optimizations is to minimize the disk access costs for file I/O and OOC processing. This is achieved by a suitable data layout on disks and data caching policies. Specifically, VIPIOS aims at optimizing data locality, storing data as close as possible to the processing node, and efficient software-controlled prefetching to partially hide memory latency by explicit (i.e., runtime system driven) execution of disk and/or network prefetch operations. Prefetching moves data (e.g., file blocks, OOC array sections) as close as possible to the processor which is going to need the data.
2. Parallelism. All file data and meta-data (description of files) are stored in a distributed and parallel form across multiple I/O devices. The user and the compilation system have the ability, in the form of hints, to influence the distribution of the file data.
3. Scalability. The architecture of VIPIOS is inherently scalable. Control of the whole system is decentralized.
4. Portability. VIPIOS is able to provide support for any HPF compilation system that supplies all the interface information in the required format. In order to interface a large class of computer systems, VIPIOS strongly relies on standards, like MPI [12] and MPI-IO [8].

### B. Coupling VIPIOS to VFCS

The VFCS overall architecture including VIPIOS is depicted in Fig. 1. VIPIOS is coupled to the modules

[2]Mass storage I/O is also necessary to handle accesses to huge arrays whose parts must be kept on disks due to main memory constraints.

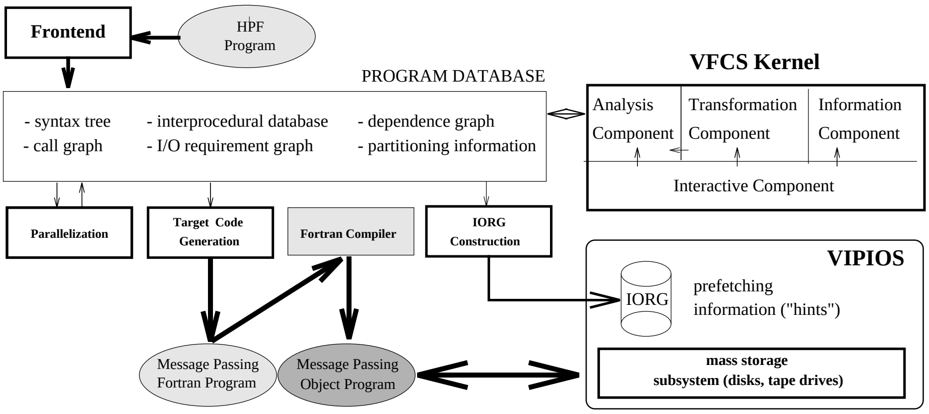


Fig. 1. VIPIOS Operational in a VFCS Environment

of VFCS that deal with I/O. The source program is processed by the *frontend* and transformed into an internal representation (IR) which is kept in the *program database*. The frontend is able to process HPF programs (as depicted in the figure) and Vienna Fortran [15] programs as well. All transformations are performed on IR. The parallelization process can conceptually be divided into four steps: (1) data distribution, (2) work distribution, (3) communication generation, and (4) insertion of I/O operations. Based on extensive analyses powerful optimizations are performed to reduce communication and I/O costs. VFCS produces explicitly parallel Fortran SPMD program with calls to the selected message-passing library (e.g., MPI), parallel I/O library (e.g., VIPIOS) and other components of the runtime system. This program is finally translated by the *Fortran compiler* into the object program. Due to the interactive component, the user may receive feedback from the system, supply information to the system, and select transformation strategies.

The input program is annotated with respect to I/O requirements. In [4], [5], we proposed language constructs to specify OOC structures and I/O operations for distributed data structures. These constructs can be used by the programmer to provide information helping VFCS and VIPIOS to operate the underlying I/O subsystem in an efficient way. Information selected from the annotation and static information extracted from the input program by VFCS at compile time is passed to VIPIOS through VIPIOS procedure calls from the target SPMD program. Placement of these calls is the result of program optimization. Program analysis and performance prediction phases extend the IR with information used for optimizing transformations and construction of an I/O requirement graph (IORG) which is used for guiding the prefetching process at runtime. Section III-F describes the structure of the IORG. This graph is incrementally constructed in the program database of VFCS during the compilation process and written to a file at its end. The IORG is passed to VIPIOS as an information base for prefetching heuristics. The generated SPMD program passes to VIPIOS the actual requirements on array section transfer as well as additional runtime data replacing symbolic information in the IORG (e.g., actual values for variable names). ¿From the VFCS point of view, VIPIOS can be considered as an *abstract mass storage I/O machine.* Its interaction with the compilation and runtime environment is depicted in Fig. 2 where the term *application process* denotes a running instance of the SPMD program. Some of the heuristics which VIPIOS applies for runtime optimizations are discussed later.

## *C. Process Model*

The framework VIPIOS distinguishes between two types of processes: application processes and VIPIOS servers. The application processes are created by the VFCS according to the SPMD paradigm in which each processor is executing the same program on different parts of the data space. The VIPIOS servers run independently on all, a number or dedicated nodes and perform the data and other requests of the application processes. The number and the location of the VIPIOS servers are dependent on the underlying hardware architecture (disk arrays, local disks, specialized I/O nodes, etc.), the system configuration (number of available nodes, types of available nodes, etc.), the VIPIOS system administration (number of serviced nodes, disks, application processes, etc.) or (simply) the user needs (I/O characteristics, regular, irregular problems, etc.).

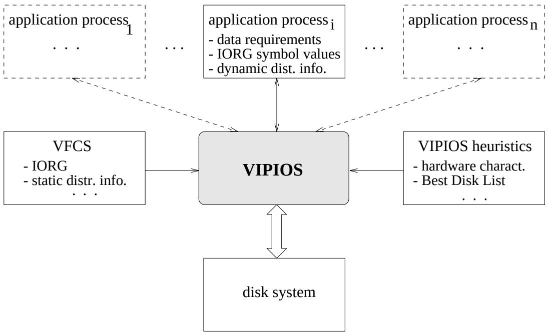


Fig. 2. VIPIOS Viewed as Abstract Mass Storage I/O Machine

These VIPIOS servers are similar to data server processes in database systems. For each application process one unique VIPIOS server is assigned and accomplishes the data requests. It is also possible that one VIPIOS server services a number of application processes. In other words one-to-one or one-to-many relationships are supported.

The process model is depicted by Fig. 3. The VIPIOS call interface VI, which is linked with the application process AP, handels the communication with the assigned VIPIOS server VS.

In this figure, different architectural models are summarized. Both local disk architectures, as disk *D2* and *D3* (nodes 2 and 3), and global disk systems as disk *D1* are supported (the surrounding frame depicts a physical processing node). Further, it can be seen that varying process models are feasible. On the one hand application process *AP3* on node 3 is serviced by VIPIOS server *VS2* running on the same node, which in turn administrates the local disk *D3* and the remote disk *D2* (local to node 2). On the other hand the server process *VS1* handles the requests of the local application process *AP2* and accesses the global disk only. The connection between the 2 servers illustrates that both can request each other in accessing remote data.

For each application process, all data requests are transparently caught by the assigned VIPIOS processes. The transfer of locally or remotely retrieved data is organized by these processes, and they ensure that each application process has access to its requested data items.

The VFCS provides information about the problem specific data distribution, the stride size of the slabs of the out-of-core data structures and the presumed data access profile. Based on this information, the VIPIOS organizes the data and tries to ensure high performance access to stored data. Additional data distribution and usage information can be provided by the Vienna Fortran programmer using new language constructs. This type of information allows the VFCS/VIPIOS system to parallelize read and write operations for out-of-core arrays by selecting a well-suited data organization in the files.

An important advantage of the proposed framework is the transparence of the architecture to the application programmer as well as to the VFCS compiler developer.

Summing up, the VIPIOS, as a component of the VFCS, is responsible for the organization and maintenance of all data held on the mass storage devices.

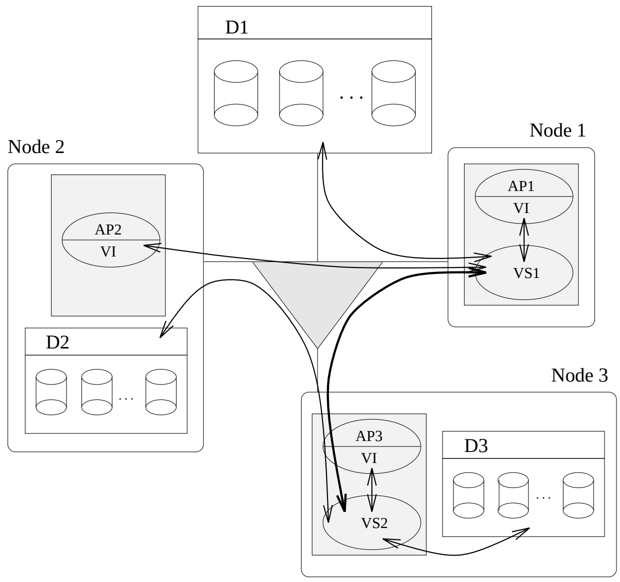


Fig. 3. Process Model: Application Processes and VIPIOS Servers on top of a Disk Architecture with Local and Global Disk Systems

## D. Data Locality

The main design principle of the VIPIOS to achieve high data access performance is *data locality.* This means that the data requested by an application process should be read/written from/to the best-suited disk.

We distinguish between logical and physical data locality.

*Logical data locality* denotes to choose the best suited VIPIOS server for an application process. This server is defined by the topological distance and/or the process characteristics. In most cases the access time is proportional to the topological distance of the application process to the VIPIOS server in the system net-

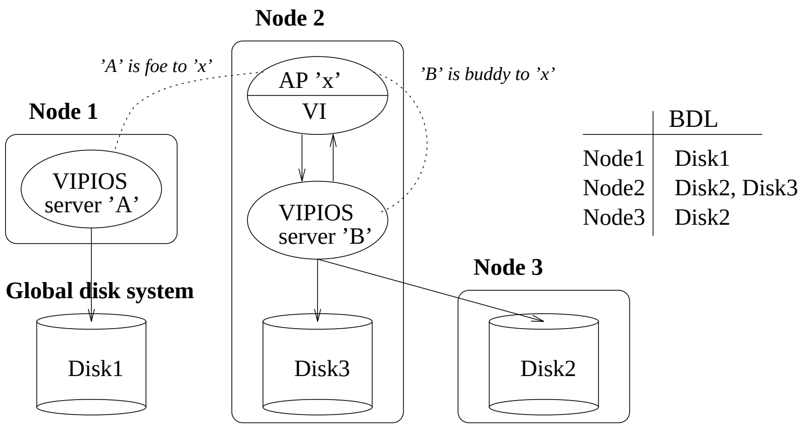


Fig. 4. "Buddy" and "Foe" Relation Between Application Processes and VIPIOS Servers

work. It is also possible that special process characteristics can influence the VIPIOS server performance, like available memory, disk priority list of the underlying node (see the following), etc. Therefore it is also possible that a remote VIPIOS server could provide better performance than a closer one. However only one specific VIPIOS server is chosen for each application process, which handles the respective requests. This process is called the *buddy server*, while all other servers are called *foe servers*. The buddy and foe relation is illustrated by Fig. 4.

The *physical data locality* principle aims to determine the disk set providing the best (mostly the fastest) data access. For each node an ordered sequence of the accessible disks of the system is defined (the *best disk list, BDL*), which orders the disk according to their access behavior. Disks with good access characteristics precede disks with bad access characteristics in this list (Fig. 4).

### *E. Two-Phase Data Administration Process*

The data administration process of a VIPIOS servers can be divided into two phases, the preparation and the administration phase. The *preparation phase* prepares the administrated data according to the data layout of the data structure, the presumed access profile and the physical restrictions of the system (available main memory, disk space, etc.). The input data for this phase are partly prepared in the program compilation process. The phase is performed during the system startup time and precedes the execution of the application processes. In this phase, the physical data layout schemas are defined, the actual VIPIOS server process for each application process and the disks for the stored data according to the locality principles are chosen. Further, the data storage areas are prepared, the necessary main memory buffers allocated, etc.

The *administration phase* accomplishes the I/O requests of the application processes. It is obvious that the preparation phase is the basis for good I/O performance.

### *F. Support for Efficient Software-Controlled Prefetching*

Software-controlled prefetching ([10]) in a distributed processing environment is a technique to reduce memory latency by explicit (i.e., runtime system driven) execution of disk and/or network prefetch operations in order to move data (e.g., file blocks, OOC array slabs, startup/restart volumes) as close as possible to the processor which is going to need the data.

In this subsection, we describe the application of software-controlled prefetching to optimization of OOC programs.

In our approach, the compile-time knowledge about loops and other out-of-core code parts is represented by a graph structure called I/O Requirement Graph (IORG). This graph is constructed incrementally during the compilation process and written to a file at its end. The IORG is used by VIPIOS to direct the prefetching and data reorganization process.

We denote the IORG by the triple G=(N,E,s) where (N,E) is a directed graph, $s \in N$ is the start node with indegree 0, and each $n \in N$ is in the transitive cover s*. The set of nodes N consists of nodes represented in-core program parts and out-of-core ones. In the IORG, code segments are condensed into single nodes encompassing the following components:

- *Branch label.* In general, IORG structures contain patterns with multiple branches including

```
START:
     SSQ1
S1:  A(lhs1) = ... B(rhs1) ...
     SSQ2
IF:  IF (condition) THEN
         SSQ3
S2:      B(lhs2) = ... B(rhs2) ...
S3:      C(lhs3) = ... B(rhs3) ...
     ELSE
S4:      D(lhs5) = ... B(rhs5) ...
         SSQ4
S5:      E(lhs6) = ... B(rhs6) ...
                   ... A(rhs7) ...
     END IF
     SSQ5
END:
```

(a) Source program

```
START:
    CALL VIPIOS_branch('b1')
    SPMD_SSQ1
    OOC_S1
    SPMD_SSQ2
    IF (condition) THEN
       CALL VIPIOS_branch('b2')
       SPMD_SSQ3
       OOC_S2
       OOC_S3
    ELSE
       CALL VIPIOS_branch('b3')
       OOC_S4
       SPMD_SSQ4
       OOC_S5
    END IF
    CALL VIPIOS_branch('b4')
    SPMD_SSQ5
END:
```

(b) SPMD program

Fig. 5. IORG Base Information.

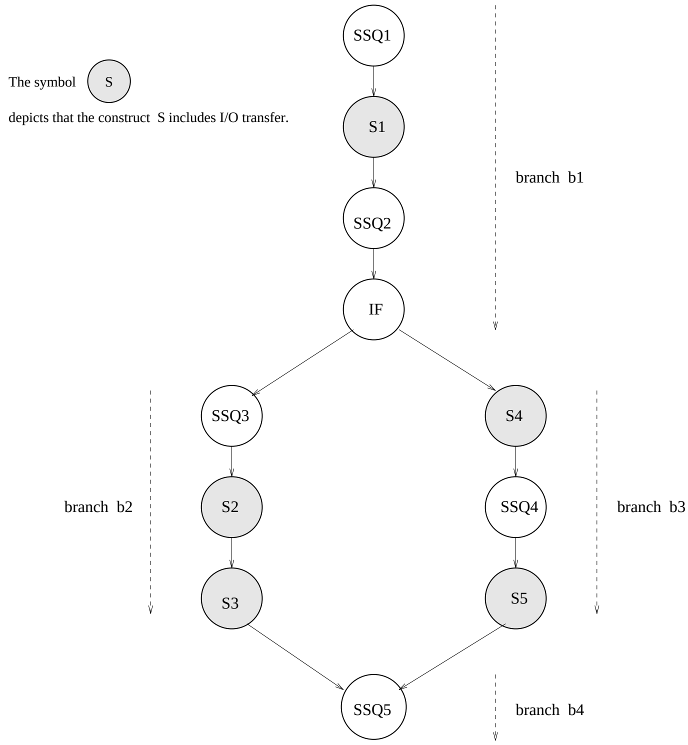


Fig. 6. Basic IORG Graph

symbolic information to be resolved at runtime. A unique label is assigned to each branch which is included in all nodes belonging to the branch. The VIPIOS is informed at runtime about the branch selection by a compiler-inserted call from the SPMD program. In the IORG these call points are annotated. When the executing program goes through the corresponding instruction sequence, the call is issued. Runtime data for the substitution of symbolic information is transferred in a similar way. Synchronisation points are determined in the context of dataflow analysis thus finding process states in which values can be bound to these symbols in the IORG.

- *Time cost estimation.*
  It estimates how much processor time is used by a program fragment represented by this node.
- *Descriptions of array sections handled by a node.*
  I/O array sections are read from or written to the so called OOC array database contained in the VIPIOS repository.

The construction of an IORG is sketched in the following example. Fig. 5.(a) introduces a fragment of the input out-of-core code, where *SSQ1–SSQ5* denote in-core code parts (statement sequences) and *S1–S5* are array statements that include out-of-core arrays *A–E*. Pseudocode of the appropriate SPMD program is introduced in Fig. 5.(b), where *SPMD_SSQi* denotes an SPMD form of *SSQi* and *OOC_Si* the out-of-core form of *Si*. In this example, the IORG consists of four branches (denoted by *b1, ..., b4)*. VIPIOS is informed about the actual selection of a particular branch by the *VIPIOS_branch* procedure call issued in the SPMD program. The corresponding IORG graph is depicted in Fig. 6 and its expanded form in Fig. 7. Edge $e_l$ points to the point of the program that constrains the movement of the prefetch for $A$ further back. With the node *S1*, for example, the following information is associated: the program part represented

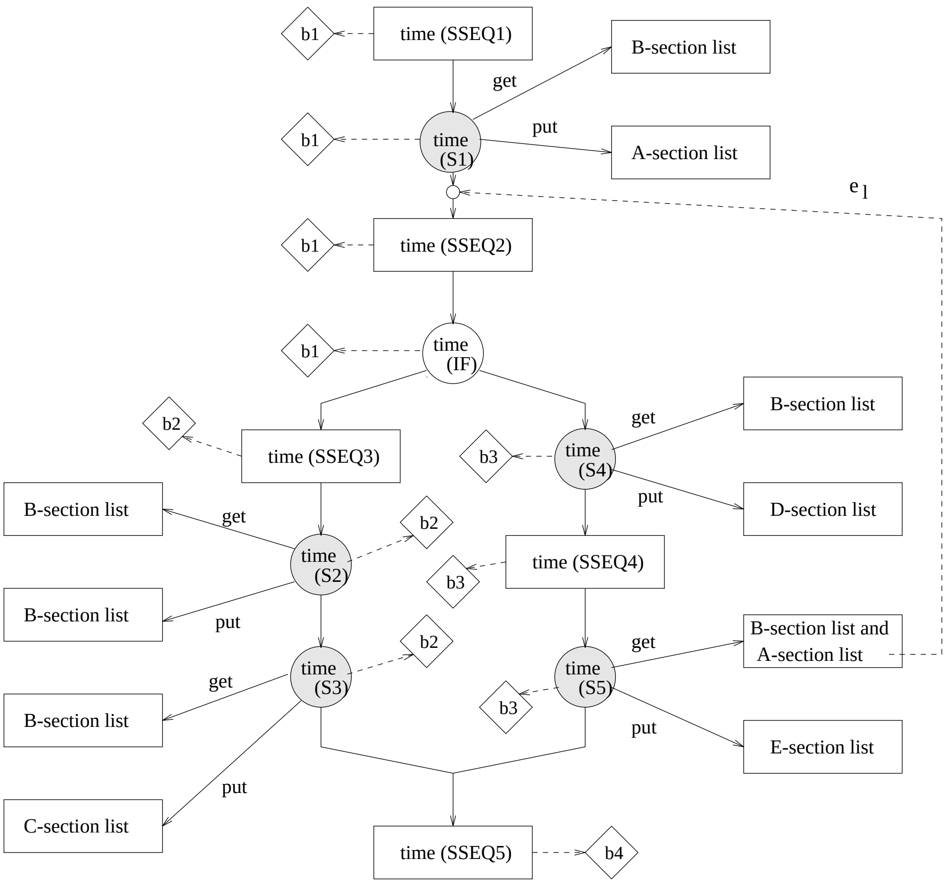


Fig. 7. Expanded IORG Graph

by this node is in branch *b1* and reads a sequence of sections of array $B$ that are denoted by *B-list* and writes a sequence of sections of array $A$ that are denoted by *A-list*. The corresponding execution time estimates are denoted by *time(S1)*.
Prefetching cost estimation is based on interconnection network distance and workload as well as I/O channel workload. Prefetching benefit estimation includes information about the probability of an actual use of prefetched items. In a control flow sequence, this probability equals 1, whereas in a control flow branch structure, the probability is less then 1. This yields to a concept commonly known as speculative (prefetching) work.

## IV. Related Work

There has been a lot of effort to improve parallel I/O performance by software means. However, many systems, for example the Intel's Concurrent File System, provide a quite unsatisfactory level of programming abstraction. Consequently, application programmers and compiler developers have to make themselves familiar with each developer's parallel I/O interface in order to deal with technical details such as buffering or physical disk layout which clearly should be hidden from the application programming level. A higher level abstraction is provided by several parallel I/O libraries like Vesta [7] PIOUS [11], PASSION [6], Jovian [3] and Galley [13]. Very little work has examined database techniques for storing multidimensional arrays on disks. Panda [14] is a server-directed I/O architecture that provides a high-level interface for array I/O.

## V. Conclusions

VIPIOS provides a new abstraction of parallel files, which is suitable for highly parallel mass storage systems. It is in the first place designed to provide support for parallelizing compilers. Four basic requirements influenced the design of VIPIOS: efficiency, parallelism, scalability and portability. Efficient data prefetching is achieved through passing the global information about the data access profile from compile time to runtime. The VIPIOS/compilation system interface presented here has been proposed in the context of VFCS, however, it can be easily integrated into any HPF compilation system.

## References


[1] S. Benkner. *Vienna Fortran 90 and its Compilation.* Ph.D. Thesis, University of Vienna, September, 1994.

[2] S. Benkner et al. *Vienna Fortran Compilation System - Version 1.2 - User's Guide*, University of Vienna, October 1995.

[3] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. *Jovian: A Framework for Optimizing Parallel I/O.* In Proceedings of the 1994 Scalable Parallel Libraries Conference, IEEE Computer Society Press, Oct. 1994

[4] P. Brezany, T. Mueck, E. Schikuta. *Language, Compiler and Parallel Database Support for I/O Intensive Applications.* Proc. HPCN Europe 1996, Milan, Italy, May 1995, Springer-Verlag, pp. 14–20.

[5] P. Brezany, T. Mück, and E. Schikuta: *Mass Storage Support for a Parallelizing Compilation System.* In Proceedings of the Conference EUROSIM – HPCN Challenges 1996, North Holland, Elsevier, June 1996.

[6] A. Choudhary et al. *PASSION: Parallel And Scalable Software for Input-Output.* Technical Report CRPC–TR94483-S, Syracuse University, Syracuse, NY, 1994.

[7] P. F. Corbett, D. G. Feitelson. *Design and Implementation of the Vesta Parallel File System.* In Proc. Scalable High Performance Computing Conference, Knoxville, May 1994, pp. 63–70.

[8] P. Corbett et al. *MPI–IO: A parallel file I/O Interface for MPI. Version 0.3.* Technical Report NAS-95-002, NAS, January 1995.

[9] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Scientific Programming, Vol. 2, No. 1, 2, 1993.

[10] T.C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching.* Ph.D.Thesis, Standford University, March 1994.

[11] S. A. Moyer, V. S. Sunderam. *PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments.* In Proc. Scalable High Performance Computing Conference, Knoxville, May 1994, pp. 71–78.

[12] MPI Forum. *MPI: A Message-Passing Interface Standard.* April 1994.

[13] N. Nieuwejaar, D. Kotz. *The Galley Parallel File System.* Proceedings of the 10th ACM International Conference on Supercomputing, May, 1996.

[14] K. E. Seamons. *Panda: Fast Access to Persistent Arrays Using High Level Interfaces and Server Directed Input/Output.* Ph.D.Thesis, University of Illinois at Urbana-Champaign, 1996.

[15] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. *Vienna Fortran – a Language Specification.* ACPC Technical Report Series, University of Vienna, Vienna, Austria, 1992.